Database Concepts

13 Data Concurrency and Consistency

This chapter explains how Oracle maintains consistent data in a multiuser database environment.

This chapter contains the following topics:

- Introduction to Data Concurrency and Consistency in a Multiuser Environment (#i5700)
- How Oracle Manages Data Concurrency and Consistency (#i5702)
- How Oracle Locks Data (#i5704)
- Overview of Oracle Flashback Query (#i20759)

Introduction to Data Concurrency and Consistency in a Multiuser Environment

In a single-user database, the user can modify data in the database without concern for other users modifying the same data at the same time. However, in a multiuser database, the statements within multiple simultaneous transactions can update the same data. Transactions executing at the same time need to produce meaningful and consistent results. Therefore, control of data concurrency and data consistency is vital in a multiuser database.

- **Data concurrency** means that many users can access data at the same time.
- **Data consistency** means that each user sees a consistent view of the data, including visible changes made by the user's own transactions and transactions of other users.

To describe consistent transaction behavior when transactions run at the same time, database researchers have defined a transaction isolation model called **serializability**. The serializable mode of transaction behavior tries to ensure that transactions run in such a way that they appear to be executed one at a time, or serially, rather than concurrently.

While this degree of isolation between transactions is generally desirable, running many applications in this mode can seriously compromise application throughput. Complete isolation of concurrently running transactions could mean that one transaction cannot perform an insert into a table being queried by another transaction. In short, real-world considerations usually require a compromise between perfect transaction isolation and performance.

Oracle offers two isolation levels, providing application developers with operational modes that preserve consistency and provide high performance.

> **See Also:**
>
> Chapter 21, "Data Integrity" (data_int.htm#g20134) for information about data integrity, which enforces business rules associated with a database

Preventable Phenomena and Transaction Isolation Levels

The ANSI/ISO SQL standard (SQL92) defines four levels of transaction isolation with differing degrees of impact on transaction processing throughput. These isolation levels are defined in terms of three phenomena that must be prevented between concurrently executing transactions.

The three preventable phenomena are:

- Dirty reads: A transaction reads data that has been written by another transaction that has not been committed yet.
- Nonrepeatable (fuzzy) reads: A transaction rereads data it has previously read and finds that another committed transaction has modified or deleted the data.
- Phantom reads (or phantoms): A transaction re-runs a query returning a set of rows that satisfies a search condition and finds that another committed transaction has inserted additional rows that satisfy the condition.

SQL92 defines four levels of isolation in terms of the phenomena a transaction running at a particular isolation level is permitted to experience. They are shown in Table 13-1 (#g35628) :

Table 13-1 Preventable Read Phenomena by Isolation Level

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read
Read uncommitted	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible
Repeatable read	Not possible	Not possible	Possible
Serializable	Not possible	Not possible	Not possible

Oracle offers the read committed and serializable isolation levels, as well as a read-only mode that is not part of SQL92. Read committed is the default.

> **See Also:**
>
> "How Oracle Manages Data Concurrency and Consistency" (#i5702) for a full discussion of read committed and serializable isolation levels

Overview of Locking Mechanisms

In general, multiuser databases use some form of data locking to solve the problems associated with data concurrency, consistency, and integrity. **Locks** are mechanisms that prevent destructive interaction between transactions accessing the same resource.

Resources include two general types of objects:

- User objects, such as tables and rows (structures and data)
- System objects not visible to users, such as shared data structures in the memory and data dictionary rows

> **See Also:**
> "How Oracle Locks Data" (#i5704) for more information about locks

How Oracle Manages Data Concurrency and Consistency

Oracle maintains data consistency in a multiuser environment by using a multiversion consistency model and various types of locks and transactions. The following topics are discussed in this section:

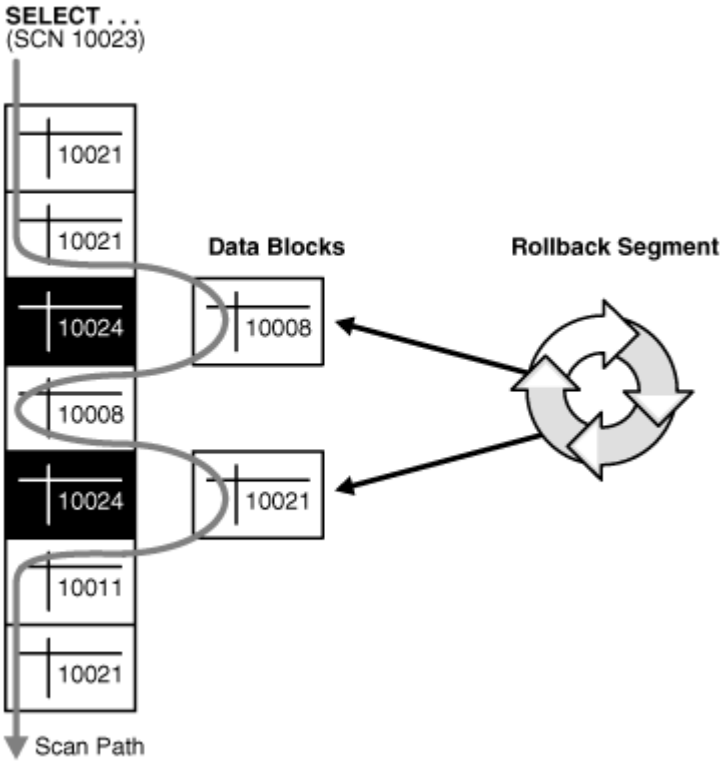
- Multiversion Concurrency Control (#i17881)
- Statement-Level Read Consistency (#i17841)
- Transaction-Level Read Consistency (#i17846)
- Read Consistency with Real Application Clusters (#i19436)
- Oracle Isolation Levels (#i17856)
- Comparison of Read Committed and Serializable Isolation (#i17894)
- Choice of Isolation Level (#i17899)

Multiversion Concurrency Control

Oracle automatically provides read consistency to a query so that all the data that the query sees comes from a single point in time **(statement-level read consistency)**. Oracle can also provide read consistency to all of the queries in a transaction **(transaction-level read consistency)**.

Oracle uses the information maintained in its rollback segments to provide these consistent views. The rollback segments contain the old values of data that have been changed by uncommitted or recently committed transactions. Figure 13-1 (#i8841) shows how Oracle provides statement-level read consistency using data in rollback segments.

Figure 13-1 Transactions and Read Consistency



Description of "Figure 13-1 Transactions and Read Consistency" (img_text/cncpt069.htm)

As a query enters the execution stage, the current system change number (SCN) is determined. In Figure 13-1 (#i8841) , this system change number is 10023. As data blocks are read on behalf of the query, only blocks written with the observed SCN are used. Blocks with changed data (more recent SCNs) are reconstructed from data in the rollback segments, and the reconstructed data is returned for the query. Therefore, each query returns all committed data with respect to the SCN recorded at the time that query execution began. Changes of other transactions that occur during a query's execution are not observed, guaranteeing that consistent data is returned for each query.

Statement-Level Read Consistency

Oracle always enforces **statement-level** read consistency. This guarantees that all the data returned by a single query comes from a single point in time—the time that the query began. Therefore, a query never sees dirty data or any of the changes made by transactions that commit during query execution. As query execution proceeds, only data committed before the query began is visible to the query. The query does not see changes committed after statement execution begins.

A consistent result set is provided for every query, guaranteeing data consistency, with no action on the user's part. The SQL statements `SELECT`, `INSERT` with a subquery, `UPDATE`, and `DELETE` all query data, either explicitly or implicitly, and all return consistent data. Each of these statements uses a query to determine which data it will affect (`SELECT`, `INSERT`, `UPDATE`, or `DELETE`, respectively).

A `SELECT` statement is an explicit query and can have nested queries or a join operation. An `INSERT` statement can use nested queries. `UPDATE` and `DELETE` statements can use `WHERE` clauses or subqueries to affect only some rows in a table rather than all rows.

Queries used in `INSERT`, `UPDATE`, and `DELETE` statements are guaranteed a consistent set of results. However, they do not see the changes made by the DML statement itself. In other words, the query in these operations sees data as it existed before the operation began to make changes.

> **Note:**

If a SELECT list contains a function, then the database applies statement-level read consistency at the statement level for SQL run within the PL/SQL function code, rather than at the parent SQL level. For example, a function could access a table whose data is changed and committed by another user. For each execution of the SELECT in the function, a new read consistent snapshot is established.

Transaction-Level Read Consistency

Oracle also offers the option of enforcing **transaction-level read consistency**. When a transaction runs in serializable mode, all data accesses reflect the state of the database as of the time the transaction began. This means that the data seen by all queries within the same transaction is consistent with respect to a single point in time, except that queries made by a serializable transaction do see changes made by the transaction itself. Transaction-level read consistency produces repeatable reads and does not expose a query to phantoms.

Read Consistency with Real Application Clusters

Real Application Clusters (RAC) use a cache-to-cache block transfer mechanism known as Cache Fusion to transfer read-consistent images of blocks from one instance to another. RAC does this using high speed, low latency interconnects to satisfy remote requests for data blocks.

See Also:

Oracle Database Oracle Clusterware and Oracle Real Application Clusters Administration and Deployment Guide (../../rac.102/b14197/toc.htm)

Oracle Isolation Levels

Oracle provides these transaction isolation levels.

Isolation Level	Description
Read committed	This is the default transaction isolation level. Each query executed by a transaction sees only data that was committed before the query (not the transaction) began. An Oracle query never reads dirty (uncommitted) data. Because Oracle does not prevent other transactions from modifying the data read by a query, that data can be changed by other transactions between two executions of the query. Thus, a transaction that runs a given query twice can experience both nonrepeatable read and phantoms.
Serializable	Serializable transactions see only those changes that were committed at the time the transaction began, plus those changes made by the transaction itself through INSERT, UPDATE, and DELETE statements. Serializable transactions do not experience nonrepeatable reads or phantoms.

Isolation Level	Description
Read-only	Read-only transactions see only those changes that were committed at the time the transaction began and do not allow `INSERT`, `UPDATE`, and `DELETE` statements.

Set the Isolation Level

Application designers, application developers, and database administrators can choose appropriate isolation levels for different transactions, depending on the application and workload. You can set the isolation level of a transaction by using one of these statements at the beginning of a transaction:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED; SET TRANSACTION ISOLATION LEVEL
SERIALIZABLE; SET TRANSACTION READ ONLY;
```

To save the networking and processing cost of beginning each transaction with a `SET TRANSACTION` statement, you can use the `ALTER SESSION` statement to set the transaction isolation level for all subsequent transactions:

```
ALTER SESSION SET ISOLATION_LEVEL SERIALIZABLE; ALTER SESSION SET ISOLATION_LEVEL READ
COMMITTED;
```

> **See Also:**
>
> *Oracle Database SQL Reference (../b14200/toc.htm)* for detailed information on any of these SQL statements

Read Committed Isolation

The default isolation level for Oracle is read committed. This degree of isolation is appropriate for environments where few transactions are likely to conflict. Oracle causes each query to run with respect to its own materialized view time, thereby permitting nonrepeatable reads and phantoms for multiple executions of a query, but providing higher potential throughput. Read committed isolation is the appropriate level of isolation for environments where few transactions are likely to conflict.

Serializable Isolation

Serializable isolation is suitable for environments:

- With large databases and short transactions that update only a few rows
- Where the chance that two concurrent transactions will modify the same rows is relatively low
- Where relatively long-running transactions are primarily read only

Serializable isolation permits concurrent transactions to make only those database changes they could have made if the transactions had been scheduled to run one after another. Specifically, Oracle permits a serializable transaction to modify a data row only if it can determine that prior changes to the row were made by transactions that had committed when the serializable transaction began.

To make this determination efficiently, Oracle uses control information stored in the data block that indicates which rows in the block contain committed and uncommitted changes. In a sense, the block contains a recent history of transactions that affected each row in the block. The amount of history that is retained is controlled by the `INITRANS` parameter of `CREATE TABLE` and `ALTER TABLE`.

Under some circumstances, Oracle can have insufficient history information to determine whether a row has been updated by a too recent transaction. This can occur when many transactions concurrently modify the same data block, or do so in a very short period. You can avoid this situation by setting higher values of `INITRANS` for tables that will experience many transactions updating the same blocks. Doing so enables Oracle to allocate sufficient storage in each block to record the history of recent transactions that accessed the block.

Oracle generates an error when a serializable transaction tries to update or delete data modified by a transaction that commits *after* the serializable transaction began:

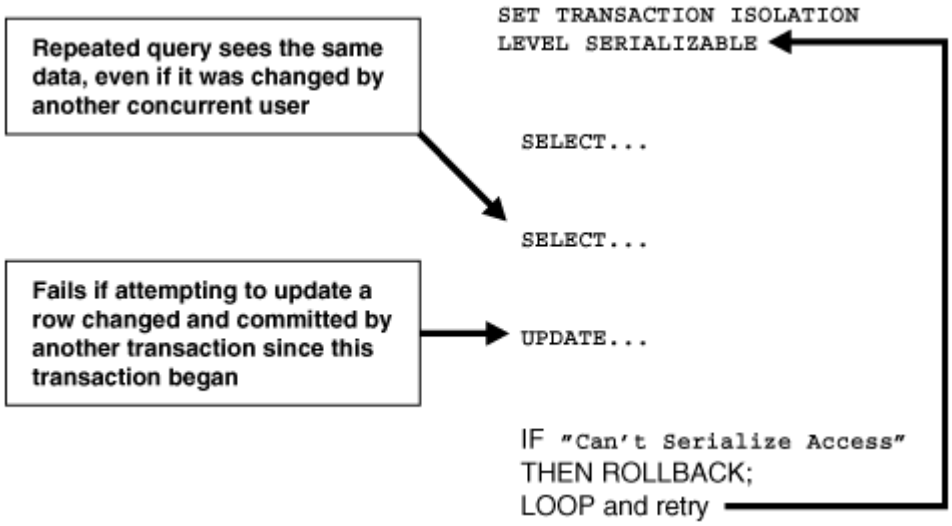
```
ORA-08177: Cannot serialize access for this transaction
```

When a serializable transaction fails with the `Cannot serialize access` error, the application can take any of several actions:

- Commit the work executed to that point
- Execute additional (but different) statements (perhaps after rolling back to a savepoint established earlier in the transaction)
- Undo the entire transaction

Figure 13-2 (#BHAGBGCH) shows an example of an application that rolls back and retries the transaction after it fails with the `Cannot serialize access` error:

Figure 13-2 Serializable Transaction Failure



Description of "Figure 13-2 Serializable Transaction Failure" (img_text/cncpt103.htm)

Comparison of Read Committed and Serializable Isolation

Oracle gives the application developer a choice of two transaction isolation levels with different characteristics. Both the read committed and serializable isolation levels provide a high degree of consistency and concurrency. Both levels provide the contention-reducing benefits of Oracle's read consistency multiversion concurrency control model and exclusive row-level locking implementation and are designed for real-world application deployment.

Transaction Set Consistency

A useful way to view the read committed and serializable isolation levels in Oracle is to consider the following scenario: Assume you have a collection of database tables (or any set of data), a particular sequence of reads of rows in those tables, and the set of transactions committed at any particular time. An operation (a query or a transaction) is **transaction set consistent** if all its reads return data written by the same set of committed transactions. An operation is not transaction set consistent if some reads reflect the changes of one set of transactions and other reads reflect changes made by other transactions. An operation that is not transaction set consistent in effect sees the database in a state that reflects no single set of committed transactions.

Oracle provides transactions executing in read committed mode with transaction set consistency for each statement. Serializable mode provides transaction set consistency for each transaction.

Table 13-2 (#g35657) summarizes key differences between read committed and serializable transactions in Oracle.

Table 13-2 Read Committed and Serializable Transactions

	Read Committed	Serializable
Dirty write	Not possible	Not possible
Dirty read	Not possible	Not possible
Nonrepeatable read	Possible	Not possible
Phantoms	Possible	Not possible
Compliant with ANSI/ISO SQL 92	Yes	Yes
Read materialized view time	Statement	Transaction
Transaction set consistency	Statement level	Transaction level
Row-level locking	Yes	Yes
Readers block writers	No	No
Writers block readers	No	No

	Read Committed	Serializable
Different-row writers block writers	No	No
Same-row writers block writers	Yes	Yes
Waits for blocking transaction	Yes	Yes
Subject to `cannot serialize access`	No	Yes
Error after blocking transaction terminates	No	No
Error after blocking transaction commits	No	Yes

Row-Level Locking

Both read committed and serializable transactions use row-level locking, and both will wait if they try to change a row updated by an uncommitted concurrent transaction. The second transaction that tries to update a given row waits for the other transaction to commit or undo and release its lock. If that other transaction rolls back, the waiting transaction, regardless of its isolation mode, can proceed to change the previously locked row as if the other transaction had not existed.

However, if the other blocking transaction commits and releases its locks, a read committed transaction proceeds with its intended update. A serializable transaction, however, fails with the error `Cannot serialize access` error, because the other transaction has committed a change that was made since the serializable transaction began.

Referential Integrity

Because Oracle does not use read locks in either read-consistent or serializable transactions, data read by one transaction can be overwritten by another. Transactions that perform database consistency checks at the application level cannot assume that the data they read will remain unchanged during the execution of the transaction even though such changes are not visible to the transaction. Database inconsistencies can result unless such application-level consistency checks are coded with this in mind, even when using serializable transactions.

> **See Also:**
>
> *Oracle Database Application Developer's Guide - Fundamentals* *(../../appdev.102/b14251/toc.htm)* for more information about referential integrity and serializable transactions

> **Note:**
>
> You can use both read committed and serializable transaction isolation levels with Real Application Clusters.

Distributed Transactions

In a distributed database environment, a given transaction updates data in multiple physical databases protected by two-phase commit to ensure all nodes or none commit. In such an environment, all servers, whether Oracle or non-Oracle, that participate in a **serializable** transaction are required to support serializable isolation mode.

If a serializable transaction tries to update data in a database managed by a server that does not support serializable transactions, the transaction receives an error. The transaction can undo and retry only when the remote server does support serializable transactions.

In contrast, **read committed** transactions can perform distributed transactions with servers that do not support serializable transactions.

See Also:

Oracle Database Administrator's Guide (../b14231/toc.htm)

Choice of Isolation Level

Application designers and developers should choose an isolation level based on application performance and consistency needs as well as application coding requirements.

For environments with many concurrent users rapidly submitting transactions, designers must assess transaction performance requirements in terms of the expected transaction arrival rate and response time demands. Frequently, for high-performance environments, the choice of isolation levels involves a trade-off between consistency and concurrency.

Application logic that checks database consistency must take into account the fact that reads do not block writes in either mode.

Oracle isolation modes provide high levels of consistency, concurrency, and performance through the combination of row-level locking and Oracle's multiversion concurrency control system. Readers and writers do not block one another in Oracle. Therefore, while queries still see consistent data, both read committed and serializable isolation provide a high level of concurrency for high performance, without the need for reading uncommitted data.

Read Committed Isolation

For many applications, read committed is the most appropriate isolation level. Read committed isolation can provide considerably more concurrency with a somewhat increased risk of inconsistent results due to phantoms and non-repeatable reads for some transactions.

Many high-performance environments with high transaction arrival rates require more throughput and faster response times than can be achieved with serializable isolation. Other environments that supports users with a very low transaction arrival rate also face very low risk of incorrect results due to phantoms and nonrepeatable reads. Read committed isolation is suitable for both of these environments.

Oracle read committed isolation provides transaction set consistency for every query. That is, every query sees data in a consistent state. Therefore, read committed isolation will suffice for many applications that might require a higher degree of isolation if run on other database management systems that do not use multiversion concurrency control.

Read committed isolation mode does not require application logic to trap the `Cannot serialize access` error and loop back to restart a transaction. In most applications, few transactions have a functional need to issue the same query twice, so for many applications protection against phantoms and non-repeatable reads is not important. Therefore many

developers choose read committed to avoid the need to write such error checking and retry code in each transaction.

Serializable Isolation

Oracle's serializable isolation is suitable for environments where there is a relatively low chance that two concurrent transactions will modify the same rows and the long-running transactions are primarily read only. It is most suitable for environments with large databases and short transactions that update only a few rows.

Serializable isolation mode provides somewhat more consistency by protecting against phantoms and nonrepeatable reads and can be important where a read/write transaction runs a query more than once.

Unlike other implementations of serializable isolation, which lock blocks for read as well as write, Oracle provides nonblocking queries and the fine granularity of row-level locking, both of which reduce read/write contention. For applications that experience mostly read/write contention, Oracle serializable isolation can provide significantly more throughput than other systems. Therefore, some applications might be suitable for serializable isolation on Oracle but not on other systems.

All queries in an Oracle serializable transaction see the database as of a single point in time, so this isolation level is suitable where multiple consistent queries must be issued in a read/write transaction. A report-writing application that generates summary data and stores it in the database might use serializable mode because it provides the consistency that a `READ ONLY` transaction provides, but also allows `INSERT`, `UPDATE`, and `DELETE`.

> **Note:**
>
> Transactions containing DML statements with subqueries should use serializable isolation to guarantee consistent read.

Coding serializable transactions requires extra work by the application developer to check for the `Cannot serialize access` error and to undo and retry the transaction. Similar extra coding is needed in other database management systems to manage deadlocks. For adherence to corporate standards or for applications that are run on multiple database management systems, it may be necessary to design transactions for serializable mode. Transactions that check for serializability failures and retry can be used with Oracle read committed mode, which does not generate serializability errors.

Serializable mode is probably not the best choice in an environment with relatively long transactions that must update the same rows accessed by a high volume of short update transactions. Because a longer running transaction is unlikely to be the first to modify a given row, it will repeatedly need to roll back, wasting work. Note that a conventional read-locking, pessimistic implementation of serializable mode would not be suitable for this environment either, because long-running transactions—even read transactions—would block the progress of short update transactions and vice versa.

Application developers should take into account the cost of rolling back and retrying transactions when using serializable mode. As with read-locking systems, where deadlocks occur frequently, use of serializable mode requires rolling back the work done by terminated transactions and retrying them. In a high contention environment, this activity can use significant resources.

In most environments, a transaction that restarts after receiving the `Cannot serialize access` error is unlikely to encounter a second conflict with another transaction. For this reason, it can help to run those statements most likely to contend with other transactions as early as possible in a serializable transaction. However, there is no guarantee that the transaction will complete successfully, so the application should be coded to limit the number of retries.

Although Oracle serializable mode is compatible with SQL92 and offers many benefits compared with read-locking implementations, it does not provide semantics identical to such systems. Application designers must take into account the fact that reads in Oracle do not block writes as they do in other systems. Transactions that check for database

consistency at the application level can require coding techniques such as the use of `SELECT FOR UPDATE`. This issue should be considered when applications using serializable mode are ported to Oracle from other environments.

Quiesce Database

You can put the system into **quiesced state**. The system is in quiesced state if there are no active sessions, other than `SYS` and `SYSTEM`. An active session is defined as a session that is currently inside a transaction, a query, a fetch or a PL/SQL procedure, or a session that is currently holding any shared resources (for example, enqueues--enqueues are shared memory structures that serialize access to database resources and are associated with a session or transaction). Database administrators are the only users who can proceed when the system is in quiesced state.

Database administrators can perform certain actions in the quiesced state that cannot be safely done when the system is not quiesced. These actions include:

- Actions that might fail if there are concurrent user transactions or queries. For example, changing the schema of a database table will fail if a concurrent transaction is accessing the same table.

- Actions whose intermediate effect could be detrimental to concurrent user transactions or queries. For example, suppose there is a big table `T` and a PL/SQL package that operates on it. You can split table `T` into two tables `T1` and `T2`, and change the PL/SQL package to make it refer to the new tables `T1` and `T2`, instead of the old table `T`.

  When the database is in quiesced state, you can do the following:

  ```
  CREATE TABLE T1 AS SELECT ... FROM T; CREATE TABLE T2 AS SELECT ... FROM T; DROP TABLE T;
  ```

  You can then drop the old PL/SQL package and re-create it.

For systems that must operate continuously, the ability to perform such actions without shutting down the database is critical.

The Database Resource Manager blocks all actions that were initiated by a user other than `SYS` or `SYSTEM` while the system is quiesced. Such actions are allowed to proceed when the system goes back to normal (unquiesced) state. Users do not get any additional error messages from the quiesced state.

How a Database Is Quiesced

The database administrator uses the `ALTER SYSTEM QUIESCE RESTRICTED` statement to quiesce the database. Only users `SYS` and `SYSTEM` can issue the `ALTER SYSTEM QUIESCE RESTRICTED` statement. For all instances with the database open, issuing this statement has the following effect:

- Oracle instructs the Database Resource Manager in all instances to prevent all inactive sessions (other than `SYS` and `SYSTEM`) from becoming active. No user other than `SYS` and `SYSTEM` can start a new transaction, a new query, a new fetch, or a new PL/SQL operation.

- Oracle waits for all existing transactions in all instances that were initiated by a user other than `SYS` or `SYSTEM` to finish (either commit or terminate). Oracle also waits for all running queries, fetches, and PL/SQL procedures in all instances that were initiated by users other than `SYS` or `SYSTEM` and that are not inside transactions to finish. If a query is carried out by multiple successive OCI fetches, Oracle does not wait for all fetches to finish. It waits for the current fetch to finish and then blocks the next fetch. Oracle also waits for all sessions (other than those of `SYS` or `SYSTEM`) that hold any shared resources (such as enqueues) to release those resources. After all these operations finish, Oracle places the database into quiesced state and finishes executing the `QUIESCE RESTRICTED` statement.

- If an instance is running in shared server mode, Oracle instructs the Database Resource Manager to block logins (other than `SYS` or `SYSTEM`) on that instance. If an instance is running in non-shared-server mode, Oracle does not impose any restrictions on user logins in that instance.

During the quiesced state, you cannot change the Resource Manager plan in any instance.

The `ALTER SYSTEM UNQUIESCE` statement puts all running instances back into normal mode, so that all blocked actions can proceed. An administrator can determine which sessions are blocking a quiesce from completing by querying the `v$blocking_quiesce` view.

> **See Also:**
>
> - *Oracle Database SQL Reference* ***(../b14200/toc.htm)***
> - *Oracle Database Administrator's Guide* ***(../b14231/toc.htm)***

How Oracle Locks Data

Locks are mechanisms that prevent destructive interaction between transactions accessing the same **resource**—either user objects such as tables and rows or system objects not visible to users, such as shared data structures in memory and data dictionary rows.

In all cases, Oracle automatically obtains necessary locks when executing SQL statements, so users need not be concerned with such details. Oracle automatically uses the lowest applicable level of restrictiveness to provide the highest degree of data concurrency yet also provide fail-safe data integrity. Oracle also allows the user to lock data manually.

> **See Also:**
>
> "Types of Locks" (#i5242)

Transactions and Data Concurrency

Oracle provides data concurrency and integrity between transactions using its locking mechanisms. Because the locking mechanisms of Oracle are tied closely to transaction control, application designers need only define transactions properly, and Oracle automatically manages locking.

Keep in mind that Oracle locking is fully automatic and requires no user action. Implicit locking occurs for all SQL statements so that database users never need to lock any resource explicitly. Oracle's default locking mechanisms lock data at the lowest level of restrictiveness to guarantee data integrity while allowing the highest degree of data concurrency.

> **See Also:**
>
> "Explicit (Manual) Data Locking" (#i5269)

Modes of Locking

Oracle uses two modes of locking in a multiuser database:

- Exclusive lock mode prevents the associates resource from being shared. This lock mode is obtained to modify data. The first transaction to lock a resource exclusively is the only transaction that can alter the resource until the exclusive lock is released.

- Share lock mode allows the associated resource to be shared, depending on the operations involved. Multiple users reading data can share the data, holding share locks to prevent concurrent access by a writer (who needs an exclusive lock). Several transactions can acquire share locks on the same resource.

Lock Duration

All locks acquired by statements within a transaction are held for the duration of the transaction, preventing destructive interference including dirty reads, lost updates, and destructive DDL operations from concurrent transactions. The changes made by the SQL statements of one transaction become visible only to other transactions that start *after* the first transaction is committed.

Oracle releases all locks acquired by the statements within a transaction when you either commit or undo the transaction. Oracle also releases locks acquired after a savepoint when rolling back to the savepoint. However, only transactions not waiting for the previously locked resources can acquire locks on the now available resources. Waiting transactions will continue to wait until after the original transaction commits or rolls back completely.

Data Lock Conversion Versus Lock Escalation

A transaction holds exclusive row locks for all rows inserted, updated, or deleted within the transaction. Because row locks are acquired at the highest degree of restrictiveness, no lock conversion is required or performed.

Oracle automatically converts a table lock of lower restrictiveness to one of higher restrictiveness as appropriate. For example, assume that a transaction uses a `SELECT` statement with the `FOR UPDATE` clause to lock rows of a table. As a result, it acquires the exclusive row locks and a row share table lock for the table. If the transaction later updates one or more of the locked rows, the row share table lock is automatically converted to a row exclusive table lock.

Lock escalation occurs when numerous locks are held at one level of granularity (for example, rows) and a database raises the locks to a higher level of granularity (for example, table). For example, if a single user locks many rows in a table, some databases automatically escalate the user's row locks to a single table. The number of locks is reduced, but the restrictiveness of what is being locked is increased.

Oracle never escalates locks. Lock escalation greatly increases the likelihood of deadlocks. Imagine the situation where the system is trying to escalate locks on behalf of transaction T1 but cannot because of the locks held by transaction T2. A deadlock is created if transaction T2 also requires lock escalation of the same data before it can proceed.

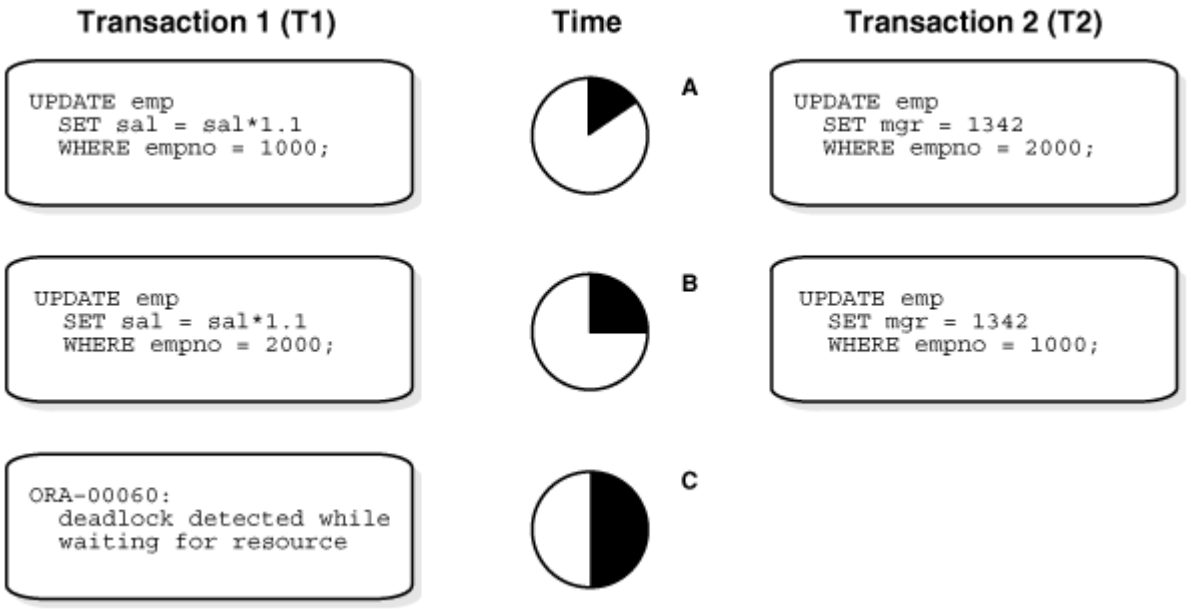
> **See Also:**
>
> "Table Locks (TM)" (#i5249)

Deadlocks

A **deadlock** can occur when two or more users are waiting for data locked by each other. Deadlocks prevent some transactions from continuing to work. Figure 13-3 (#i6885) is a hypothetical illustration of two transactions in a deadlock.

In Figure 13-3 (#i6885) , no problem exists at time point A, as each transaction has a row lock on the row it attempts to update. Each transaction proceeds without being terminated. However, each tries next to update the row currently held by the other transaction. Therefore, a deadlock results at time point B, because neither transaction can obtain the resource it needs to proceed or terminate. It is a deadlock because no matter how long each transaction waits, the conflicting locks are held.

Figure 13-3 Two Transactions in a Deadlock



Description of "Figure 13-3 Two Transactions in a Deadlock" (img_text/cncpt068.htm)

Deadlock Detection

Oracle automatically detects deadlock situations and resolves them by rolling back one of the statements involved in the deadlock, thereby releasing one set of the conflicting row locks. A corresponding message also is returned to the transaction that undergoes statement-level rollback. The statement rolled back is the one belonging to the transaction that detects the deadlock. Usually, the signalled transaction should be rolled back explicitly, but it can retry the rolled-back statement after waiting.

> **Note:**
>
> In distributed transactions, local deadlocks are detected by analyzing wait data, and global deadlocks are detected by a time out. Once detected, nondistributed and distributed deadlocks are handled by the database and application in the same way.

Deadlocks most often occur when transactions explicitly override the default locking of Oracle. Because Oracle itself does no lock escalation and does not use read locks for queries, but does use row-level locking (rather than page-level locking), deadlocks occur infrequently in Oracle.

> **See Also:**
>
> "Explicit (Manual) Data Locking" (#i5269) for more information about manually acquiring locks

Avoid Deadlocks

Multitable deadlocks can usually be avoided if transactions accessing the same tables lock those tables in the same order, either through implicit or explicit locks. For example, all application developers might follow the rule that when both a master and detail table are updated, the master table is locked first and then the detail table. If such rules are properly designed and then followed in all applications, deadlocks are very unlikely to occur.

When you know you will require a sequence of locks for one transaction, consider acquiring the most exclusive (least compatible) lock first.

Types of Locks

Oracle automatically uses different types of locks to control concurrent access to data and to prevent destructive interaction between users. Oracle automatically locks a resource on behalf of a transaction to prevent other transactions from doing something also requiring exclusive access to the same resource. The lock is released automatically when some event occurs so that the transaction no longer requires the resource.

Throughout its operation, Oracle automatically acquires different types of locks at different levels of restrictiveness depending on the resource being locked and the operation being performed.

Oracle locks fall into one of three general categories.

Lock	Description
DML locks (data locks)	DML locks protect data. For example, table locks lock entire tables, row locks lock selected rows.
DDL locks (dictionary locks)	DDL locks protect the structure of schema objects—for example, the definitions of tables and views.
Internal locks and latches	Internal locks and latches protect internal database structures such as datafiles. Internal locks and latches are entirely automatic.

The following sections discuss DML locks, DDL locks, and internal locks.

DML Locks

The purpose of a DML lock (data lock) is to guarantee the integrity of data being accessed concurrently by multiple users. DML locks prevent destructive interference of simultaneous conflicting DML or DDL operations. DML statements automatically acquire both table-level locks and row-level locks.

> **Note:**
> The acronym in parentheses after each type of lock or lock mode is the abbreviation used in the Locks Monitor of Enterprise Manager. Enterprise Manager might display TM for any table lock, rather than indicate the mode of table lock (such as RS or SRX).

Row Locks (TX)

Row-level locks are primarily used to prevent two transactions from modifying the same row. When a transaction needs to modify a row, a row lock is acquired.

There is no limit to the number of row locks held by a statement or transaction, and Oracle does not escalate locks from the row level to a coarser granularity. Row locking provides the finest grain locking possible and so provides the best possible concurrency and throughput.

The combination of multiversion concurrency control and row-level locking means that users contend for data only when accessing the same rows, specifically:

- Readers of data do not wait for writers of the same data rows.

- Writers of data do not wait for readers of the same data rows unless `SELECT ... FOR UPDATE` is used, which specifically requests a lock for the reader.

- Writers only wait for other writers if they attempt to update the same rows at the same time.

> **Note:**
>
> Readers of data may have to wait for writers of the same data blocks in some very special cases of pending distributed transactions.

A transaction acquires an exclusive row lock for each individual row modified by one of the following statements: `INSERT`, `UPDATE`, `DELETE`, and `SELECT` with the `FOR UPDATE` clause.

A modified row is **always** locked exclusively so that other transactions cannot modify the row until the transaction holding the lock is committed or rolled back. However, if the transaction dies due to instance failure, block-level recovery makes a row available before the entire transaction is recovered. Row locks are always acquired automatically by Oracle as a result of the statements listed previously.

If a transaction obtains a row lock for a row, the transaction also acquires a table lock for the corresponding table. The table lock prevents conflicting DDL operations that would override data changes in a current transaction.

> **See Also:**
>
> "DDL Locks" (#i5281)

Table Locks (TM)

Table-level locks are primarily used to do concurrency control with concurrent DDL operations, such as preventing a table from being dropped in the middle of a DML operation. When a DDL or DML statement is on a table, a table lock is acquired. Table locks do not affect concurrency of DML operations. For partitioned tables, table locks can be acquired at both the table and the subpartition level.

A transaction acquires a table lock when a table is modified in the following DML statements: `INSERT`, `UPDATE`, `DELETE`, `SELECT` with the `FOR UPDATE` clause, and `LOCK TABLE`. These DML operations require table locks for two purposes: to reserve DML access to the table on behalf of a transaction and to prevent DDL operations that would conflict with the transaction. Any table lock prevents the acquisition of an exclusive DDL lock on the same table and thereby prevents DDL operations that require such locks. For example, a table cannot be altered or dropped if an uncommitted transaction holds a table lock for it.

A table lock can be held in any of several modes: row share (RS), row exclusive (RX), share (S), share row exclusive (SRX), and exclusive (X). The restrictiveness of a table lock's mode determines the modes in which other table locks on the same table can be obtained and held.

Table 13-3 (#BABCJIAJ) shows the table lock modes that statements acquire and operations that those locks permit and prohibit.

Table 13-3 Summary of Table Locks

SQL Statement	Mode of Table Lock	Lock Modes Permitted?				
		RS	RX	S	SRX	X

SQL Statement	Mode of Table Lock	Lock Modes Permitted?				
		RS	RX	S	SRX	X
`SELECT...FROM table...`	none	Y	Y	Y	Y	Y
`INSERT INTO table ...`	RX	Y	Y	N	N	N
`UPDATE table ...`	RX	Y*	Y*	N	N	N
`DELETE FROM table ...`	RX	Y*	Y*	N	N	N
`SELECT ... FROM table FOR UPDATE OF ...`	RS	Y*	Y*	Y*	Y*	N
`LOCK TABLE table IN ROW SHARE MODE`	RS	Y	Y	Y	Y	N
`LOCK TABLE table IN ROW EXCLUSIVE MODE`	RX	Y	Y	N	N	N
`LOCK TABLE table IN SHARE MODE`	S	Y	N	Y	N	N

SQL Statement	Mode of Table Lock	Lock Modes Permitted?				
		RS	RX	S	SRX	X
`LOCK TABLE table IN SHARE ROW EXCLUSIVE MODE`	SRX	Y	N	N	N	N
`LOCK TABLE table IN EXCLUSIVE MODE`	X	N	N	N	N	N

RS: row share

RX: row exclusive

S: share

SRX: share row exclusive

X: exclusive

*Yes, if no conflicting row locks are held by another transaction. Otherwise, waits occur.

The following sections explain each mode of table lock, from least restrictive to most restrictive. They also describe the actions that cause the transaction to acquire a table lock in that mode and which actions are permitted and prohibited in other transactions by a lock in that mode.

> **See Also:**
>
> "Explicit (Manual) Data Locking" (#i5269)

Row Share Table Locks (RS)

A row share table lock (also sometimes called a **subshare table lock, SS**) indicates that the transaction holding the lock on the table has locked rows in the table and intends to update them. A row share table lock is automatically acquired for a *table* when one of the following SQL statements is run:

```
SELECT ... FROM table ... FOR UPDATE OF ... ; LOCK TABLE table IN ROW SHARE MODE;
```

A row share table lock is the least restrictive mode of table lock, offering the highest degree of concurrency for a table.

Permitted Operations: A row share table lock held by a transaction allows other transactions to query, insert, update, delete, or lock rows concurrently in the same table. Therefore, other transactions can obtain simultaneous row share, row exclusive, share, and share row exclusive table locks for the same table.

Prohibited Operations: A row share table lock held by a transaction prevents other transactions from exclusive write access to the same table using only the following statement:

```
LOCK TABLE table IN EXCLUSIVE MODE;
```

Row Exclusive Table Locks (RX)

A row exclusive table lock (also called a **subexclusive table lock, SX**) generally indicates that the transaction holding the lock has made one or more updates to rows in the table. A row exclusive table lock is acquired automatically for a *table* modified by the following types of statements:

```
INSERT INTO table ... ; UPDATE table ... ; DELETE FROM table ... ; LOCK TABLE table IN ROW EXCLUSIVE MODE;
```

A row exclusive table lock is slightly more restrictive than a row share table lock.

Permitted Operations: A row exclusive table lock held by a transaction allows other transactions to query, insert, update, delete, or lock rows concurrently in the same table. Therefore, row exclusive table locks allow multiple transactions to obtain simultaneous row exclusive and row share table locks for the same table.

Prohibited Operations: A row exclusive table lock held by a transaction prevents other transactions from manually locking the table for exclusive reading or writing. Therefore, other transactions cannot concurrently lock the table using the following statements:

```
LOCK TABLE table IN SHARE MODE; LOCK TABLE table IN SHARE EXCLUSIVE MODE; LOCK TABLE table IN EXCLUSIVE MODE;
```

Share Table Locks (S)

A share table lock is acquired automatically for the *table* specified in the following statement:

```
LOCK TABLE table IN SHARE MODE;
```

Permitted Operations: A share table lock held by a transaction allows other transactions only to query the table, to lock specific rows with `SELECT ... FOR UPDATE`, or to run `LOCK TABLE ... IN SHARE MODE` statements successfully. No updates are allowed by other transactions. Multiple transactions can hold share table locks for the same table concurrently. In this case, no transaction can update the table (even if a transaction holds row locks as the result of a `SELECT` statement with the `FOR UPDATE` clause). Therefore, a transaction that has a share table lock can update the table only if no other transactions also have a share table lock on the same table.

Prohibited Operations: A share table lock held by a transaction prevents other transactions from modifying the same table and from executing the following statements:

```
LOCK TABLE table IN SHARE ROW EXCLUSIVE MODE; LOCK TABLE table IN EXCLUSIVE MODE; LOCK TABLE table IN ROW EXCLUSIVE MODE;
```

Share Row Exclusive Table Locks (SRX)

A share row exclusive table lock (also sometimes called a **share-subexclusive table lock, SSX**) is more restrictive than a share table lock. A share row exclusive table lock is acquired for a *table* as follows:

```
LOCK TABLE table IN SHARE ROW EXCLUSIVE MODE;
```

Permitted Operations: Only one transaction at a time can acquire a share row exclusive table lock on a given table. A share row exclusive table lock held by a transaction allows other transactions to query or lock specific rows using `SELECT` with the `FOR UPDATE` clause, but not to update the table.

Prohibited Operations: A share row exclusive table lock held by a transaction prevents other transactions from obtaining row exclusive table locks and modifying the same table. A share row exclusive table lock also prohibits other transactions from obtaining share, share row exclusive, and exclusive table locks, which prevents other transactions from executing the following statements:

```
LOCK TABLE table IN SHARE MODE; LOCK TABLE table IN SHARE ROW EXCLUSIVE MODE; LOCK TABLE
table IN ROW EXCLUSIVE MODE; LOCK TABLE table IN EXCLUSIVE MODE;
```

Exclusive Table Locks (X)

An exclusive table lock is the most restrictive mode of table lock, allowing the transaction that holds the lock exclusive write access to the table. An exclusive table lock is acquired for a *table* as follows:

```
LOCK TABLE table IN EXCLUSIVE MODE;
```

Permitted Operations: Only one transaction can obtain an exclusive table lock for a table. An exclusive table lock permits other transactions only to query the table.

Prohibited Operations: An exclusive table lock held by a transaction prohibits other transactions from performing any type of DML statement or placing any type of lock on the table.

DML Locks Automatically Acquired for DML Statements

The previous sections explained the different types of data locks, the modes in which they can be held, when they can be obtained, when they are obtained, and what they prohibit. The following sections summarize how Oracle automatically locks data on behalf of different DML operations.

Table 13-4 (#g35741) summarizes the information in the following sections.

Table 13-4 Locks Obtained By DML Statements

DML Statement	Row Locks?	Mode of Table Lock
`SELECT ... FROM table`		
`INSERT INTO table ...`	X	RX
`UPDATE table ...`	X	RX
`DELETE FROM table ...`	X	RX
`SELECT ... FROM table ... FOR UPDATE OF ...`	X	RS

DML Statement	Row Locks?	Mode of Table Lock
`LOCK TABLE table IN ...`		
`ROW SHARE MODE`		RS
`ROW EXCLUSIVE MODE`		RX
`SHARE MODE`		S
`SHARE EXCLUSIVE MODE`		SRX
`EXCLUSIVE MODE`		X

X: exclusive

RX: row exclusive

RS: row share

S: share

SRX: share row exclusive

Default Locking for Queries

Queries are the SQL statements least likely to interfere with other SQL statements because they only read data. `INSERT`, `UPDATE`, and `DELETE` statements can have implicit queries as part of the statement. Queries include the following kinds of statements:

```
SELECT INSERT ... SELECT ... ; UPDATE ... ; DELETE ... ;
```

They do **not** include the following statement:

```
SELECT ... FOR UPDATE OF ... ;
```

The following characteristics are true of all queries that do not use the `FOR UPDATE` clause:

- A query acquires no data locks. Therefore, other transactions can query and update a table being queried, including the specific rows being queried. Because queries lacking `FOR UPDATE` clauses do not acquire any data locks to block other operations, such queries are often referred to in Oracle as **nonblocking queries**.
- A query does not have to wait for any data locks to be released; it can always proceed. (Queries may have to wait for data locks in some very specific cases of pending distributed transactions.)

Default Locking for INSERT, UPDATE, DELETE, and SELECT ... FOR UPDATE

The locking characteristics of `INSERT`, `UPDATE`, `DELETE`, and `SELECT ... FOR UPDATE` statements are as follows:

- The transaction that contains a DML statement acquires exclusive row locks on the rows modified by the statement. Other transactions cannot update or delete the locked rows until the locking transaction either commits or rolls back.
- The transaction that contains a DML statement does not need to acquire row locks on any rows selected by a subquery or an implicit query, such as a query in a `WHERE` clause. A subquery or implicit query in a DML statement is guaranteed to be consistent as of the start of the query and does not see the effects of the DML statement it is part of.
- A query in a transaction can see the changes made by previous DML statements in the same transaction, but cannot see the changes of other transactions begun after its own transaction.
- In addition to the necessary exclusive row locks, a transaction that contains a DML statement acquires at least a row exclusive table lock on the table that contains the affected rows. If the containing transaction already holds a share, share row exclusive, or exclusive table lock for that table, the row exclusive table lock is not acquired. If the containing transaction already holds a row share table lock, Oracle automatically converts this lock to a row exclusive table lock.

DDL Locks

A data dictionary lock (DDL) protects the definition of a schema object while that object is acted upon or referred to by an ongoing DDL operation. Recall that a DDL statement implicitly commits its transaction. For example, assume that a user creates a procedure. On behalf of the user's single-statement transaction, Oracle automatically acquires DDL locks for all schema objects referenced in the procedure definition. The DDL locks prevent objects referenced in the procedure from being altered or dropped before the procedure compilation is complete.

Oracle acquires a dictionary lock automatically on behalf of any DDL transaction requiring it. Users cannot explicitly request DDL locks. Only individual schema objects that are modified or referenced are locked during DDL operations. The whole data dictionary is never locked.

DDL locks fall into three categories: exclusive DDL locks, share DDL locks, and breakable parse locks.

Exclusive DDL Locks

Most DDL operations, except for those listed in the section, "Share DDL Locks" (#CIHIGAFG) require exclusive DDL locks for a resource to prevent destructive interference with other DDL operations that might modify or reference the same schema object. For example, a `DROP TABLE` operation is not allowed to drop a table while an `ALTER TABLE` operation is adding a column to it, and vice versa.

During the acquisition of an exclusive DDL lock, if another DDL lock is already held on the schema object by another operation, the acquisition waits until the older DDL lock is released and then proceeds.

DDL operations also acquire DML locks (data locks) on the schema object to be modified.

Share DDL Locks

Some DDL operations require share DDL locks for a resource to prevent destructive interference with conflicting DDL operations, but allow data concurrency for similar DDL operations. For example, when a `CREATE PROCEDURE` statement is run, the containing transaction acquires share DDL locks for all referenced tables. Other transactions can concurrently create procedures that reference the same tables and therefore acquire concurrent share DDL locks on the same tables,

but no transaction can acquire an exclusive DDL lock on any referenced table. No transaction can alter or drop a referenced table. As a result, a transaction that holds a share DDL lock is guaranteed that the definition of the referenced schema object will remain constant for the duration of the transaction.

A share DDL lock is acquired on a schema object for DDL statements that include the following statements: `AUDIT`, `NOAUDIT`, `COMMENT`, `CREATE [OR REPLACE] VIEW/ PROCEDURE/PACKAGE/PACKAGE BODY/FUNCTION/ TRIGGER`, `CREATE SYNONYM`, and `CREATE TABLE` (when the `CLUSTER` parameter is not included).

Breakable Parse Locks

A SQL statement (or PL/SQL program unit) in the shared pool holds a parse lock for each schema object it references. Parse locks are acquired so that the associated shared SQL area can be invalidated if a referenced object is altered or dropped. A parse lock does not disallow any DDL operation and can be broken to allow conflicting DDL operations, hence the name **breakable parse lock**.

A parse lock is acquired during the parse phase of SQL statement execution and held as long as the shared SQL area for that statement remains in the shared pool.

> **See Also:**
>
> Chapter 6, "Dependencies Among Schema Objects" (depend.htm#g9236)

Duration of DDL Locks

The duration of a DDL lock depends on its type. Exclusive and share DDL locks last for the duration of DDL statement execution and automatic commit. A parse lock persists as long as the associated SQL statement remains in the shared pool.

DDL Locks and Clusters

A DDL operation on a cluster acquires exclusive DDL locks on the cluster and on all tables and materialized views in the cluster. A DDL operation on a table or materialized view in a cluster acquires a share lock on the cluster, in addition to a share or exclusive DDL lock on the table or materialized view. The share DDL lock on the cluster prevents another operation from dropping the cluster while the first operation proceeds.

Latches and Internal Locks

Latches and internal locks protect internal database and memory structures. Both are inaccessible to users, because users have no need to control over their occurrence or duration. The following section helps to interpret the Enterprise Manager `LOCKS` and `LATCHES` monitors.

Latches

Latches are simple, low-level serialization mechanisms to protect shared data structures in the system global area (SGA). For example, latches protect the list of users currently accessing the database and protect the data structures describing the blocks in the buffer cache. A server or background process acquires a latch for a very short time while manipulating or looking at one of these structures. The implementation of latches is operating system dependent, particularly in regard to whether and how long a process will wait for a latch.

Internal Locks

Internal locks are higher-level, more complex mechanisms than latches and serve a variety of purposes.

Dictionary Cache Locks

These locks are of very short duration and are held on entries in dictionary caches while the entries are being modified or used. They guarantee that statements being parsed do not see inconsistent object definitions.

Dictionary cache locks can be shared or exclusive. Shared locks are released when the parse is complete. Exclusive locks are released when the DDL operation is complete.

File and Log Management Locks

These locks protect various files. For example, one lock protects the control file so that only one process at a time can change it. Another lock coordinates the use and archiving of the redo log files. Datafiles are locked to ensure that multiple instances mount a database in shared mode or that one instance mounts it in exclusive mode. Because file and log locks indicate the status of files, these locks are necessarily held for a long time.

Tablespace and Rollback Segment Locks

These locks protect tablespaces and rollback segments. For example, all instances accessing a database must agree on whether a tablespace is online or offline. Rollback segments are locked so that only one instance can write to a segment.

Explicit (Manual) Data Locking

Oracle always performs locking automatically to ensure data concurrency, data integrity, and statement-level read consistency. However, you can override the Oracle default locking mechanisms. Overriding the default locking is useful in situations such as these:

- Applications require transaction-level read consistency or **repeatable reads**. In other words, queries in them must produce consistent data for the duration of the transaction, not reflecting changes by other transactions. You can achieve transaction-level read consistency by using explicit locking, read-only transactions, serializable transactions, or by overriding default locking.

- Applications require that a transaction have exclusive access to a resource so that the transaction does not have to wait for other transactions to complete.

Oracle's automatic locking can be overridden at the transaction level or the session level.

At the transaction level, transactions that include the following SQL statements override Oracle's default locking:

- The `SET TRANSACTION ISOLATION LEVEL` statement

- The `LOCK TABLE` statement (which locks either a table or, when used with views, the underlying base tables)

- The `SELECT ... FOR UPDATE` statement

Locks acquired by these statements are released after the transaction commits or rolls back.

At the session level, a session can set the required transaction isolation level with the `ALTER SESSION` statement.

> **Note:**
>
> If Oracle's default locking is overridden at any level, the database administrator or application developer should ensure that the overriding locking procedures operate correctly. The locking procedures must satisfy the following criteria: data integrity is guaranteed, data concurrency is acceptable, and deadlocks are not possible or are appropriately handled.

> **See Also:**
>
> *Oracle Database SQL Reference (../b14200/toc.htm)* for detailed descriptions of the SQL statements `LOCK TABLE` and `SELECT ... FOR UPDATE`

Oracle Lock Management Services

With Oracle Lock Management services, an application developer can include statements in PL/SQL blocks that:

- Request a lock of a specific type
- Give the lock a unique name recognizable in another procedure in the same or in another instance
- Change the lock type
- Release the lock

Because a reserved user lock is the same as an Oracle lock, it has all the Oracle lock functionality including deadlock detection. User locks never conflict with Oracle locks, because they are identified with the prefix `UL`.

The Oracle Lock Management services are available through procedures in the `DBMS_LOCK` package.

> **See Also:**
>
> - *Oracle Database Application Developer's Guide - Fundamentals (../../appdev.102/b14251/toc.htm)* for more information about Oracle Lock Management services
> - *Oracle Database PL/SQL Packages and Types Reference (../../appdev.102/b14258/toc.htm)* for information about `DBMS_LOCK`

Overview of Oracle Flashback Query

Oracle Flashback Query lets you view and repair historical data. You can perform queries on the database as of a certain wall clock time or user-specified system change number (SCN).

Flashback Query uses Oracle's multiversion read-consistency capabilities to restore data by applying undo as needed. Oracle Database 10*g* automatically tunes a parameter called the undo retention period. The undo retention period indicates the amount of time that must pass before old undo information—that is, undo information for committed transactions—can be overwritten. The database collects usage statistics and tunes the undo retention period based on these statistics and on undo tablespace size.

Using Flashback Query, you can query the database as it existed this morning, yesterday, or last week. The speed of this operation depends only on the amount of data being queried and the number of changes to the data that need to be backed out.

You can query the history of a given row or a transaction. Using undo data stored in the database, you can view all versions of a row and revert to a previous version of that row. Flashback Transaction Query history lets you examine changes to the database at the transaction level.

You can audit the rows of a table and get information about the transactions that changed the rows and the times when it was changed. With the transaction ID, you can do transaction mining through LogMiner to get complete information about the transaction.

See Also:

- *Oracle Database Administrator's Guide* (../b14231/toc.htm) for more information on the automatic tuning of undo retention and on LogMiner
- "Automatic Undo Retention" (logical.htm#CIHCHAIF)

You set the date and time you want to view. Then, any SQL query you run operates on data as it existed at that time. If you are an authorized user, then you can correct errors and back out the restored data without needing the intervention of an administrator.

With the `AS OF` SQL clause, you can choose different snapshots for each table in the query. Associating a snapshot with a table is known as *table decoration*. If you do not decorate a table with a snapshot, then a default snapshot is used for it. All tables without a specified snapshot get the same default snapshot.

For example, suppose you want to write a query to find all the new customer accounts created in the past hour. You could do set operations on two instances of the same table decorated with different `AS OF` clauses.

DML and DDL operations can use table decoration to choose snapshots within subqueries. Operations such as `INSERT TABLE AS SELECT` and `CREATE TABLE AS SELECT` can be used with table decoration in the subqueries to repair tables from which rows have been mistakenly deleted. Table decoration can be any arbitrary expression: a bind variable, a constant, a string, date operations, and so on. You can open a cursor and dynamically bind a snapshot value (a timestamp or an SCN) to decorate a table with.

See Also:

- "Overview of Oracle Flashback Features" (high_av.htm#i36677) for an overview of all Oracle Flashback features
- *Oracle Database SQL Reference* (../b14200/toc.htm) for information on the `AS OF` clause

Flashback Query Benefits

This section lists some of the benefits of using Flashback Query.

- Application Transparency

  Packaged applications, like report generation tools that only do queries, can run in Flashback Query mode by using logon triggers. Applications can run transparently without requiring changes to code. All the constraints that the application needs to be satisfied are guaranteed to hold good, because there is a consistent version of the database as of the Flashback Query time.

- Application Performance

If an application requires recovery actions, it can do so by saving SCNs and flashing back to those SCNs. This is a lot easier and faster than saving data sets and restoring them later, which would be required if the application were to do explicit versioning. Using Flashback Query, there are no costs for logging that would be incurred by explicit versioning.

- Online Operation

  Flashback Query is an online operation. Concurrent DMLs and queries from other sessions are allowed while an object is queried inside Flashback Query.The speed of these operations is unaffected. Moreover, different sessions can flash back to different Flashback times or SCNs on the same object concurrently. The speed of the Flashback Query itself depends on the amount of undo that needs to be applied, which is proportional to how far back in time the query goes.

- Easy Manageability

  There is no additional management on the part of the user, except setting the appropriate retention interval, having the right privileges, and so on. No additional logging has to be turned on, because past versions are constructed automatically, as needed.

> **Note:**
>
> - Flashback Query does *not* undo anything. It is only a query mechanism. You can take the output from a Flashback Query and perform an undo yourself in many circumstances.
> - Flashback Query does *not* tell you what changed. LogMiner does that.
> - Flashback Query can undo changes and can be very efficient if you know the rows that need to be moved back in time. You can use it to move a full table back in time, but this is very expensive if the table is large since it involves a full table copy.
> - Flashback Query does not work through DDL operations that modify columns, or drop or truncate tables.
> - LogMiner is very good for getting change history, but it gives you changes in terms of deltas (insert, update, delete), not in terms of the before and after image of a row. These can be difficult to deal with in some applications.

Some Uses of Flashback Query

This section lists some ways to use Flashback Query.

- Self-Service Repair

  Perhaps you accidentally deleted some important rows from a table and wanted to recover the deleted rows. To do the repair, you can move backward in time and see the missing rows and re-insert the deleted row into the current table.

- E-mail or Voice Mail Applications

  You might have deleted mail in the past. Using Flashback Query, you can restore the deleted mail by moving back in time and re-inserting the deleted message into the current message box.

- Account Balances

You can view account prior account balances as of a certain day in the month.

- Packaged Applications

  Packaged applications (like report generation tools) can make use of Flashback Query without any changes to application logic. Any constraints that the application expects are guaranteed to be satisfied, because users see a consistent version of the Database as of the given time or SCN.

In addition, Flashback Query could be used after examination of audit information to see the before-image of the data. In DSS environments, it could be used for extraction of data as of a consistent point in time from OLTP systems.

See Also:

- *Oracle Database Application Developer's Guide - Fundamentals* ***(../../appdev.102/b14251/toc.htm)*** for more information about using Oracle Flashback Query
- *Oracle Database PL/SQL Packages and Types Reference* ***(../../appdev.102/b14258/toc.htm)*** for a description of the `DBMS_FLASHBACK` package
- *Oracle Database Administrator's Guide* ***(../b14231/toc.htm)*** for information about undo tablespaces and setting retention period

/