Part VII

Data Protection

Part VII describes how Oracle protects the data in a database and explains what the database administrator can do to provide additional protection for data.

Part VII contains the following chapters:

20

Data Concurrency and Consistency

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

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 execute 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 execute 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" 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: A transaction re-executes 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 20- 1:

Table 20- 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" on page 20-4 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" on page 20-17 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:

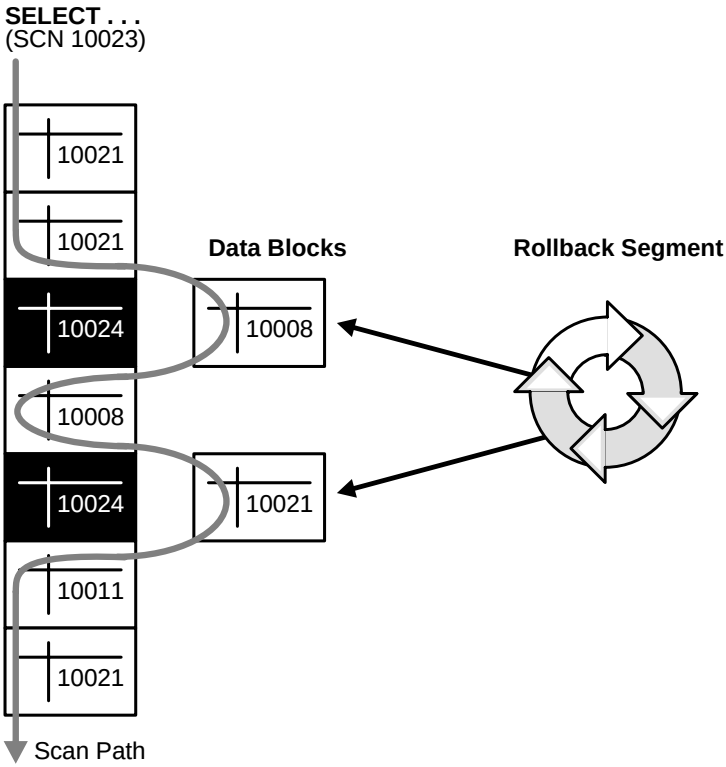
- Multiversion Concurrency Control
- Statement-Level Read Consistency
- Transaction-Level Read Consistency
- Read Consistency with Real Application Clusters
- Oracle Isolation Levels
- Comparison of Read Committed and Serializable Isolation
- Choice of Isolation Level

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 20- 1 shows how Oracle provides statement-level read consistency using data in rollback segments.

Figure 20- 1 Transactions and Read Consistency



As a query enters the execution stage, the current system change number (SCN) is determined. In Figure 20- 1, 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 nor 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.

Transaction-Level Read Consistency

Oracle also offers the option of enforcing transaction-level read consistency. When a transaction executes 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 use a cache-to-cache block transfer mechanism known as Cache Fusion to transfer read-consistent images of blocks from one instance to another. Real Application Clusters does this using high speed, low latency interconnects to satisfy remote requests for data blocks.

> **See Also:** *Oracle9i Real Application Clusters Concepts* for more information

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 executes 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.
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 ISOLATION LEVEL 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:** *Oracle9i SQL Reference* 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 execute 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 execute 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:

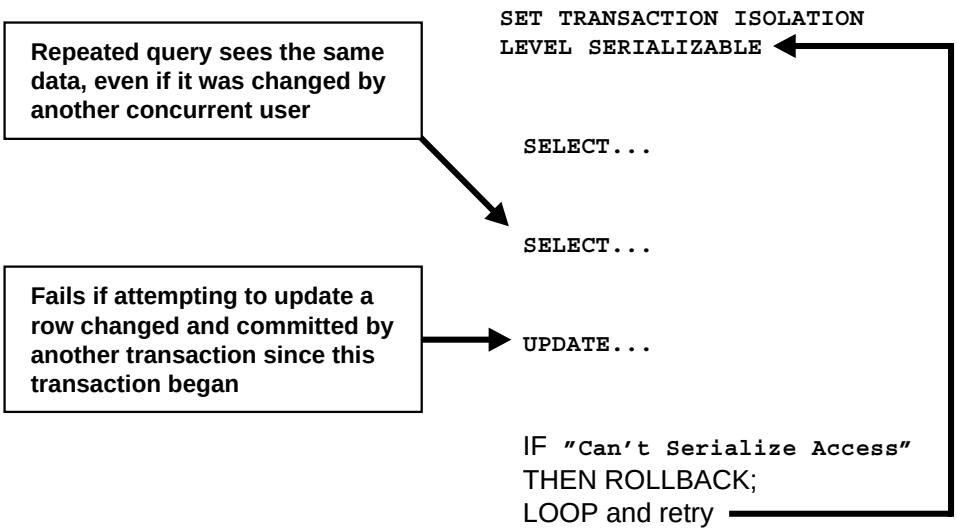
```
ORA-08177: Cannot serialize access for this transaction
```

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)
- Roll back the entire transaction

Figure 20- 2 shows an example of an application that rolls back and retries the transaction after it fails with the "Cannot serialize access" error:

Figure 20- 2 Serializable Transaction Failure



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 20- 2 summarizes key differences between read committed and serializable transactions in Oracle.

Table 20- 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
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 terminatess	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 roll back 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", 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:** Oracle9i Application Developer's Guide - Fundamentals 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 roll back 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:** Oracle9i Database Administrator's Guide

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 ("dirty") 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 executes 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 write / 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 roll back 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 execute 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). 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:

  1. Change the schema of a database table.
  2. Update a PL/SQL procedure to a new version that uses this new schema of the database table.

  Between Step 1 and Step 2, the new schema of the table is inconsistent with the implementation of the PL/SQL procedure. This inconsistency would adversely affect users concurrently trying to execute the PL/SQL procedure.

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.

See Also:

- *Oracle9i SQL Reference*
- *Oracle9i Database Administrator's Guide*

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" on page 20-21

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" on page 20-32

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 roll back 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.

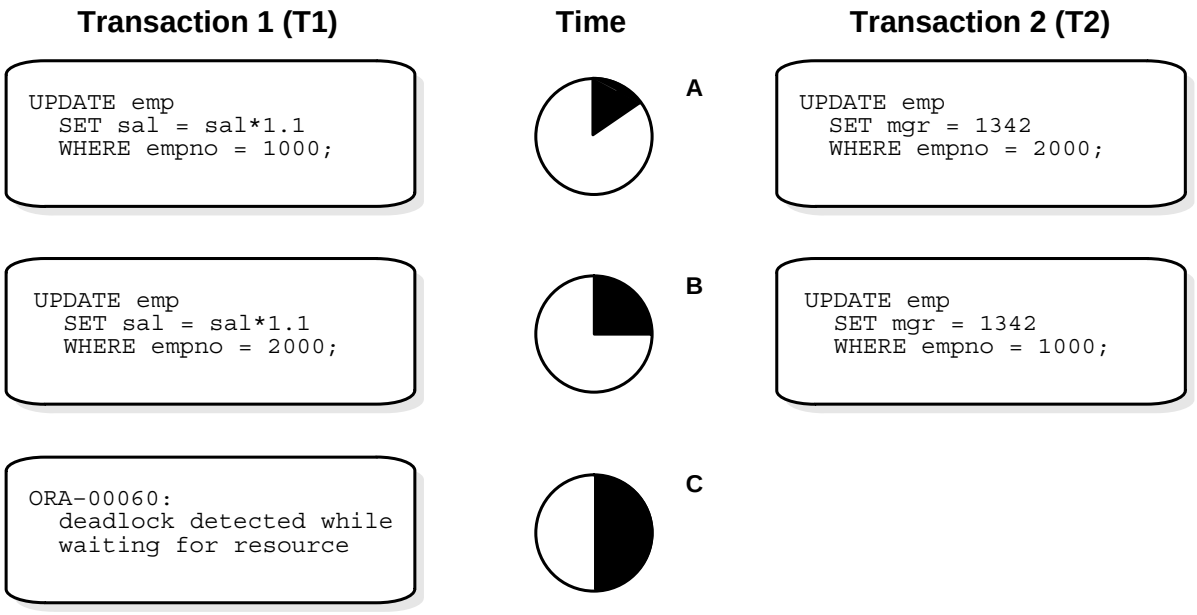
See Also: "Table Locks (TM)" on page 20-23

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 20- 3 illustrates two transactions in a deadlock.

In Figure 20- 3, 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 20- 3 Two Transactions in a Deadlock



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 a "waits for" graph, 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" on page 20-32 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 (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. For example, Oracle DML locks guarantee that a specific row in a table can be updated by only one transaction at a time and that a table cannot be dropped if an uncommitted transaction contains an insert into the table.

DML operations can acquire data locks at two different levels: for specific rows and for entire tables.

> **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)

The only DML locks Oracle acquires automatically are row-level locks. 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 DML 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 users 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" on page 20-30

Table Locks (TM)

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 20- 3 shows the table lock modes that statements acquire and operations that those locks permit and prohibit.

Table 20- 3 Summary of Table Locks

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
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" on page 20-32

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 executed:

```
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 execute `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 20- 4 summarizes the information in the following sections.

Table 20- 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
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 next section, "Share DDL Locks", 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 executed, 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 15, "Dependencies Among Schema Objects"

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 or SQL*Plus 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: *Oracle9i SQL Reference* for detailed descriptions of the SQL statements `LOCK TABLE` and `SELECT ... FOR UPDATE`

Examples of Concurrency under Explicit Locking

The following illustration shows how Oracle maintains data concurrency, integrity, and consistency when `LOCK TABLE` and `SELECT` with the `FOR UPDATE` clause statements are used.

Note: For brevity, the message text for `ORA-00054` ("resource busy and acquire with `NOWAIT` specified") is not included. User-entered text is in bold.

Transaction 1	Time Point	Transaction 2
LOCK TABLE scott.dept **IN ROW SHARE MODE;** Statement processed	1	
	2	**DROP TABLE scott.dept;** DROP TABLE scott.dept * ORA-00054 *(exclusive DDL lock not possible because of T1's table lock)*
	3	**LOCK TABLE scott.dept** **IN EXCLUSIVE MODE NOWAIT;** ORA-00054
	4	**SELECT LOC** **FROM scott.dept** **WHERE deptno = 20** **FOR UPDATE OF loc;** LOC - - - - - - - DALLAS 1 row selected
UPDATE scott.dept **SET loc = 'NEW YORK'** **WHERE deptno = 20;** *(waits because T2 has locked same rows)*	5	
	6	**ROLLBACK;** *(releases row locks)*
1 row processed. **ROLLBACK;**	7	
LOCK TABLE scott.dept **IN ROW EXCLUSIVE MODE;** Statement processed.	8	
	9	**LOCK TABLE scott.dept** **IN EXCLUSIVE MODE** **NOWAIT;** ORA-00054

Transaction 1	Time Point	Transaction 2
	10	**LOCK TABLE scott.dept IN SHARE ROW EXCLUSIVE MODE NOWAIT;** ORA-00054
	11	**LOCK TABLE scott.dept IN SHARE ROW EXCLUSIVE MODE NOWAIT;** ORA-00054
	12	**UPDATE scott.dept SET loc = 'NEW YORK' WHERE deptno = 20;** 1 row processed.
	13	**ROLLBACK;**
SELECT loc FROM scott.dept WHERE deptno = 20 FOR UPDATE OF loc; LOC - - - - - - DALLAS 1 row selected.	14	
	15	**UPDATE scott.dept SET loc = 'NEW YORK' WHERE deptno = 20;** *(waits because T1 has locked same rows)*
ROLLBACK;	16	
	17	1 row processed. *(conflicting locks were released)* **ROLLBACK;**
LOCK TABLE scott.dept IN SHARE MODE Statement processed	18	
	19	**LOCK TABLE scott.dept IN EXCLUSIVE MODE NOWAIT;** ORA-00054

Transaction 1	Time Point	Transaction 2
	20	**LOCK TABLE scott.dept** ** IN SHARE ROW EXCLUSIVE** ** MODE NOWAIT;** ORA-00054
	21	**LOCK TABLE scott.dept** ** IN SHARE MODE;** Statement processed.
	22	**SELECT loc** ** FROM scott.dept** ** WHERE deptno = 20;** LOC - - - - - - DALLAS 1 row selected.
	23	**SELECT loc** ** FROM scott.dept** ** WHERE deptno = 20** ** FOR UPDATE OF loc;** LOC - - - - - - DALLAS 1 row selected.
	24	**UPDATE scott.dept** ** SET loc = 'NEW YORK'** ** WHERE deptno = 20;** *(waits because T1 holds conflicting table lock)*
ROLLBACK;	25	
	26	1 row processed. *(conflicting table lock released)* **ROLLBACK;**
LOCK TABLE scott.dept ** IN SHARE ROW** ** EXCLUSIVE MODE;** Statement processed.	27	

Transaction 1	Time Point	Transaction 2
	28	**LOCK TABLE scott.dept** **IN EXCLUSIVE MODE** **NOWAIT;** ORA-00054
	29	**LOCK TABLE scott.dept** **IN SHARE ROW** **EXCLUSIVE MODE** **NOWAIT;** ORA-00054
	30	**LOCK TABLE scott.dept** **IN SHARE MODE NOWAIT;** ORA-00054
	31	**LOCK TABLE scott.dept** **IN ROW EXCLUSIVE** **MODE NOWAIT;** ORA-00054
	32	**LOCK TABLE scott.dept** **IN SHARE MODE NOWAIT;** ORA-00054
	33	**SELECT loc** **FROM scott.dept** **WHERE deptno = 20;** LOC - - - - - - DALLAS 1 row selected.
	34	**SELECT loc** **FROM scott.dept** **WHERE deptno = 20** **FOR UPDATE OF loc;** LOC - - - - - - DALLAS 1 row selected.

Transaction 1	Time Point	Transaction 2
	35	**UPDATE scott.dept** **SET loc = 'NEW YORK'** **WHERE deptno = 20;** *(waits because T1 holds conflicting table lock)*
UPDATE scott.dept **SET loc = 'NEW YORK'** **WHERE deptno = 20;** *(waits because T2 has locked same rows)*	36	*(deadlock)*
Cancel operation **ROLLBACK;**	37	
	38	1 row processed.
LOCK TABLE scott.dept **IN EXCLUSIVE MODE;**	39	
	40	**LOCK TABLE scott.dept** **IN EXCLUSIVE MODE;** ORA-00054
	41	**LOCK TABLE scott.dept** **IN ROW EXCLUSIVE MODE** **NOWAIT;** ORA-00054
	42	**LOCK TABLE scott.dept** **IN SHARE MODE;** ORA-00054
	43	**LOCK TABLE scott.dept** **IN ROW EXCLUSIVE** **MODE NOWAIT;** ORA-00054
	44	**LOCK TABLE scott.dept** **IN ROW SHARE MODE** **NOWAIT;** ORA-00054

Transaction 1	Time Point	Transaction 2
	45	**SELECT loc** ** FROM scott.dept** ** WHERE deptno = 20;** LOC - - - - - - DALLAS 1 row selected.
	46	**SELECT loc** ** FROM scott.dept** ** WHERE deptno = 20** ** FOR UPDATE OF loc;** *(waits because T1 has conflicting table lock)*
UPDATE scott.dept ** SET deptno = 30** ** WHERE deptno = 20;** 1 row processed.	47	
COMMIT;	48	
	49	0 rows selected. *(T1 released conflicting lock)*
SET TRANSACTION READ ONLY;	50	
SELECT loc ** FROM scott.dept** ** WHERE deptno = 10;** LOC - - - - - - BOSTON	51	
	52	**UPDATE scott.dept** ** SET loc = 'NEW YORK'** ** WHERE deptno = 10;** 1 row processed.

Transaction 1	Time Point	Transaction 2
SELECT loc ** FROM scott.dept** ** WHERE deptno = 10;** LOC - - - - - - BOSTON *(T1 does not see uncommitted data)*	53	
	54	**COMMIT;**
SELECT loc ** FROM scott.dept** ** WHERE deptno = 10;** LOC - - - - - - *(same results seen even after T2 commits)*	55	
COMMIT;	56	
SELECT loc ** FROM scott.dept** ** WHERE deptno = 10;** LOC - - - - - - NEW YORK *(committed data is seen)*	57	

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:

- *Oracle9i Application Developer's Guide - Fundamentals* for more information about Oracle Lock Management services
- *Oracle9i Supplied PL/SQL Packages and Types Reference* for information about `DBMS_LOCK`

Flashback Query

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 commit number (SCN).

Flashback query uses Oracle's multiversion read-consistency capabilities to restore data by applying undo as needed. Administrators can configure undo retention by simply specifying how long undo should be kept in the database. Using flashback query, a user 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 set the date and time you want to view. Then, any SQL query you execute 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:** *Oracle9i SQL Reference* for information on the AS OF clause

Flashback Query Benefits

- 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 ther 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 permitted while an object is being 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.

Notes:

- 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 be used to undo changes and can be very efficient if you know the rows that need to be moved back in time. You can in theory 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

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 enviornments, it could be used for extraction of data as of a consistent point in time from OLTP systems.

See Also:

- *Oracle9i Application Developer's Guide - Fundamentals* for more information about using flashback query
- *Oracle9i Supplied PL/SQL Packages and Types Reference* for a description of the `DBMS_FLASHBACK` package
- *Oracle9i Database Administrator's Guide* for information about undo tablespaces and setting retention period

21

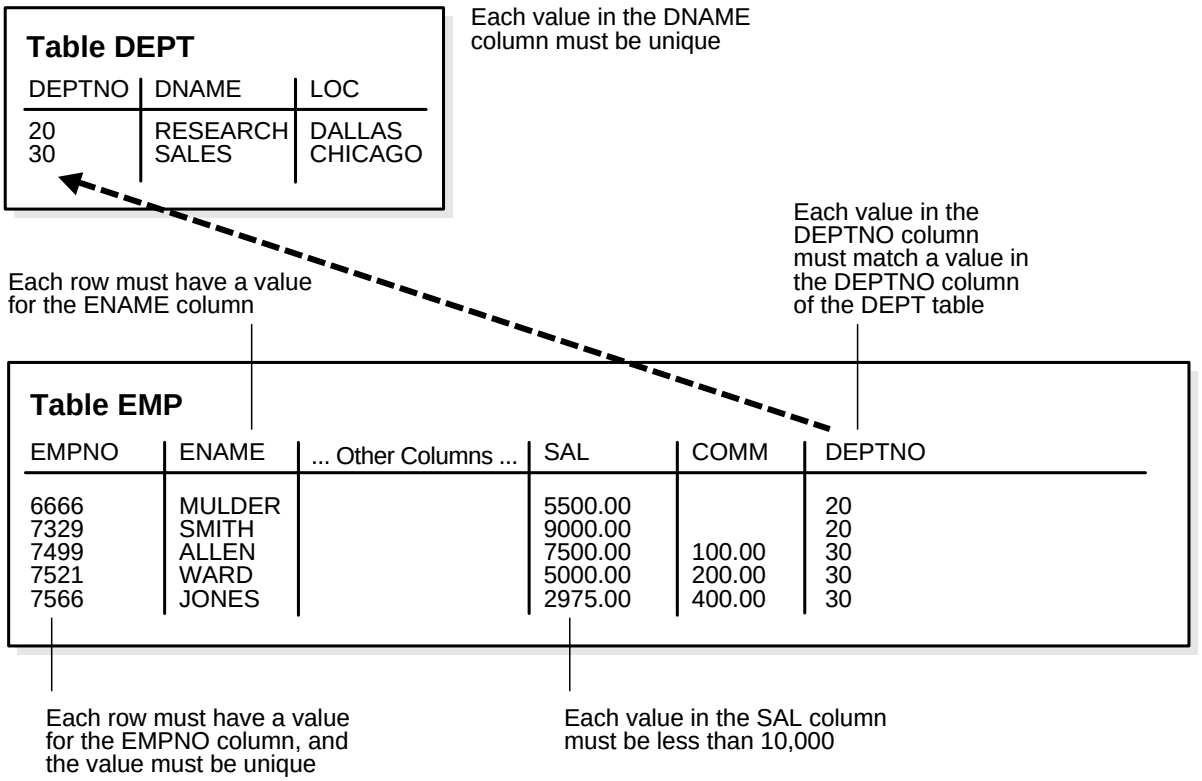
Data Integrity

This chapter explains how to use integrity constraints to enforce the business rules associated with your database and prevent the entry of invalid information into tables. The chapter includes:

Introduction to Data Integrity

It is important that data adhere to a predefined set of rules, as determined by the database administrator or application developer. As an example of data integrity, consider the tables `employees` and `departments` and the business rules for the information in each of the tables, as illustrated in Figure 21- 1.

Figure 21- 1 Examples of Data Integrity



Note that some columns in each table have specific rules that constrain the data contained within them.

Types of Data Integrity

This section describes the rules that can be applied to table columns to enforce different types of data integrity.

Null Rule

A null is a rule defined on a single column that allows or disallows inserts or updates of rows containing a null (the absence of a value) in that column.

Unique Column Values

A unique value defined on a column (or set of columns) allows the insert or update of a row only if it contains a unique value in that column (or set of columns).

Primary Key Values

A primary key value defined on a key (a column or set of columns) specifies that each row in the table can be uniquely identified by the values in the key.

Referential Integrity Rules

A rule defined on a key (a column or set of columns) in one table that guarantees that the values in that key match the values in a key in a related table (the referenced value).

Referential integrity also includes the rules that dictate what types of data manipulation are allowed on referenced values and how these actions affect dependent values. The rules associated with referential integrity are:

- Restrict: Disallows the update or deletion of referenced data.
- Set to Null: When referenced data is updated or deleted, all associated dependent data is set to `NULL`.
- Set to Default: When referenced data is updated or deleted, all associated dependent data is set to a default value.
- Cascade: When referenced data is updated, all associated dependent data is correspondingly updated. When a referenced row is deleted, all associated dependent rows are deleted.
- No Action: Disallows the update or deletion of referenced data. This differs from `RESTRICT` in that it is checked at the end of the statement, or at the end of the transaction if the constraint is deferred. (Oracle uses No Action as its default action.)

Complex Integrity Checking

Complex integrity checking is a user-defined rule for a column (or set of columns) that allows or disallows inserts, updates, or deletes of a row based on the value it contains for the column (or set of columns).

How Oracle Enforces Data Integrity

Oracle enables you to define and enforce each type of data integrity rule defined in the previous section. Most of these rules are easily defined using integrity constraints or database triggers.

Integrity Constraints Description

An integrity constraint is a declarative method of defining a rule for a column of a table. Oracle supports the following integrity constraints:

- `NOT NULL` constraints for the rules associated with nulls in a column
- `UNIQUE` key constraints for the rule associated with unique column values
- `PRIMARY KEY` constraints for the rule associated with primary identification values
- `FOREIGN KEY` constraints for the rules associated with referential integrity. Oracle supports the use of `FOREIGN KEY` integrity constraints to define the referential integrity actions, including:
  - Update and delete No Action
  - Delete `CASCADE`
  - Delete `SET NULL`
- `CHECK` constraints for complex integrity rules

Note: You cannot enforce referential integrity using declarative integrity constraints if child and parent tables are on different nodes of a distributed database. However, you can enforce referential integrity in a distributed database using database triggers (see next section).

Database Triggers

Oracle also lets you enforce integrity rules with a non-declarative approach using database triggers (stored database procedures automatically invoked on insert, update, or delete operations).

> **See Also:** Chapter 17, "Triggers" for examples of triggers used to enforce data integrity

Introduction to Integrity Constraints

Oracle uses integrity constraints to prevent invalid data entry into the base tables of the database. You can define integrity constraints to enforce the business rules you want to associate with the information in a database. If any of the results of a DML statement execution violate an integrity constraint, then Oracle rolls back the statement and returns an error.

> **Note:** Operations on views (and synonyms for tables) are subject to the integrity constraints defined on the underlying base tables.

For example, assume that you define an integrity constraint for the `salary` column of the `employees` table. This integrity constraint enforces the rule that no row in this table can contain a numeric value greater than 10,000 in this column. If an `INSERT` or `UPDATE` statement attempts to violate this integrity constraint, then Oracle rolls back the statement and returns an information error message.

The integrity constraints implemented in Oracle fully comply with ANSI X3.135-1989 and ISO 9075-1989 standards.

Advantages of Integrity Constraints

This section describes some of the advantages that integrity constraints have over other alternatives, which include:

- Enforcing business rules in the code of a database application
- Using stored procedures to completely control access to data
- Enforcing business rules with triggered stored database procedures

> **See Also:** Chapter 17, "Triggers"

Declarative Ease

Define integrity constraints using SQL statements. When you define or alter a table, no additional programming is required. The SQL statements are easy to write and eliminate programming errors. Oracle controls their functionality. For these reasons, declarative integrity constraints are preferable to application code and database triggers. The declarative approach is also better than using stored procedures, because the stored procedure solution to data integrity controls data access, but integrity constraints do not eliminate the flexibility of ad hoc data access.

Centralized Rules

Integrity constraints are defined for tables (not an application) and are stored in the data dictionary. Any data entered by any application must adhere to the same integrity constraints associated with the table. By moving business rules from application code to centralized integrity constraints, the tables of a database are guaranteed to contain valid data, no matter which database application manipulates the information. Stored procedures cannot provide the same advantage of centralized rules stored with a table. Database triggers can provide this benefit, but the complexity of implementation is far greater than the declarative approach used for integrity constraints.

Maximum Application Development Productivity

If a business rule enforced by an integrity constraint changes, then the administrator need only change that integrity constraint and all applications automatically adhere to the modified constraint. In contrast, if the business rule were enforced by the code of each database application, developers would have to modify all application source code and recompile, debug, and test the modified applications.

Immediate User Feedback

Oracle stores specific information about each integrity constraint in the data dictionary. You can design database applications to use this information to provide immediate user feedback about integrity constraint violations, even before Oracle executes and checks the SQL statement. For example, a SQL*Forms application can use integrity constraint definitions stored in the data dictionary to check for violations as values are entered into the fields of a form, even before the application issues a statement.

Superior Performance

The semantics of integrity constraint declarations are clearly defined, and performance optimizations are implemented for each specific declarative rule. The

Oracle query optimizer can use declarations to learn more about data to improve overall query performance. (Also, taking integrity rules out of application code and database triggers guarantees that checks are only made when necessary.)

Flexibility for Data Loads and Identification of Integrity Violations

You can disable integrity constraints temporarily so that large amounts of data can be loaded without the overhead of constraint checking. When the data load is complete, you can easily enable the integrity constraints, and you can automatically report any new rows that violate integrity constraints to a separate exceptions table.

The Performance Cost of Integrity Constraints

The advantages of enforcing data integrity rules come with some loss in performance. In general, the cost of including an integrity constraint is, at most, the same as executing a SQL statement that evaluates the constraint.

Types of Integrity Constraints

You can use the following integrity constraints to impose restrictions on the input of column values:

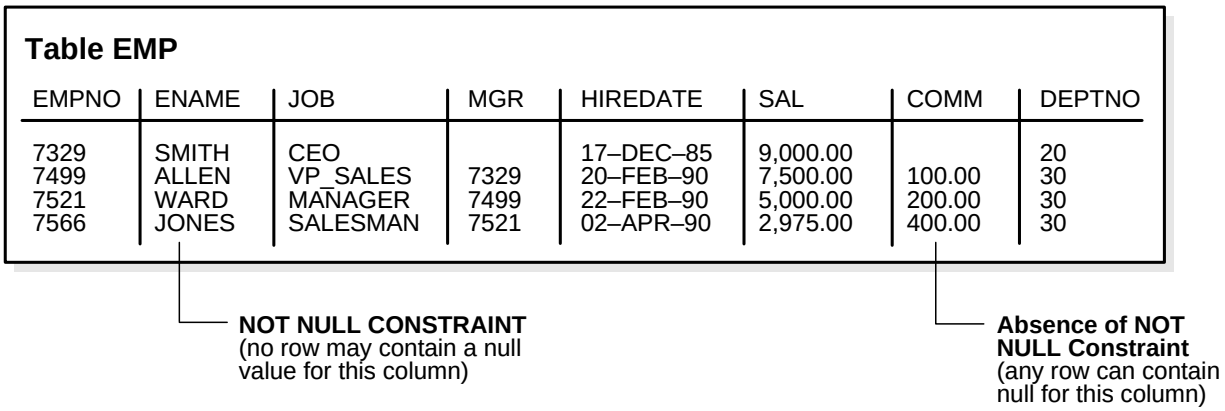
- NOT NULL Integrity Constraints
- UNIQUE Key Integrity Constraints
- PRIMARY KEY Integrity Constraints
- Referential Integrity Constraints
- CHECK Integrity Constraints

NOT NULL Integrity Constraints

By default, all columns in a table allow nulls. Null means the absence of a value. A `NOT NULL` constraint requires a column of a table contain no null values. For example, you can define a `NOT NULL` constraint to require that a value be input in the `last_name` column for every row of the `employees` table.

Figure 21- 2 illustrates a `NOT NULL` integrity constraint.

Figure 21- 2 NOT NULL Integrity Constraints

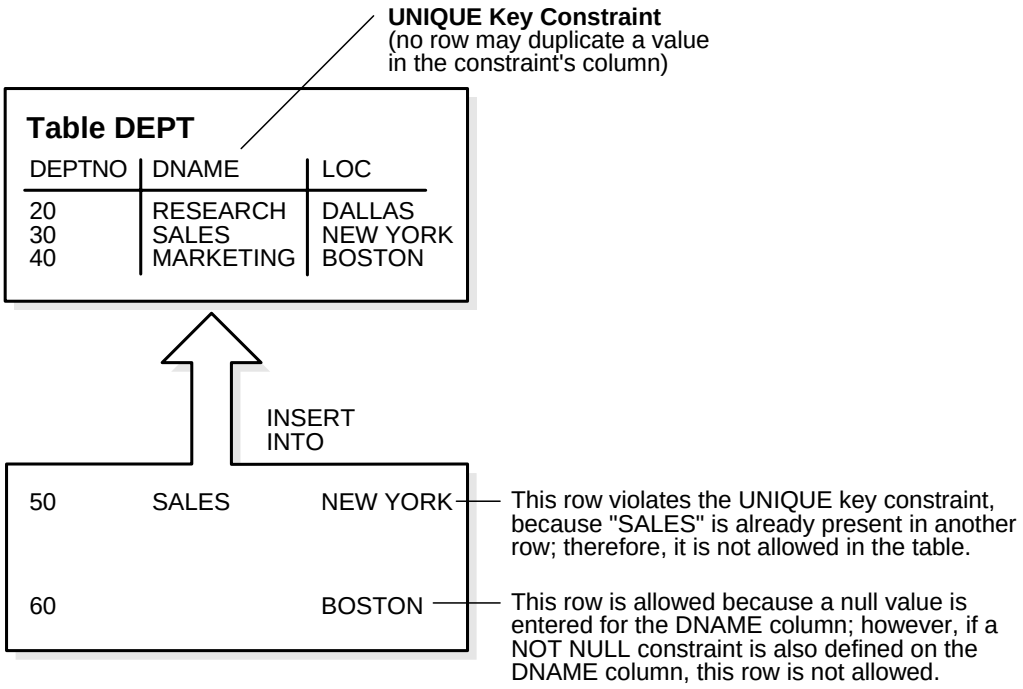


UNIQUE Key Integrity Constraints

A `UNIQUE` key integrity constraint requires that every value in a column or set of columns (key) be unique- that is, no two rows of a table have duplicate values in a specified column or set of columns.

For example, in Figure 21- 3 a `UNIQUE` key constraint is defined on the `DNAME` column of the `departments` table to disallow rows with duplicate department names.

Figure 21- 3 A UNIQUE Key Constraint

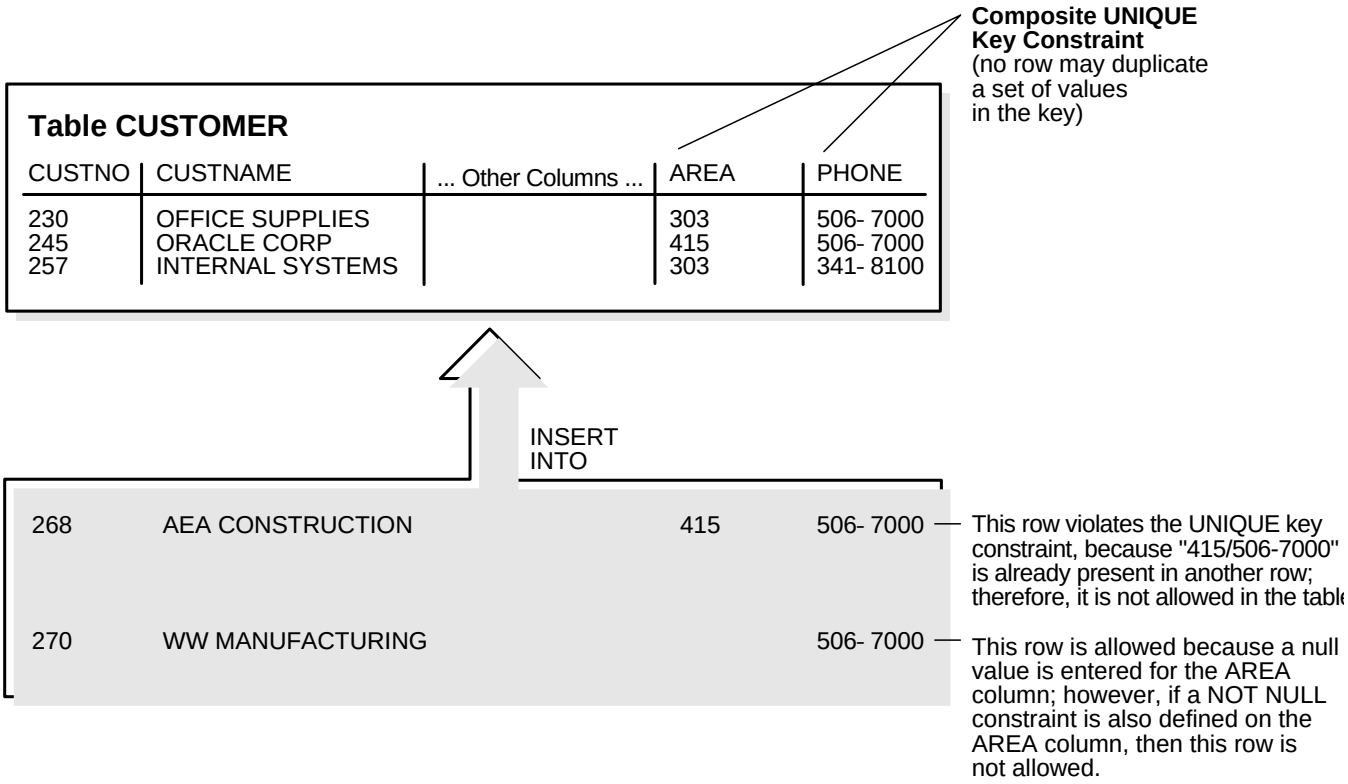


Unique Keys

The columns included in the definition of the `UNIQUE` key constraint are called the unique key. Unique key is often incorrectly used as a synonym for the terms UNIQUE key constraint or UNIQUE index. However, note that key refers only to the column or set of columns used in the definition of the integrity constraint.

If the `UNIQUE` key consists of more than one column, that group of columns is said to be a composite unique key. For example, in Figure 21- 4 the `customer` table has a `UNIQUE` key constraint defined on the composite unique key: the `area` and `phone` columns.

Figure 21- 4 A Composite UNIQUE Key Constraint



This `UNIQUE` key constraint lets you enter an area code and telephone number any number of times, but the combination of a given area code and given telephone number cannot be duplicated in the table. This eliminates unintentional duplication of a telephone number.

UNIQUE Key Constraints and Indexes

Oracle enforces unique integrity constraints with indexes. For example, in Figure 21- 4, Oracle enforces the `UNIQUE` key constraint by implicitly creating a unique index on the composite unique key. Therefore, composite `UNIQUE` key constraints have the same limitations imposed on composite indexes: up to 32 columns can constitute a composite unique key.

Note: If compatibility is set to Oracle9*i* or higher, then the total size in bytes of a key value can be almost as large as a full block. In previous releases key size could not exceed approximately half the associated database's block size.

If a usable index exists when a unique key constraint is created, the constraint uses that index rather than implicitly creating a new one.

Combine UNIQUE Key and NOT NULL Integrity Constraints

In Figure 21- 3 and Figure 21- 4, UNIQUE key constraints allow the input of nulls unless you also define NOT NULL constraints for the same columns. In fact, any number of rows can include nulls for columns without NOT NULL constraints because nulls are not considered equal to anything. A null in a column (or in all columns of a composite UNIQUE key) always satisfies a UNIQUE key constraint.

Columns with both unique keys and NOT NULL integrity constraints are common. This combination forces the user to enter values in the unique key and also eliminates the possibility that any new row's data will ever conflict with an existing row's data.

Note: Because of the search mechanism for UNIQUE constraints on more than one column, you cannot have identical values in the non-null columns of a partially null composite UNIQUE key constraint.

PRIMARY KEY Integrity Constraints

Each table in the database can have at most one PRIMARY KEY constraint. The values in the group of one or more columns subject to this constraint constitute the unique identifier of the row. In effect, each row is named by its primary key values.

The Oracle implementation of the PRIMARY KEY integrity constraint guarantees that both of the following are true:

- No two rows of a table have duplicate values in the specified column or set of columns.
- The primary key columns do not allow nulls. That is, a value must exist for the primary key columns in each row.

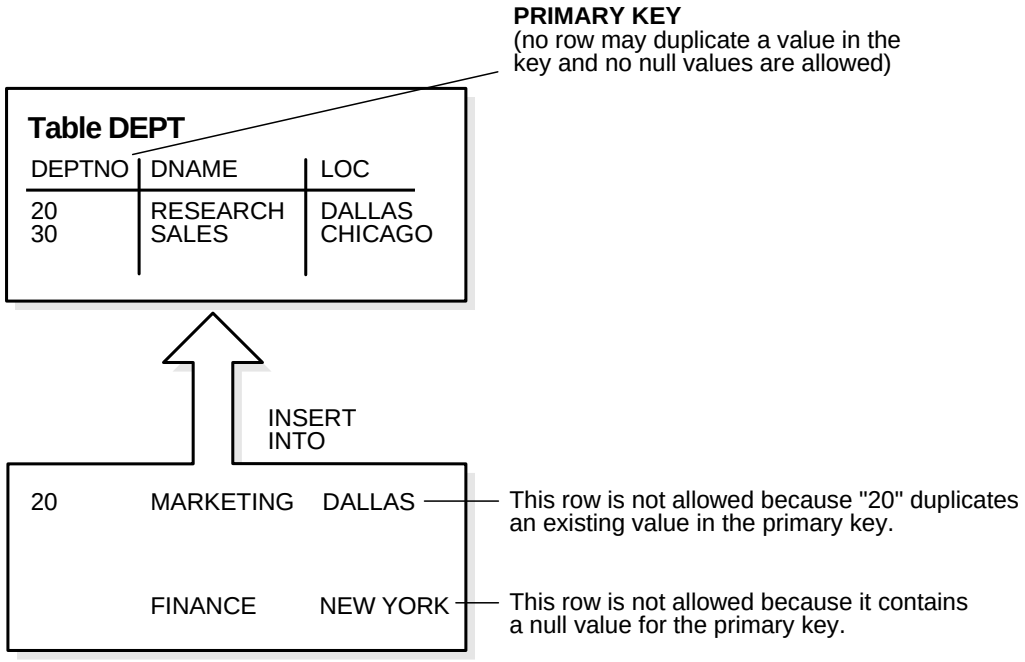
Primary Keys

The columns included in the definition of a table's `PRIMARY KEY` integrity constraint are called the *primary key*. Although it is not required, every table should have a primary key so that:

- Each row in the table can be uniquely identified
- No duplicate rows exist in the table

Figure 21- 5 illustrates a `PRIMARY KEY` constraint in the `departments` table and examples of rows that violate the constraint.

Figure 21- 5 A Primary Key Constraint

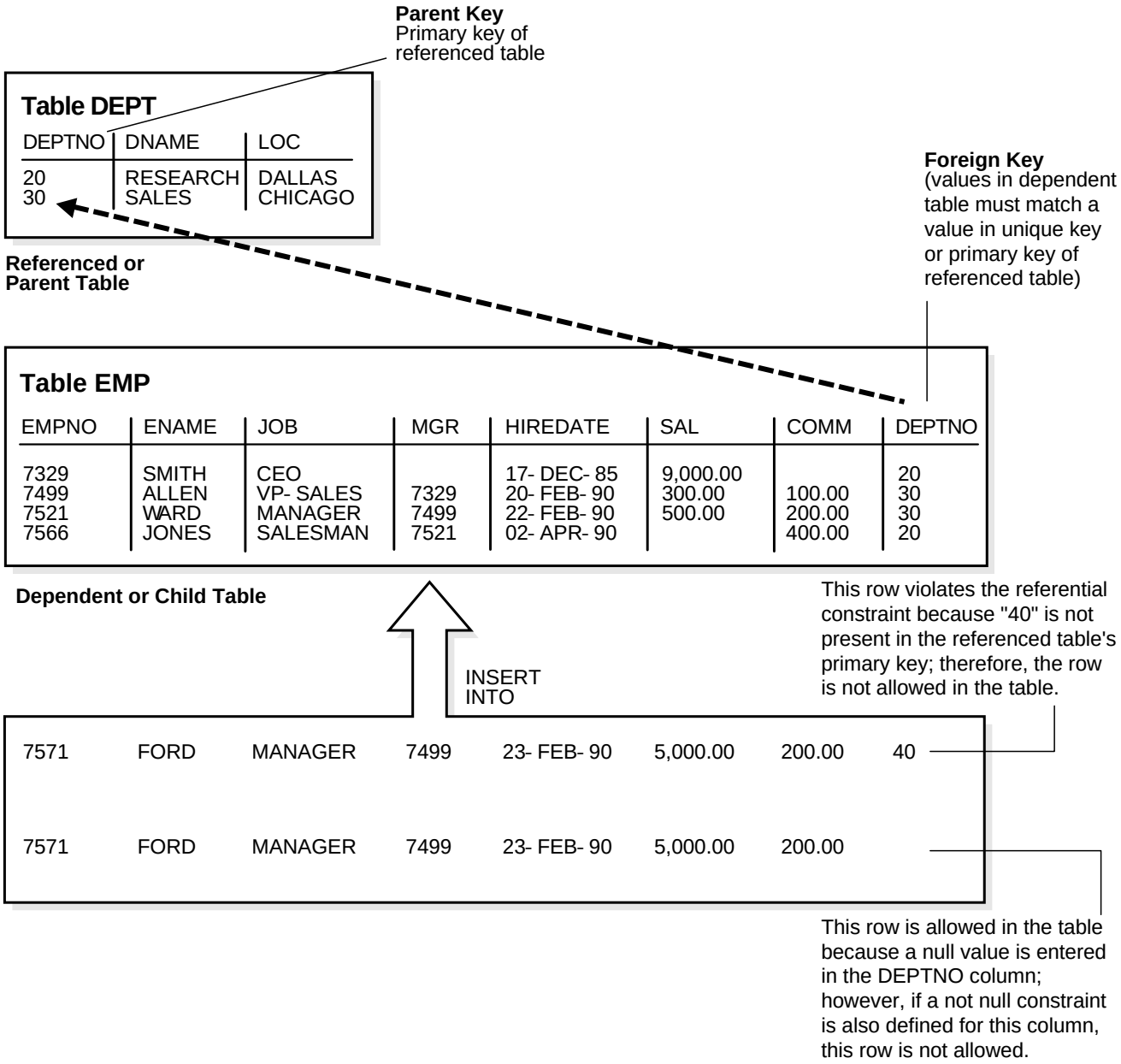


PRIMARY KEY Constraints and Indexes

Oracle enforces all `PRIMARY KEY` constraints using indexes. In Figure 21- 5, the primary key constraint created for the `department_id` column is enforced by the implicit creation of:

- A unique index on that column
- A `NOT NULL` constraint for that column

Figure 21- 6 Referential Integrity Constraints



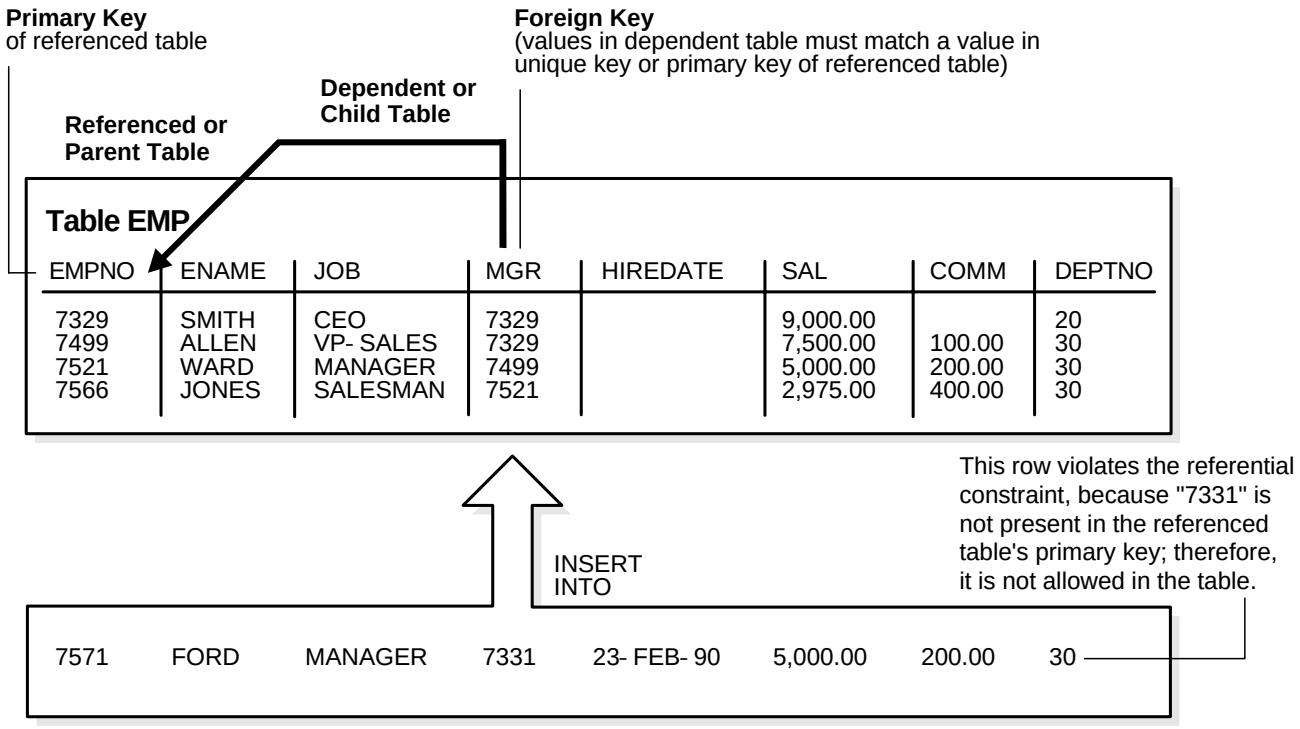
Self-Referential Integrity Constraints

Another type of referential integrity constraint, shown in Figure 21- 7, is called a self-referential integrity constraint. This type of foreign key references a parent key

in the same table.

In Figure 21- 7, the referential integrity constraint ensures that every value in the `manager_id` column of the `employees` table corresponds to a value that currently exists in the `employee_id` column of the same table, but not necessarily in the same row, because every manager must also be an employee. This integrity constraint eliminates the possibility of erroneous employee numbers in the `manager_id` column.

Figure 21- 7 Single Table Referential Constraints



Nulls and Foreign Keys

The relational model permits the value of foreign keys either to match the referenced primary or unique key value, or be null. If any column of a composite foreign key is null, then the non-null portions of the key do not have to match any corresponding portion of a parent key.

Actions Defined by Referential Integrity Constraints

Referential integrity constraints can specify particular actions to be performed on the dependent rows in a child table if a referenced parent key value is modified. The referential actions supported by the FOREIGN KEY integrity constraints of Oracle are UPDATE and DELETE NO ACTION, and DELETE CASCADE.

Note: Other referential actions not supported by FOREIGN KEY integrity constraints of Oracle can be enforced using database triggers.

See Chapter 17, "Triggers" for more information.

Update and Delete No Action The No Action (default) option specifies that referenced key values cannot be updated or deleted if the resulting data would violate a referential integrity constraint. For example, if a primary key value is referenced by a value in the foreign key, then the referenced primary key value cannot be deleted because of the dependent data.

Delete Cascade A delete cascades when rows containing referenced key values are deleted, causing all rows in child tables with dependent foreign key values to also be deleted. For example, if a row in a parent table is deleted, and this row's primary key value is referenced by one or more foreign key values in a child table, then the rows in the child table that reference the primary key value are also deleted from the child table.

Delete Set Null A delete sets null when rows containing referenced key values are deleted, causing all rows in child tables with dependent foreign key values to set those values to null. For example, if employee_id references manager_id in the TMP table, then deleting a manager causes the rows for all employees working for that manager to have their manager_id value set to null.

DML Restrictions with Respect to Referential Actions Table 21- 1 outlines the DML statements allowed by the different referential actions on the primary / unique key values in the parent table, and the foreign key values in the child table.

Table 21- 1 DML Statements Allowed by Update and Delete No Action

DML Statement	Issued Against Parent Table	Issued Against Child Table
`INSERT`	Always OK if the parent key value is unique.	OK only if the foreign key value exists in the parent key or is partially or all null.
`UPDATE No Action`	Allowed if the statement does not leave any rows in the child table without a referenced parent key value.	Allowed if the new foreign key value still references a referenced key value.
`DELETE No Action`	Allowed if no rows in the child table reference the parent key value.	Always OK.
`DELETE Cascade`	Always OK.	Always OK.
`DELETE Set Null`	Always OK.	Always OK.

Concurrency Control, Indexes, and Foreign Keys

You almost always index foreign keys. The only exception is when the matching unique or primary key is never updated or deleted.

Oracle maximizes the concurrency control of parent keys in relation to dependent foreign key values. You can control what concurrency mechanisms are used to maintain these relationships, and, depending on the situation, this can be highly beneficial. The following sections explain the possible situations and give recommendations for each.

No Index on the Foreign Key Figure 21- 8 illustrates the locking mechanisms used by Oracle when no index is defined on the foreign key and when rows are being updated or deleted in the parent table. Inserts into the parent table do not require any locks on the child table.

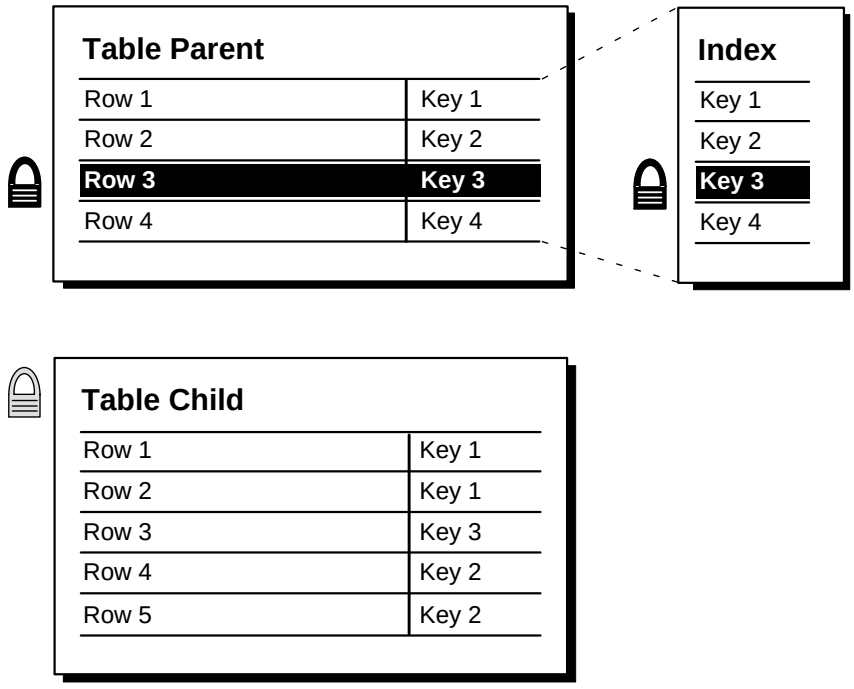
Oracle no longer requires a share lock on unindexed foreign keys when doing an update or delete on the primary key. It still obtains the table-level share lock, but then releases it immediately after obtaining it. If multiple primary keys are update or deleted, the lock is obtained and released once for each row.

In previous releases, a share lock of the entire child table was required until the transaction containing the `DELETE` statement for the parent table was committed. If the foreign key specifies `ON DELETE CASCADE`, then the `DELETE` statement resulted

in a table-level share-subexclusive lock on the child table. A share lock of the entire child table was also required for an UPDATE statement on the parent table that affected any columns referenced by the child table. Share locks allow reading only. Therefore, no INSERT, UPDATE, or DELETE statements could be issued on the child table until the transaction containing the UPDATE or DELETE was committed. Queries were allowed on the child table.

INSERT, UPDATE, and DELETE statements on the child table do not acquire any locks on the parent table, although INSERT and UPDATE statements wait for a row-lock on the index of the parent table to clear.

Figure 21- 8 Locking Mechanisms When No Index Is Defined on the Foreign Key



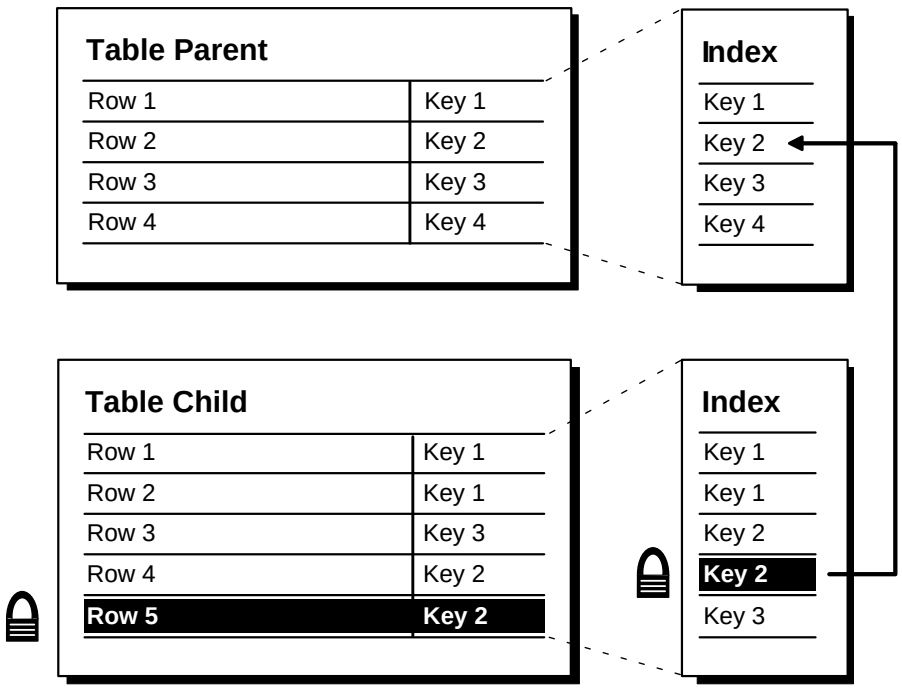
Index on the Foreign Key Figure 21- 9 illustrates the locking mechanisms used by Oracle when an index is defined on the foreign key, and new rows are inserted, updated, or deleted in the child table.

Notice that no table locks of any kind are acquired on the parent table or any of its indexes as a result of the insert, update, or delete. Therefore, any type of DML statement can be issued on the parent table, including inserts, updates, deletes, and queries.

This situation is preferable if there is any update or delete activity on the parent table while update activity is taking place on the child table. Inserts, updates, and deletes on the parent table do not require any locks on the child table, although

updates and deletes will wait for row-level locks on the indexes of the child table to clear.

Figure 21- 9 Locking Mechanisms When Index Is Defined on the Foreign Key



If the child table specifies `ON DELETE CASCADE`, then deletes from the parent table can result in deletes from the child table. In this case, waiting and locking rules are the same as if you deleted yourself from the child table after performing the delete from the parent table.

CHECK Integrity Constraints

A `CHECK` integrity constraint on a column or set of columns requires that a specified condition be true or unknown for every row of the table. If a DML statement results

in the condition of the CHECK constraint evaluating to false, then the statement is rolled back.

The Check Condition

CHECK constraints enable you to enforce very specific integrity rules by specifying a check condition. The condition of a CHECK constraint has some limitations:

- It must be a Boolean expression evaluated using the values in the row being inserted or updated, and
- It cannot contain subqueries; sequences; the SQL functi
- SYSDATE, UID, USER, or USERENV; or the pseudocolumns LEVEL or ROWNUM.

In evaluating CHECK constraints that contain string literals or SQL functions with globalization support parameters as arguments (such as TO_CHAR, TO_DATE, and TO_NUMBER), Oracle uses the database globalization support settings by default. You can override the defaults by specifying globalization support parameters explicitly in such functions within the CHECK constraint definition.

> **See Also:** *Oracle9i Database Globalization Support Guide* for more information on globalization support features

Multiple CHECK Constraints

A single column can have multiple CHECK constraints that reference the column in its definition. There is no limit to the number of CHECK constraints that you can define on a column.

If you create multiple CHECK constraints for a column, design them carefully so their purposes do not conflict. Do not assume any particular order of evaluation of the conditions. Oracle does not verify that CHECK conditions are not mutually exclusive.

The Mechanisms of Constraint Checking

To know what types of actions are permitted when constraints are present, it is useful to understand when Oracle actually performs the checking of constraints. To illustrate this, an example or two is helpful. Assume the following:

- The employees table has been defined as in Figure 21- 7 on page 21-15.
- The self-referential constraint makes the entries in the manager_id column dependent on the values of the employee_id column. For simplicity, the rest

of this discussion addresses only the `employee_id` and `manager_id` columns of the `employees` table.

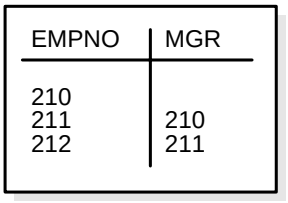
Consider the insertion of the first row into the `employees` table. No rows currently exist, so how can a row be entered if the value in the `manager_id` column cannot reference any existing value in the `employee_id` column? Three possibilities for doing this are:

- A null can be entered for the `manager_id` column of the first row, assuming that the `manager_id` column does not have a `NOT NULL` constraint defined on it. Because nulls are allowed in foreign keys, this row is inserted successfully into the table.
- The same value can be entered in both the `employee_id` and `manager_id` columns. This case reveals that Oracle performs its constraint checking *after* the statement has been completely executed. To allow a row to be entered with the same values in the parent key and the foreign key, Oracle must first execute the statement (that is, insert the new row) and then check to see if any row in the table has an `employee_id` that corresponds to the new row's `manager_id`.
- A multiple row `INSERT` statement, such as an `INSERT` statement with nested `SELECT` statement, can insert rows that reference one another. For example, the first row might have `employee_id` as 200 and `manager_id` as 300, while the second row might have `employee_id` as 300 and `manager_id` as 200.

  This case also shows that constraint checking is deferred until the complete execution of the statement. All rows are inserted first, then all rows are checked for constraint violations. You can also defer the checking of constraints until the end of the tr**ansaction**.

Consider the same self-referential integrity constraint in this scenario. The company has been sold. Because of this sale, all employee numbers must be updated to be the current value plus 5000 to coordinate with the new company's employee numbers. Because manager numbers are really employee numbers, these values must also increase by 5000 (see Figure 21- 10).

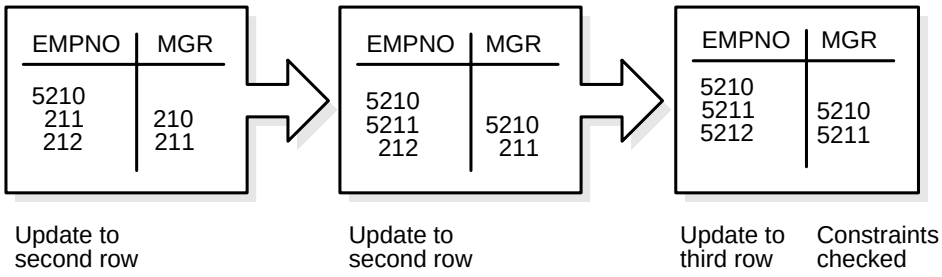
Figure 21–10 The EMP Table Before Updates




```
UPDATE employees
  SET employee_id = employee_id + 5000,
      manager_id = manager_id + 5000;
```

Even though a constraint is defined to verify that each `manager_id` value matches an `employee_id` value, this statement is legal because Oracle effectively performs its constraint checking after the statement completes. Figure 21- 11 shows that Oracle performs the actions of the entire SQL statement before any constraints are checked.

Figure 21–11 Constraint Checking



The examples in this section illustrate the constraint checking mechanism during `INSERT` and `UPDATE` statements. The same mechanism is used for all types of DML statements, including `UPDATE`, `INSERT`, and `DELETE` statements.

The examples also used self-referential integrity constraints to illustrate the checking mechanism. The same mechanism is used for all types of constraints, including the following:

- `NOT NULL`
- `UNIQUE` key
- `PRIMARY KEY`

- All types of FOREIGN KEY constraints
- CHECK constraints

See Also: "Deferred Constraint Checking" on page 21-24

Default Column Values and Integrity Constraint Checking

Default values are included as part of an INSERT statement before the statement is parsed. Therefore, default column values are subject to all integrity constraint checking.

Deferred Constraint Checking

You can defer checking constraints for validity until the end of the transaction.

- A constraint is deferred if the system checks that it is satisfied only on commit. If a deferred constraint is violated, then commit causes the transaction to roll back.
- If a constraint is immediate (not deferred), then it is checked at the end of each statement. If it is violated, the statement is rolled back immediately.

If a constraint causes an action (for example, delete cascade), that action is always taken as part of the statement that caused it, whether the constraint is deferred or immediate.

Constraint Attributes

You can define constraints as either deferrable or not deferrable, and either initially deferred or initially immediate. These attributes can be different for each constraint. You specify them with keywords in the CONSTRAINT clause:

- DEFERRABLE or NOT DEFERRABLE
- INITIALLY DEFERRED or INITIALLY IMMEDIATE

Constraints can be added, dropped, enabled, disabled, or validated. You can also modify a constraint's attributes.

See Also:

- *Oracle9i SQL Reference* for information about constraint attributes and their default values
- "Constraint States" on page 21-26
- "Constraint State Modification" on page 21-27

SET CONSTRAINTS Mode

The `SET CONSTRAINTS` statement makes constraints either `DEFERRED` or `IMMEDIATE` for a particular transaction (following the ANSI SQL92 standards in both syntax and semantics). You can use this statement to set the mode for a list of constraint names or for `ALL` constraints.

The `SET CONSTRAINTS` mode lasts for the duration of the transaction or until another `SET CONSTRAINTS` statement resets the mode.

`SET CONSTRAINTS ... IMMEDIATE` causes the specified constraints to be checked immediately on execution of each constrained statement. Oracle first checks any constraints that were deferred earlier in the transaction and then continues immediately checking constraints of any further statements in that transaction, as long as all the checked constraints are consistent and no other `SET CONSTRAINTS` statement is issued. If any constraint fails the check, an error is signaled. At that point, a `COMMIT` causes the whole transaction to roll back.

The `ALTER SESSION` statement also has clauses to `SET CONSTRAINTS IMMEDIATE` or `DEFERRED`. These clauses imply setting `ALL` deferrable constraints (that is, you cannot specify a list of constraint names). They are equivalent to making a `SET CONSTRAINTS` statement at the start of each transaction in the current session.

Making constraints immediate at the end of a transaction is a way of checking whether `COMMIT` can succeed. You can avoid unexpected rollbacks by setting constraints to `IMMEDIATE` as the last statement in a transaction. If any constraint fails the check, you can then correct the error before committing the transaction.

The `SET CONSTRAINTS` statement is disallowed inside of triggers.

`SET CONSTRAINTS` can be a distributed statement. Existing database links that have transactions in process are told when a `SET CONSTRAINTS ALL` statement occurs, and new links learn that it occurred as soon as they start a transaction.

Unique Constraints and Indexes

A user sees inconsistent constraints, including duplicates in unique indexes, when that user's transaction produces these inconsistencies.

You can place deferred unique and foreign key constraints on materialized views, allowing fast and complete refresh to complete successfully.

Deferrable unique constraints always use nonunique indexes. When you remove a deferrable constraint, its index remains. This is convenient because the storage information remains available after you disable a constraint. Not-deferrable unique constraints and primary keys also use a nonunique index if the nonunique index is placed on the key columns before the constraint is enforced.

Constraint States

You can enable or disable integrity constraints at the table level using the `CREATE TABLE` or `ALTER TABLE` statement. You can also set constraints to `VALIDATE` or `NOVALIDATE`, in any combination with `ENABLE` or `DISABLE`, where:

- `ENABLE` ensures that all incoming data conforms to the constraint
- `DISABLE` allows incoming data, regardless of whether it conforms to the constraint
- `VALIDATE` ensures that existing data conforms to the constraint
- `NOVALIDATE` means that some existing data may not conform to the constraint

In addition:

- `ENABLE VALIDATE` is the same as `ENABLE`. The constraint is checked and is guaranteed to hold for all rows.
- `ENABLE NOVALIDATE` means that the constraint is checked, but it does not have to be true for all rows. This allows existing rows to violate the constraint, while ensuring that all new or modified rows are valid.

  In an `ALTER TABLE` statement, `ENABLE NOVALIDATE` resumes constraint checking on disabled constraints without first validating all data in the table.
- `DISABLE NOVALIDATE` is the same as `DISABLE`. The constraint is not checked and is not necessarily true.
- `DISABLE VALIDATE` disables the constraint, drops the index on the constraint, and disallows any modification of the constrained columns.

For a UNIQUE constraint, the DISABLE VALIDATE state enables you to load data efficiently from a nonpartitioned table into a partitioned table using the EXCHANGE PARTITION clause of the ALTER TABLE statement.

Transitions between these states are governed by the following rules:

- ENABLE implies VALIDATE, unless NOVALIDATE is specified.
- DISABLE implies NOVALIDATE, unless VALIDATE is specified.
- VALIDATE and NOVALIDATE do not have any default implications for the ENABLE and DISABLE states.
- When a unique or primary key moves from the DISABLE state to the ENABLE state, if there is no existing index, a unique index is automatically created. Similarly, when a unique or primary key moves from ENABLE to DISABLE and it is enabled with a unique index, the unique index is dropped.
- When any constraint is moved from the NOVALIDATE state to the VALIDATE state, all data must be checked. (This can be very slow.) However, moving from VALIDATE to NOVALIDATE simply forgets that the data was ever checked.
- Moving a single constraint from the ENABLE NOVALIDATE state to the ENABLE VALIDATE state does not block reads, writes, or other DDL statements. It can be done in parallel.

See Also: *Oracle9i Database Administrator's Guide* for more information about how to use the ENABLE, DISABLE, VALIDATE, and NOVALIDATE CONSTRAINT clauses.

Constraint State Modification

You can use the MODIFY CONSTRAINT clause of the ALTER TABLE statement to change the following constraint states:

- DEFERRABLE or NOT DEFERRABLE
- INITIALLY DEFERRED or INITIALLY IMMEDIATE
- RELY or NORELY
- USING INDEX ...
- ENABLE or DISABLE
- VALIDATE or NOVALIDATE
- EXCEPTIONS INTO ...

See Also: *Oracle9i SQL Reference* for information about these constraint states

22

Controlling Database Access

This chapter explains how to control access to an Oracle database. It includes the following sections:

Introduction to Database Security

Database security entails allowing or disallowing user actions on the database and the objects within it. Oracle uses schemas and security domains to control access to data and to restrict the use of various database resources.

Oracle provides comprehensive discretionary access control. Discretionary access control regulates all user access to named objects through privileges. A privilege is permission to access a named object in a prescribed manner; for example, permission to query a table. Privileges are granted to users at the discretion of other users- hence the term discretionary access control.

See Also: Chapter 23, "Privileges, Roles, and Security Policies"

Schemas, Database Users, and Security Domains

A user (sometimes called a username) is a name defined in the database that can connect to and access objects. A schema is a named collection of objects, such as tables, views, clusters, procedures, and packages. Schemas and users help database administrators manage database security.

Enterprise users are managed in a directory and can be given access to multiple schemas and databases without having to create an account or schema in each database. This arrangement is simpler for users and for DBAs and also offers better security because their privileges can be altered in one place.

When creating a new database user or altering an existing one, the security administrator must make several decisions concerning a user's security domain. These include:

- Whether user authentication information is maintained by the database, the operating system, or a network authentication service
- Settings for the user's default and temporary tablespaces
- A list of tablespaces accessible to the user, if any, and the associated quotas for each listed tablespace
- The user's resource limit profile; that is, limits on the amount of system resources available to the user
- The privileges, roles, and security policies that provide the user with appropriate access to schema objects needed to perform database operations

This chapter describes the first four security domain options listed.

Note: The information in this chapter applies to all user-defined database users. It does not apply to the special database users SYS and SYSTEM. Settings for these users' security domains should never be altered.

See Also:

- Chapter 23, "Privileges, Roles, and Security Policies"
- *Oracle Advanced Security Administrator's Guide* for more information about enterprise users
- *Oracle9i Database Administrator's Guide* for more information about the special users SYS and SYSTEM, and for information about security administrators

User Authentication

To prevent unauthorized use of a database username, Oracle provides user validation through several different methods for normal database users. You can perform authentication by:

- The operating system
- A network service
- The associated Oracle database
- The Oracle database of a middle-tier application that performs transactions on behalf of the user
- The Secure Socket Layer (SSL) protocol

For simplicity, one method is usually used to authenticate all users of a database. However, Oracle allows use of all methods within the same database instance.

Oracle also encrypts passwords during transmission to ensure the security of network authentication.

Oracle requires special authentication procedures for database administrators, because they perform special database operations.

Authentication by the Operating System

Some operating systems permit Oracle to use information maintained by the operating system to authenticate users. The benefits of authentication by the operating system are:

- Users can connect to Oracle more conveniently, without specifying a username or password. For example, a user can invoke SQL*Plus and skip the username and password prompts by entering

  ```
  SQLPLUS /
  ```

- Control over user authorization is centralized in the operating system. Oracle need not store or manage user passwords. However, Oracle still maintains usernames in the database.
- Username entries in the database and operating system audit trails correspond.

If the operating system is used to authenticate database users, some special considerations arise with respect to distributed database environments and database links.

See Also:

- *Oracle9i Database Administrator's Guide*
- Your Oracle operating system-specific documentation for more information about authenticating by way of your operating system

Authentication by the Network

Oracle supports the following methods of authentication by the network.

Third Party-Based Authentication Technologies

If network authentication services are available to you (such as DCE, Kerberos, or SESAME), Oracle can accept authentication from the network service. To use a network authentication service with Oracle, you need Oracle9*i* Enterprise Edition with the Oracle Advanced Security option.

See Also:

- *Oracle9i Database Administrator's Guide* for more information about network authentication. If you use a network authentication service, some special considerations arise for network roles and database links.
- *Oracle Advanced Security Administrator's Guide* for information about the Oracle Advanced Security option

Public Key Infrastructure-Based Authentication

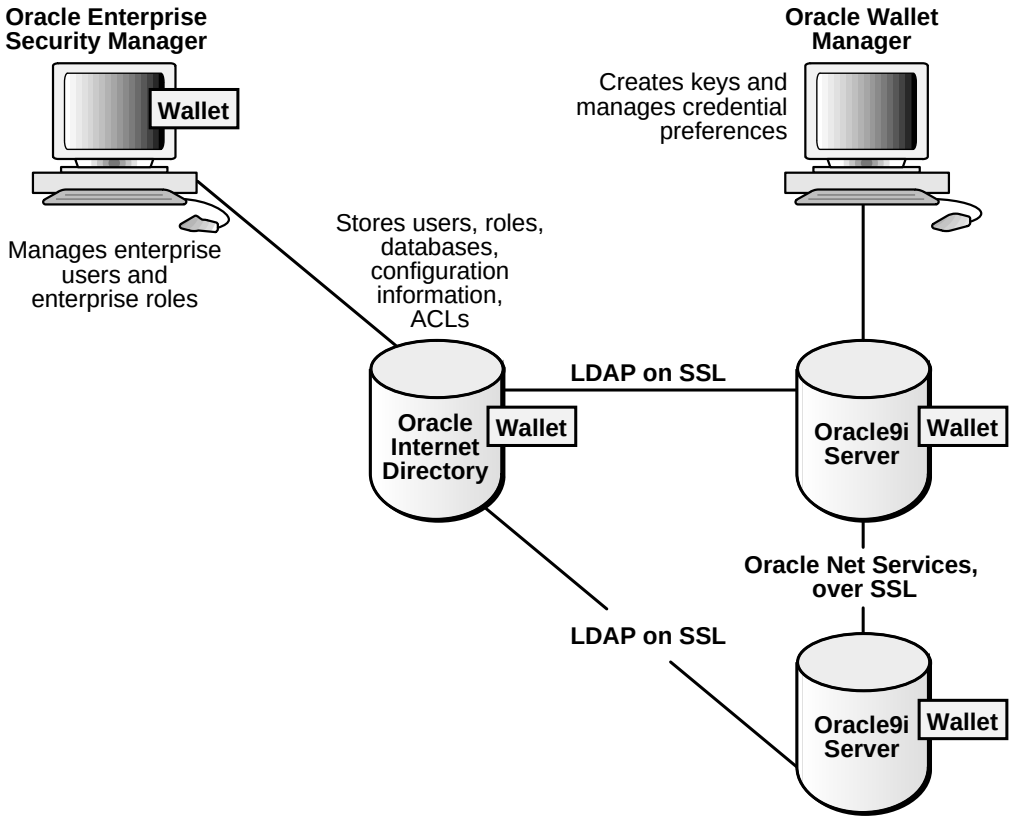
Authentication systems based on public key cryptography systems issue digital certificates to user clients, which use them to authenticate directly to servers in the enterprise without direct involvement of an authentication server. Oracle provides a public key infrastructure (PKI) for using public keys and certificates. It consists of the following components:

- Authentication and secure session key management using Secure Sockets Layer (SSL).
- Oracle Call Interface (OCI) and PL/SQL functions to sign user-specified data using a private key and certificate, and verify the signature on data using a trusted certificate.
- A trusted certificate, which is a third-party identity that is trusted. The trust is used when an identity is being validated as the entity it claims to be. Typically, the certificate authorities you trust issue user certificates. If there are several levels of trusted certificates, a trusted certificate at a lower level in the certificate chain does not need to have all its higher level certificates reverified.
- Oracle wallets, which are data structures that contain a user private key, a user certificate, and a set of trust points (the list of root certificates the user trusts).
- Oracle Wallet Manager, which is a standalone Java application used to manage and edit the security credentials in Oracle wallets. Wallet Manager:
  - Protects user keys
  - Manages X.509v3 certificates on Oracle clients and servers
  - Generates a public-private key pair and creates a certificate request for submission to a certificate authority
  - Installs a certificate for the entity
  - Configures trusted certificates for the entity

  - Opens a wallet to enable access to PKI-based services
  - Creates a wallet that can be opened using the Oracle Enterprise Login Assistant
- X.509v3 certificates that you obtain from a certificate authority outside of Oracle. It is created when an entity's public key is signed by a trusted entity (a certificate authority outside of Oracle). The certificate ensures that the entity's information is correct and the public key belongs to the entity. The certificates are loaded into Oracle wallets to enable authentication.
- Oracle Enterprise Security Manager, which provides centralized privilege management to make administration easier and increase your level of security. Oracle Enterprise Security Manager lets you store and retrieve roles from Oracle Internet Directory if the roles support the Lightweight Directory Access Protocol (LDAP). Oracle Enterprise Security Manager may also allow you to store roles in other LDAP v3-compliant directory servers if they can support the installation of the Oracle schema and related Access Control Lists.
- Oracle Internet Directory, which is an LDAP v3-compliant directory built on the Oracle9*i* database. It lets you manage the user and system configuration environment, including security attributes and privileges, for users authenticated using X.509 certificates. Oracle Internet Directory enforces attribute-level access control, allowing the directory to restrict read, write, or update privileges on specific attributes to specific named users (for example, an enterprise security administrator). It also supports protection and authentication of directory queries and responses through SSL encryption.
- Oracle Enterprise Login Assistant, which is a Java-based tool for opening and closing a user wallet in order to enable or disable secure SSL-based communications for an application. This tool provides a subset of the functionality proved by Oracle Wallet Manager. The wallet must be configured with Oracle Wallet Manager first.

Oracle's public key infrastructure is illustrated in Figure 22- 1.

Figure 22–1 Oracle Public Key Infrastructure



Note: To use public key infrastructure-based authentication with Oracle, you need Oracle9*i* Enterprise Edition with the Oracle Advanced Security option.

Remote Authentication

Oracle supports remote authentication of users through Remote Dial-In User Service (RADIUS), a standard lightweight protocol used for user authentication, authorization, and accounting.

Note: To use remote authentication of users through RADIUS with Oracle, you need Oracle9*i* Enterprise Edition with the Advanced Security option.

> **See Also:** *Oracle Advanced Security Administrator's Guide* for information about Oracle Advanced Security

Authentication by the Oracle Database

Oracle can authenticate users attempting to connect to a database by using information stored in that database.

When Oracle uses database authentication, you create each user with an associated password. A user provides the correct password when establishing a connection to prevent unauthorized use of the database. Oracle stores a user's password in the data dictionary in an encrypted format. A user can change his or her password at any time.

Password Encryption While Connecting

To protect password confidentiality, Oracle lets you encrypt passwords during network (client/server and server/server) connections. If you enable this functionality on the client and server machines, Oracle encrypts passwords using a modified DES (Data Encryption Standard) algorithm before sending them across the network. It is strongly recommended that you enable password encryption for connections to protect your passwords from network intrusion.

> **See Also:** *Oracle9i Database Administrator's Guide* for more information about encrypting passwords in network systems

Account Locking

Oracle can lock a user's account if the user fails to login to the system within a specified number of attempts. Depending on how the account is configured, it can be unlocked automatically after a specified time interval or it must be unlocked by the database administrator.

The `CREATE PROFILE` statement configures the number of failed logins a user can attempt and the amount of time the account remains locked before automatic unlock.

The database administrator can also lock accounts manually. When this occurs, the account cannot be unlocked automatically but must be unlocked explicitly by the database administrator.

> **See Also:** "Profiles" on page 22-20

Password Lifetime and Expiration

Password lifetime and expiration options allow the database administrator to specify a lifetime for passwords, after which time they expire and must be changed before a login to the account can be completed. On first attempt to login to the database account after the password expires, the user's account enters the grace period, and a warning message is issued to the user every time the user tries to login until the grace period is over.

The user is expected to change the password within the grace period. If the password is not changed within the grace period, the account is locked and no further logins to that account are allowed without assistance by the database administrator.

The database administrator can also set the password state to expired. When this happens, the user's account status is changed to expired, and the user or the database administrator must change the password before the user can log in to the database.

Password History

The password history option checks each newly specified password to ensure that a password is not reused for the specified amount of time or for the specified number of password changes. The database administrator can configure the rules for password reuse with `CREATE PROFILE` statements.

Password Complexity Verification

Complexity verification checks that each password is complex enough to provide reasonable protection against intruders who try to break into the system by guessing passwords.

The Oracle default password complexity verification routine requires that each password:

- Be a minimum of four characters in length
- Not equal the userid
- Include at least one alphabet character, one numeric character, and one punctuation mark
- Not match any word on an internal list of simple words like welcome, account, database, user, and so on
- Differ from the previous password by at least three characters

Multitier Authentication and Authorization

In a multitier environment, Oracle controls the security of middle-tier applications by limiting their privileges, preserving client identities through all tiers, and auditing actions taken on behalf of clients. In applications that use a heavy middle tier, such as a transaction processing monitor, it is important to be able to preserve the identity of the client connecting to the middle tier. Yet one advantage of a middle tier is connection pooling, which allows multiple users to access a data server without each of them needing a separate connection. In such environments, you need to be able to set up and break down connections very quickly. For these environments, Oracle offers the creation of lightweight sessions through the Oracle Call Interface. These lightweight sessions allow each user to be authenticated by a database password without the overhead of a separate database connection, as well as preserving the identity of the real user through the middle tier.

You can create lightweight sessions with or without passwords. If a middle tier is outside or on a firewall, it would be appropriate to establish the lightweight session with passwords for each lightweight user session. For an internal application server, it might be appropriate to create a lightweight session that does not require passwords.

Clients, Application Servers, and Database Servers

In a multitier architecture environment, an application server provides data for clients and serves as an interface between clients and one or more database servers.

This architecture lets you use an application server to validate the credentials of a client, such as a web browser. In addition, the database server can audit operations performed by the application server and operations performed by the application server on behalf of the client. For example, an operation performed by the application server on behalf of the client might be a request for information to be displayed on the client whereas an operation performed by the application server might be a request for a connection to the database server.

Authentication in a multitier environment is based on trust regions, including the following:

- The client provides proof of authentication to the application server, typically using a password or an X.509 certificate.
- The application server verifies the client authentication and then authenticates itself to the database server.

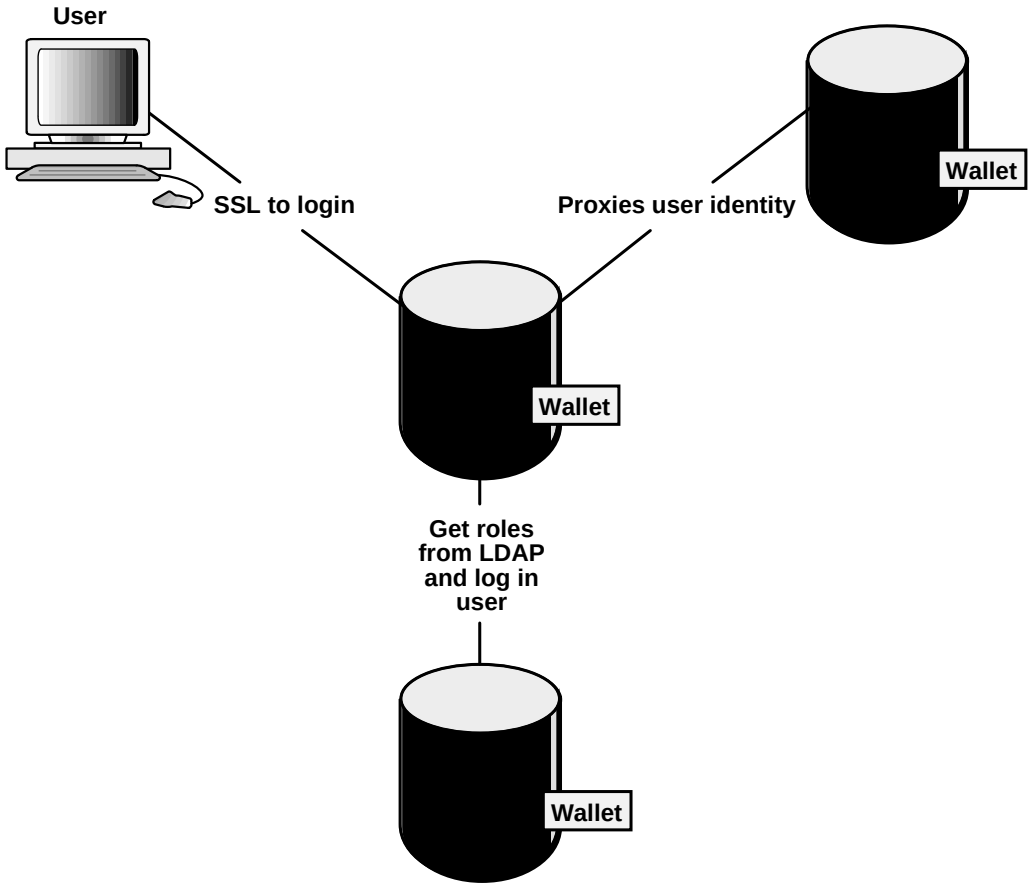
- The database server checks the application server authentication, verifies that the client exists, and verifies that the application server has the privilege to connect for this client.

Application servers can also enable roles for the client on whose behalf it is connecting. The application server can obtain these roles from a directory, which thus serves as an authorization repository. The application server can only request that these roles be enabled. The database verifies that:

- The client has these roles by checking its internal role repository.
- The application server has the privilege to connect on behalf of the user, using these roles for the user.

Figure 22- 2 shows an example of multitier authentication.

Figure 22–2 Multitier Authentication



Security Issues for Middle-Tier Applications

There are a number of security issues for middle-tier applications:

- Accountability: The database server must be able to distinguish between the actions of a client and the actions an application takes on behalf of a client. It must be possible to audit both kinds of actions.
- Differentiation: The database server must be able to distinguish between a web server transaction, a web server transaction on behalf of a browser client, and a client accessing the database directly.
- Least privilege: Users and middle tiers should be given the fewest privileges necessary to do their jobs.

Identity Issues in a Multitier Environment

Multitier authentication maintains the identify of the client through all tiers of the connection. This is necessary because if the identity of the originating client is lost, it is not possible to maintain useful audit records. In addition, it is not possible to distinguish operations performed by the application server on behalf of the client from those done by the application server for itself.

Restricted Privileges in a Multitier Environment

Privileges in a multitier environment are limited to what is necessary to perform the requested operation.

Client Privileges Client privileges are as limited as possible in a multitier environment. Operations are performed on behalf of the client by the application server.

Application Server Privileges Application server privileges in a multitier environment are limited so that the application server cannot perform unwanted or unneeded operations while performing a client operation.

> **See Also:** *Oracle9i Database Administrator's Guide* for more information about multitier authentication

Authentication by the Secure Socket Layer Protocol

The Secure Socket Layer (SSL) protocol is an application layer protocol. It can be used for user authentication to a database, independent of global user management in Oracle Internet Directory. That is, users can use SSL to authenticate to the database without implying anything about their directory access. However, if you

wish to use the enterprise user functionality to manage users and their privileges in a directory, the user must use SSL to authenticate to the database. A parameter in the initialization file governs which use of SSL is expected.

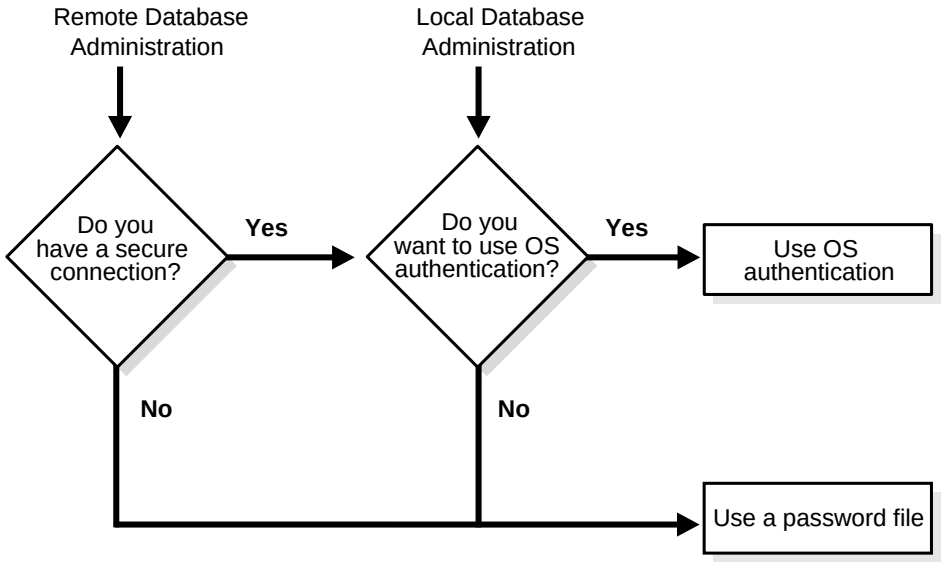
Authentication of Database Administrators

Database administrators perform special operations (such as shutting down or starting up a database) that should not be performed by normal database users. Oracle provides a more secure authentication scheme for database administrator usernames.

You can choose between operating system authentication or password files to authenticate database administrators.

Figure 22- 3 illustrates the choices you have for database administrator authentication schemes, depending on whether you administer your database locally (on the same machine on which the database resides) or if you administer many different database machines from a single remote client.

Figure 22–3 Database Administrator Authentication Methods



On most operating systems, operating system authentication for database administrators involves placing the operating system username of the database administrator in a special group (on UNIX systems, this is the dba group) or giving that operating system username a special process right.

The database uses password files to keep track of database usernames who have been granted the SYSDBA and SYSOPER privileges.

- SYSOPER lets database administrators perform STARTUP, SHUTDOWN, ALTER DATABASE OPEN/MOUNT, ALTER DATABASE BACKUP, ARCHIVE LOG, and RECOVER, and includes the RESTRICTED SESSION privilege.
- SYSDBA contains all system privileges with ADMIN OPTION, and the SYSOPER system privilege. Permits CREATE DATABASE and time-based recovery.

See Also:

- Your Oracle operating system-specific documentation for information about operating system authentication of database administrators
- *Oracle9i Database Administrator's Guide*

Oracle Internet Directory

Oracle Internet Directory is a directory service implemented as an application on the Oracle database. It enables retrieval of information about dispersed users and network resources. Oracle Internet Directory combines Lightweight Directory Access Protocol (LDAP), version 3, the open Internet standard directory access protocol, with the high performance, scalability, robustness, and availability of the Oracle Server.

Oracle Internet Directory includes the following:

- Oracle directory server, which responds to client requests for information about people and resources, and to updates of that information, using a multitier architecture directly over TCP/IP
- Oracle directory replication server, which replicates LDAP data between Oracle directory servers
- Oracle Directory Manager, a graphical user interface administration tool
- A variety of command line administration and data management tools

See Also: *Oracle Internet Directory Administrator's Guide*

User Tablespace Settings and Quotas

As part of every user's security domain, the database administrator can set several options regarding tablespace use:

Default Tablespace Option

When a user creates a schema object without specifying a tablespace to contain the object, Oracle places the object in the user's default tablespace. You set a user's default tablespace when the user is created, and you can change it after the user has been created.

Temporary Tablespace Option

When a user executes a SQL statement that requires the creation of a temporary segment, Oracle allocates that segment in the user's temporary tablespace.

Tablespace Access and Quotas

You can assign to each user a tablespace quota for any tablespace of the database. Doing so can accomplish two things:

- You allow the user to use the specified tablespace to create schema objects, if the user has the appropriate privileges.
- You can limit the amount of space allocated for storage of a user's schema objects in the specified tablespace.

By default, each user has no quota on any tablespace in the database. Therefore, if the user has the privilege to create some type of schema object, he or she must also have been either assigned a tablespace quota in which to create the object or been given the privilege to create that object in the schema of another user who was assigned a sufficient tablespace quota.

You can assign two types of tablespace quotas to a user: a quota for a specific amount of disk space in the tablespace (specified in bytes, kilobytes, or megabytes), or a quota for an unlimited amount of disk space in the tablespace. You should assign specific quotas to prevent a user's objects from consuming too much space in a tablespace.

Tablespace quotas and temporary segments have no effect on each other:

- Temporary segments do not consume any quota that a user might possess. The schema objects that Oracle automatically creates in temporary segments are owned by SYS and therefore are not subject to quotas.
- Temporary segments can be created in a tablespace for which a user has no quota.

You can assign a tablespace quota to a user when you create that user, and you can change that quota or add a different quota later.

Revoke a user's tablespace access by altering the user's current quota to zero. With a quota of zero, the user's objects in the revoked tablespace remain, but the objects cannot be allocated any new space.

The User Group PUBLIC

Each database contains a user group called PUBLIC. The PUBLIC user group provides public access to specific schema objects, such as tables and views, and provides all users with specific system privileges. Every user automatically belongs to the PUBLIC user group.

As members of PUBLIC, users can see (select from) all data dictionary tables prefixed with USER and ALL. Additionally, a user can grant a privilege or a role to PUBLIC. All users can use the privileges granted to PUBLIC.

You can grant or revoke any system privilege, object privilege, or role to PUBLIC. However, to maintain tight security over access rights, grant only privileges and roles that are of interest to all users to PUBLIC.

Granting and revoking some system and object privileges to and from PUBLIC can cause every view, procedure, function, package, and trigger in the database to be recompiled.

PUBLIC has the following restrictions:

- You cannot assign tablespace quotas to PUBLIC, although you can assign the UNLIMITED TABLESPACE system privilege to PUBLIC.
- You can create database links and synonyms as PUBLIC (using CREATE PUBLIC DATABASE LINK/SYNONYM), but no other schema object can be owned by PUBLIC. For example, the following statement is not legal:

```
CREATE TABLE public.employees ... ;
```

Note: Rollback segments can be created with the keyword `PUBLIC`, but these are not owned by the `PUBLIC` user group. All rollback segments are owned by `SYS`.

See Also:

- Chapter 2, "Data Blocks, Extents, and Segments"
- Chapter 23, "Privileges, Roles, and Security Policies"

User Resource Limits and Profiles

You can set limits on the amount of various system resources available to each user as part of a user's security domain. By doing so, you can prevent the uncontrolled consumption of valuable system resources such as CPU time.

This resource limit feature is very useful in large, multiuser systems, where system resources are very expensive. Excessive consumption of these resources by one or more users can detrimentally affect the other users of the database. In single-user or small-scale multiuser database systems, the system resource feature is not as important, because users' consumption of system resources is less likely to have detrimental impact.

You manage a user's resource limits and password management preferences with his or her profile- a named set of resource limits that you can assign to that user. Each Oracle database can have an unlimited number of profiles. Oracle allows the security administrator to enable or disable the enforcement of profile resource limits universally.

If you set resource limits, a slight degradation in performance occurs when users create sessions. This is because Oracle loads all resource limit data for the user when a user connects to a database.

See Also: *Oracle9i Database Administrator's Guide* for information about security administrators

Types of System Resources and Limits

Oracle can limit the use of several types of system resources, including CPU time and logical reads. In general, you can control each of these resources at the session level, the call level, or both.

- Session Level

Each time a user connects to a database, a session is created. Each session consumes CPU time and memory on the computer that executes Oracle. You can set several resource limits at the session level.

If a user exceeds a session-level resource limit, Oracle terminates (rolls back) the current statement and returns a message indicating the session limit has been reached. At this point, all previous statements in the current transaction are intact, and the only operations the user can perform are `COMMIT`, `ROLLBACK`, or disconnect (in this case, the current transaction is committed). All other operations produce an error. Even after the transaction is committed or rolled back, the user can accomplish no more work during the current session.

- Call Level

  Each time a SQL statement is executed, several steps are taken to process the statement. During this processing, several calls are made to the database as part of the different execution phases. To prevent any one call from using the system excessively, Oracle lets you set several resource limits at the call level.

  If a user exceeds a call-level resource limit, Oracle halts the processing of the statement, rolls back the statement, and returns an error. However, all previous statements of the current transaction remain intact, and the user's session remains connected.

CPU Time

When SQL statements and other types of calls are made to Oracle, an amount of CPU time is necessary to process the call. Average calls require a small amount of CPU time. However, a SQL statement involving a large amount of data or a runaway query can potentially consume a large amount of CPU time, reducing CPU time available for other processing.

To prevent uncontrolled use of CPU time, you can limit the CPU time for each call and the total amount of CPU time used for Oracle calls during a session. The limits are set and measured in CPU one-hundredth seconds (0.01 seconds) used by a call or a session.

Logical Reads

Input/output (I/O) is one of the most expensive operations in a database system. SQL statements that are I/O intensive can monopolize memory and disk use and cause other database operations to compete for these resources.

To prevent single sources of excessive I/O, Oracle let you limit the logical data block reads for each call and for each session. Logical data block reads include data

block reads from both memory and disk. The limits are set and measured in number of block reads performed by a call or during a session.

Other Resources

Oracle also provides for the limitation of several other resources at the session level:

- You can limit the number of concurrent sessions for each user. Each user can create only up to a predefined number of concurrent sessions.
- You can limit the idle time for a session. If the time between Oracle calls for a session reaches the idle time limit, the current transaction is rolled back, the session is aborted, and the resources of the session are returned to the system. The next call receives an error that indicates the user is no longer connected to the instance. This limit is set as a number of elapsed minutes.

  Note: Shortly after a session is aborted because it has exceeded an idle time limit, the process monitor (PMON) background process cleans up after the aborted session. Until PMON completes this process, the aborted session is still counted in any session/user resource limit.

- You can limit the elapsed connect time for each session. If a session's duration exceeds the elapsed time limit, the current transaction is rolled back, the session is dropped, and the resources of the session are returned to the system. This limit is set as a number of elapsed minutes.

  Note: Oracle does not constantly monitor the elapsed idle time or elapsed connection time. Doing so would reduce system performance. Instead, it checks every few minutes. Therefore, a session can exceed this limit slightly (for example, by five minutes) before Oracle enforces the limit and aborts the session.

- You can limit the amount of private SGA space (used for private SQL areas) for a session. This limit is only important in systems that use the shared server configuration. Otherwise, private SQL areas are located in the PGA. This limit is set as a number of bytes of memory in an instance's SGA. Use the characters K or M to specify kilobytes or megabytes.

See Also: *Oracle9i Database Administrator's Guide* for instructions about enabling and disabling resource limits

Profiles

A profile is a named set of specified resource limits that can be assigned to a valid username of an Oracle database. Profiles provide for easy management of resource limits. Profiles are also the way in which you administer password policy.

When to Use Profiles

You need to create and manage user profiles only if resource limits are a requirement of your database security policy. To use profiles, first categorize the related types of users in a database. Just as roles are used to manage the privileges of related users, profiles are used to manage the resource limits of related users. Determine how many profiles are needed to encompass all types of users in a database and then determine appropriate resource limits for each profile.

Determine Values for Resource Limits of a Profile

Before creating profiles and setting the resource limits associated with them, you should determine appropriate values for each resource limit. You can base these values on the type of operations a typical user performs. For example, if one class of user does not normally perform a high number of logical data block reads, then set the `LOGICAL_READS_PER_SESSION` and `LOGICAL_READS_PER_CALL` limits conservatively.

Usually, the best way to determine the appropriate resource limit values for a given user profile is to gather historical information about each type of resource usage. For example, the database or security administrator can use the `AUDIT SESSION` clause to gather information about the limits `CONNECT_TIME`, `LOGICAL_READS_PER_SESSION`, and `LOGICAL_READS_PER_CALL`.

You can gather statistics for other limits using the Monitor feature of Oracle Enterprise Manager (or SQL*Plus), specifically the Statistics monitor.

See Also: Chapter 24, "Auditing"

23

Privileges, Roles, and Security Policies

This chapter explains how you can control users' ability to execute system operations and to access schema objects by using privileges, roles, and security policies. The chapter includes:

Introduction to Privileges

A privilege is a right to execute a particular type of SQL statement or to access another user's object. Some examples of privileges include the right to:

- Connect to the database (create a session)
- Create a table
- Select rows from another user's table
- Execute another user's stored procedure

You grant privileges to users so these users can accomplish tasks required for their job. You should grant a privilege only to a user who absolutely requires the privilege to accomplish necessary work. Excessive granting of unnecessary privileges can compromise security. A user can receive a privilege in two different ways:

- You can grant privileges to users explicitly. For example, you can explicitly grant the privilege to insert records into the `employees` table to the user `SCOTT`.
- You can also grant privileges to a role (a named group of privileges), and then grant the role to one or more users. For example, you can grant the privileges to select, insert, update, and delete records from the `employees` table to the role named `clerk`, which in turn you can grant to the users `scott` and `brian`.

Because roles allow for easier and better management of privileges, you should normally grant privileges to roles and not to specific users.

There are two distinct categories of privileges:

- System privileges
- Schema object privileges

> **See Also:** *Oracle9i Database Administrator's Guide* for a complete list of all system and schema object privileges, as well as instructions for privilege management

System Privileges

A system privilege is the right to perform a particular action, or to perform an action on any schema objects of a particular type. For example, the privileges to create tablespaces and to delete the rows of any table in a database are system privileges. There are over 60 distinct system privileges.

Grant and Revoke System Privileges

You can grant or revoke system privileges to users and roles. If you grant system privileges to roles, then you can use the roles to manage system privileges. For example, roles permit privileges to be made selectively available.

> **Note:** In general, you grant system privileges only to administrative personnel and application developers. End users normally do not require the associated capabilities.

Use either of the following to grant or revoke system privileges to users and roles:

- Oracle Enterprise Manager Console
- The SQL statements `GRANT` and `REVOKE`

Who Can Grant or Revoke System Privileges?

Only users who have been granted a specific system privilege with the `ADMIN OPTION` or users with the system privileges `GRANT ANY PRIVILEGE` or `GRANT ANY OBJECT PRIVILEGE` can grant or revoke system privileges to other users.

Schema Object Privileges

A schema object privilege is a privilege or right to perform a particular action on a specific schema object:

- Table
- View
- Sequence
- Procedure
- Function
- Package

Different object privileges are available for different types of schema objects. For example, the privilege to delete rows from the `departments` table is an object privilege.

Some schema objects, such as clusters, indexes, triggers, and database links, do not have associated object privileges. Their use is controlled with system privileges. For

example, to alter a cluster, a user must own the cluster or have the ALTER ANY CLUSTER system privilege.

A schema object and its synonym are equivalent with respect to privileges. That is, the object privileges granted for a table, view, sequence, procedure, function, or package apply whether referencing the base object by name or using a synonym.

For example, assume there is a table jward.emp with a synonym named jward.employee and the user jward issues the following statement:

```
GRANT SELECT ON emp TO swilliams;
```

The user swilliams can query jward.emp by referencing the table by name or using the synonym jward.employee:

```
SELECT * FROM jward.emp;
SELECT * FROM jward.employee;
```

If you grant object privileges on a table, view, sequence, procedure, function, or package to a synonym for the object, the effect is the same as if no synonym were used. For example, if jward wanted to grant the SELECT privilege for the emp table to swilliams, jward could issue either of the following statements:

```
GRANT SELECT ON emp TO swilliams;
GRANT SELECT ON employee TO swilliams;
```

If a synonym is dropped, all grants for the underlying schema object remain in effect, even if the privileges were granted by specifying the dropped synonym.

Grant and Revoke Schema Object Privileges

Schema object privileges can be granted to and revoked from users and roles. If you grant object privileges to roles, you can make the privileges selectively available. Object privileges for users and roles can be granted or revoked using the following:

- The SQL statements GRANT and REVOKE, respectively
- The Add Privilege to Role/User dialog box and the Revoke Privilege from Role/User dialog box of Oracle Enterprise Manager.

Who Can Grant Schema Object Privileges?

A user automatically has all object privileges for schema objects contained in his or her schema. A user can grant any object privilege on any schema object he or she owns to any other user or role. A user with the GRANT ANY OBJECT PRIVILEGE can grant or revoke any specified object privilege to another user with or without the

GRANT OPTION of the GRANT statement. Otherwise, the grantee can use the privilege, but cannot grant it to other users.

For example, assume user SCOTT has a table named t2:

```
SQL>GRANT grant any object privilege TO U1;
SQL> connect u1/u1
Connected.
SQL> GRANT select on scott.t2 \TO U2;
SQL> SELECT GRANTEE, OWNER, GRANTOR, PRIVILEGE, GRANTABLE FROM DBA_TAB_PRIVS
 WHERE TABLE_NAME = 'employees';

GRANTEE                        OWNER
------------------------------ ------------------------------
GRANTOR                        PRIVILEGE                                GRA
------------------------------ ---------------------------------------- ---
U2                             SCOTT
SCOTT                          SELECT                                   NO
```

See Also: *Oracle9i SQL Reference*

Table Security

Schema object privileges for tables allow table security at the level of DML and DDL operations.

Data Manipulation Language Operations

You can grant privileges to use the DELETE, INSERT, SELECT, and UPDATE DML operations on a table or view. Grant these privileges only to users and roles that need to query or manipulate a table's data.

You can restrict INSERT and UPDATE privileges for a table to specific columns of the table. With selective INSERT, a privileged user can insert a row with values for the selected columns. All other columns receive NULL or the column's default value. With selective UPDATE, a user can update only specific column values of a row. Selective INSERT and UPDATE privileges are used to restrict a user's access to sensitive data.

For example, if you do not want data entry users to alter the salary column of the employees table, selective INSERT or UPDATE privileges can be granted that exclude the salary column. Alternatively, a view that excludes the salary column could satisfy this need for additional security.

See Also: *Oracle9i SQL Reference* for more information about these DML operations

Data Definition Language Operations

The ALTER, INDEX, and REFERENCES privileges allow DDL operations to be performed on a table. Because these privileges allow other users to alter or create dependencies on a table, you should grant privileges conservatively. A user attempting to perform a DDL operation on a table may need additional system or object privileges. For example, to create a trigger on a table, the user requires both the ALTER TABLE object privilege for the table and the CREATE TRIGGER system privilege.

As with the INSERT and UPDATE privileges, the REFERENCES privilege can be granted on specific columns of a table. The REFERENCES privilege enables the grantee to use the table on which the grant is made as a parent key to any foreign keys that the grantee wishes to create in his or her own tables. This action is controlled with a special privilege because the presence of foreign keys restricts the data manipulation and table alterations that can be done to the parent key. A column-specific REFERENCES privilege restricts the grantee to using the named columns (which, of course, must include at least one primary or unique key of the parent table).

See Also: Chapter 21, "Data Integrity" for more information about primary keys, unique keys, and integrity constraints

View Security

Schema object privileges for views allow various DML operations, which actually affect the base tables from which the view is derived. DML object privileges for tables can be applied similarly to views.

Privileges Required to Create Views

To create a view, you must meet the following requirements:

- You must have been granted one of the following system privileges, either explicitly or through a role:
  - The CREATE VIEW system privilege (to create a view in your schema)
  - The CREATE ANY VIEW system privilege (to create a view in another user's schema)
- You must have been explicitly granted one of the following privileges:

  - The SELECT, INSERT, UPDATE, or DELETE object privileges on all base objects underlying the view
  - The SELECT ANY TABLE, INSERT ANY TABLE, UPDATE ANY TABLE, or DELETE ANY TABLE system privileges

- Additionally, in order to grant other users access to your view, you must have received object privileges to the base objects with the GRANT OPTION clause or appropriate system privileges with the ADMIN OPTION clause. If you have not, grantees cannot access your view.

  See Also: *Oracle9i SQL Reference*

Increase Table Security with Views

To use a view, you require appropriate privileges only for the view itself. You do not require privileges on base objects underlying the view.

Views add two more levels of security for tables, column-level security and value-based security:

- A view can provide access to selected columns of base tables. For example, you can define a view on the employees table to show only the employee_id, last_name, and manager_id columns:

```
CREATE VIEW employees_manager AS
    SELECT last_name, employee_id, manager_id FROM employees;
```

- A view can provide value-based security for the information in a table. A WHERE clause in the definition of a view displays only selected rows of base tables. Consider the following two examples:

```
CREATE VIEW lowsal AS
    SELECT * FROM employees
    WHERE salary < 10000;
```

  The LOWSAL view allows access to all rows of the employees table that have a salary value less than 10000. Notice that all columns of the employees table are accessible in the LOWSAL view.

```
CREATE VIEW own_salary AS
    SELECT last_name, salary
    FROM employees
    WHERE last_name = USER;
```

In the own_salary view, only the rows with an last_name that matches the current user of the view are accessible. The own_salary view uses the user pseudocolumn, whose values always refer to the current user. This view combines both column-level security and value-based security.

Procedure Security

The only schema object privilege for procedures, including standalone procedures and functions as well as packages, is EXECUTE. Grant this privilege only to users who need to execute a procedure or compile another procedure that calls it.

Procedure Execution and Security Domains

A user with the EXECUTE object privilege for a specific procedure can execute the procedure or compile a program unit that references the procedure. No runtime privilege check is made when the procedure is called. A user with the EXECUTE ANY PROCEDURE system privilege can execute any procedure in the database.

A user can be granted privileges through roles to execute procedures.

Additional privileges on referenced objects are required for invoker-rights procedures, but not for definer-rights procedures.

See Also: "PL/SQL Blocks and Roles" on page 23-21

Definer Rights A user of a definer-rights procedure requires only the privilege to execute the procedure and no privileges on the underlying objects that the procedure accesses, because a definer-rights procedure operates under the security domain of the user who owns the procedure, regardless of who is executing it. The procedure's owner must have all the necessary object privileges for referenced objects. Fewer privileges have to be granted to users of a definer-rights procedure, resulting in tighter control of database access.

You can use definer-rights procedures to control access to private database objects and add a level of database security. By writing a definer-rights procedure and granting only EXECUTE privilege to a user, the user can be forced to access the referenced objects only through the procedure.

At runtime, the privileges of the owner of a definer-rights stored procedure are always checked before the procedure is executed. If a necessary privilege on a referenced object has been revoked from the owner of a definer-rights procedure, then the procedure cannot be executed by the owner or any other user.

Note: Trigger execution follows the same patterns as definer-rights procedures. The user executes a SQL statement, which that user is privileged to execute. As a result of the SQL statement, a trigger is fired. The statements within the triggered action temporarily execute under the security domain of the user that owns the trigger.

See Also: Chapter 17, "Triggers"

Invoker Rights An invoker-rights procedure executes with all of the invoker's privileges. Roles are enabled unless the invoker-rights procedure was called directly or indirectly by a definer-rights procedure. A user of an invoker-rights procedure needs privileges (either directly or through a role) on objects that the procedure accesses through external references that are resolved in the invoker's schema.

The invoker needs privileges at runtime to access program references embedded in DML statements or dynamic SQL statements, because they are effectively recompiled at runtime.

For all other external references, such as direct PL/SQL function calls, the owner's privileges are checked at compile time, and no runtime check is made. Therefore, the user of an invoker-rights procedure needs no privileges on external references outside DML or dynamic SQL statements. Alternatively, the developer of an invoker-rights procedure only needs to grant privileges on the procedure itself, not on all objects directly referenced by the invoker-rights procedure.

Many packages provided by Oracle, such as most of the `DBMS_*` packages, run with invoker rights- they do not run as the owner (`SYS`) but rather as the current user. However, some exceptions exist such as the `DBMS_RLS` package.

You can create a software bundle that consists of multiple program units, some with definer rights and others with invoker rights, and restrict the program entry points (controlled step-in). A user who has the privilege to execute an entry-point procedure can also execute internal program units indirectly, but cannot directly call the internal programs.

See Also:

- "Fine-Grained Access Control" on page 23-24
- Oracle9i Supplied PL/SQL Packages and Types Reference for detailed documentation of the Oracle supplied packages

System Privileges Needed to Create or Alter a Procedure

To create a procedure, a user must have the CREATE PROCEDURE or CREATE ANY PROCEDURE system privilege. To alter a procedure, that is, to manually recompile a procedure, a user must own the procedure or have the ALTER ANY PROCEDURE system privilege.

The user who owns the procedure also must have privileges for schema objects referenced in the procedure body. To create a procedure, you must have been explicitly granted the necessary privileges (system or object) on all objects referenced by the procedure. You cannot have obtained the required privileges through roles. This includes the EXECUTE privilege for any procedures that are called inside the procedure being created.

Triggers also require that privileges to referenced objects be granted explicitly to the trigger owner. Anonymous PL/SQL blocks can use any privilege, whether the privilege is granted explicitly or through a role.

Packages and Package Objects

A user with the EXECUTE object privilege for a package can execute any public procedure or function in the package and access or modify the value of any public package variable. Specific EXECUTE privileges cannot be granted for a package's constructs. Therefore, you may find it useful to consider two alternatives for establishing security when developing procedures, functions, and packages for a database application. These alternatives are described in the following examples.

Packages and Package Objects Example 1 This example shows four procedures created in the bodies of two packages.

```
CREATE PACKAGE BODY hire_fire AS
  PROCEDURE hire(...) IS
    BEGIN
      INSERT INTO employees . . .
    END hire;
  PROCEDURE fire(...) IS
    BEGIN
      DELETE FROM employees . . .
    END fire;
END hire_fire;

CREATE PACKAGE BODY raise_bonus AS
  PROCEDURE give_raise(...) IS
    BEGIN
      UPDATE employees SET salary = . . .
```

```
    END give_raise;
  PROCEDURE give_bonus(...) IS
    BEGIN
      UPDATE employees SET bonus = . . .
    END give_bonus;
END raise_bonus;
```

Access to execute the procedures is given by granting the EXECUTE privilege for the package, using the following statements:

```
GRANT EXECUTE ON hire_fire TO big_bosses;
GRANT EXECUTE ON raise_bonus TO little_bosses;
```

Granting EXECUTE privilege granted for a package provides uniform access to all package objects.

Packages and Package Objects Example 2 This example shows four procedure definitions within the body of a single package. Two additional standalone procedures and a package are created specifically to provide access to the procedures defined in the main package.

```
CREATE PACKAGE BODY employee_changes AS
  PROCEDURE change_salary(...) IS BEGIN ... END;
  PROCEDURE change_bonus(...) IS BEGIN ... END;
  PROCEDURE insert_employee(...) IS BEGIN ... END;
  PROCEDURE delete_employee(...) IS BEGIN ... END;
END employee_changes;

CREATE PROCEDURE hire
  BEGIN
    employee_changes.insert_employee(...)
  END hire;

CREATE PROCEDURE fire
  BEGIN
    employee_changes.delete_employee(...)
  END fire;

PACKAGE raise_bonus IS
  PROCEDURE give_raise(...) AS
    BEGIN
      employee_changes.change_salary(...)
    END give_raise;

  PROCEDURE give_bonus(...)
```

```
BEGIN
  employee_changes.change_bonus(...)
END give_bonus;
```

Using this method, the procedures that actually do the work (the procedures in the `employee_changes` package) are defined in a single package and can share declared global variables, cursors, on so on. By declaring top-level procedures `hire` and `fire`, and an additional package `raise_bonus`, you can grant selective `EXECUTE` privileges on procedures in the main package:

```
GRANT EXECUTE ON hire, fire TO big_bosses;
GRANT EXECUTE ON raise_bonus TO little_bosses;
```

Type Security

This section describes privileges for types, methods, and objects.

System Privileges for Named Types

Oracle defines system privileges shown in Table 23- 1 for named types (object types, `VARRAY`s, and nested tables):

Table 23–1 System Privileges for Named Types

Privilege	Allows you to...
`CREATE TYPE`	Create named types in your own schemas.
`CREATE ANY TYPE`	Create a named type in any schema.
`ALTER ANY TYPE`	Alter a named type in any schema.
`DROP ANY TYPE`	Drop a named type in any schema.
`EXECUTE ANY TYPE`	Use and reference a named type in any schema.

The `CONNECT` and `RESOURCE` roles include the `CREATE TYPE` system privilege. The DBA role includes all of these privileges.

Object Privileges

The only object privilege that applies to named types is `EXECUTE`. If the `EXECUTE` privilege exists on a named type, a user can use the named type to:

- Define a table
- Define a column in a relational table

- Declare a variable or parameter of the named type

The EXECUTE privilege permits a user to invoke the type's methods, including the type constructor. This is similar to EXECUTE privilege on a stored PL/SQL procedure.

Method Execution Model

Method execution is the same as any other stored PL/SQL procedure.

See Also: "Procedure Security" on page 23-8

Privileges Required to Create Types and Tables Using Types

To create a type, you must meet the following requirements:

- You must have the CREATE TYPE system privilege to create a type in your schema or the CREATE ANY TYPE system privilege to create a type in another user's schema. These privileges can be acquired explicitly or through a role.
- The owner of the type must have been explicitly granted the EXECUTE object privileges to access all other types referenced within the definition of the type, or have been granted the EXECUTE ANY TYPE system privilege. The owner cannot have obtained the required privileges through roles.
- If the type owner intends to grant access to the type to other users, the owner must have received the EXECUTE privileges to the referenced types with the GRANT OPTION or the EXECUTE ANY TYPE system privilege with the ADMIN OPTION. If not, the type owner has insufficient privileges to grant access on the type to other users.

To create a table using types, you must meet the requirements for creating a table and these additional requirements:

- The owner of the table must have been explicitly granted the EXECUTE object privileges to access all types referenced by the table, or have been granted the EXECUTE ANY TYPE system privilege. The owner cannot have obtained the required privileges through roles.
- If the table owner intends to grant access to the table to other users, the owner must have received the EXECUTE privileges to the referenced types with the GRANT OPTION or the EXECUTE ANY TYPE system privilege with the ADMIN OPTION. If not, the table owner has insufficient privileges to grant access on the type to other users.

See Also: "Table Security" on page 23-5 for the requirements for creating a table

Privileges Required to Create Types and Tables Using Types Example

Assume that three users exist with the CONNECT and RESOURCE roles:

- user1
- user2
- user3

User1 performs the following DDL in his schema:

```
CREATE TYPE type1 AS OBJECT (
  attr1 NUMBER);

CREATE TYPE type2 AS OBJECT (
  attr2 NUMBER);

GRANT EXECUTE ON type1 TO user2;
GRANT EXECUTE ON type2 TO user2 WITH GRANT OPTION;
```

User2 performs the following DDL in his schema:

```
CREATE TABLE tab1 OF user1.type1;
CREATE TYPE type3 AS OBJECT (
  attr3 user1.type2);
CREATE TABLE tab2 (
  col1 user1.type2);
```

The following statements succeed because user2 has EXECUTE privilege on user1's TYPE2 with the GRANT OPTION:

```
GRANT EXECUTE ON type3 TO user3;
GRANT SELECT on tab2 TO user3;
```

However, the following grant fails because user2 does not have EXECUTE privilege on user1's TYPE1 with the GRANT OPTION:

```
GRANT SELECT ON tab1 TO user3;
```

User3 can successfully perform the following statements:

```
CREATE TYPE type4 AS OBJECT (
  attr4 user2.type3);
CREATE TABLE tab3 OF type4;
```

Privileges on Type Access and Object Access

Existing column-level and table-level privileges for DML statements apply to both column objects and row objects. Oracle defines the privileges shown in Table 23- 2 for object tables:

Table 23–2 Privileges for Object Tables

Privilege	Allows you to...
SELECT	Access an object and its attributes from the table
UPDATE	Modify the attributes of the objects that make up the table's rows
INSERT	Create new objects in the table
DELETE	Delete rows

Similar table privileges and column privileges apply to column objects. Retrieving instances does not in itself reveal type information. However, clients must access named type information in order to interpret the type instance images. When a client requests such type information, Oracle checks for EXECUTE privilege on the type.

Consider the following schema:

```
CREATE TYPE emp_type (
    eno NUMBER, ename CHAR(31), eaddr addr_t);
CREATE TABLE emp OF emp_t;
```

and the following two queries:

```
SELECT VALUE(emp) FROM emp;
SELECT eno, ename FROM emp;
```

For either query, Oracle checks the user's SELECT privilege for the emp table. For the first query, the user needs to obtain the emp_type type information to interpret the data. When the query accesses the emp_type type, Oracle checks the user's EXECUTE privilege.

Execution of the second query, however, does not involve named types, so Oracle does not check type privileges.

Additionally, using the schema from the previous section, user3 can perform the following queries:

```
SELECT tab1.col1.attr2 FROM user2.tab1 tab1;
SELECT attr4.attr3.attr2 FROM tab3;
```

Note that in both SELECT statements, user3 does not have explicit privileges on the underlying types, but the statement succeeds because the type and table owners have the necessary privileges with the GRANT OPTION.

Oracle checks privileges on the following events, and returns an error if the client does not have the privilege for the action:

- Pinning an object in the object cache using its REF value causes Oracle to check SELECT privilege on the containing object table.
- Modifying an existing object or flushing an object from the object cache causes Oracle to check UPDATE privilege on the destination object table.
- Flushing a new object causes Oracle to check INSERT privilege on the destination object table.
- Deleting an object causes Oracle to check DELETE privilege on the destination table.
- Pinning an object of named type causes Oracle to check EXECUTE privilege on the object.

Modifying an object's attributes in a client 3GL application causes Oracle to update the entire object. Hence, the user needs UPDATE privilege on the object table. UPDATE privilege on only certain columns of the object table is not sufficient, even if the application only modifies attributes corresponding to those columns. Therefore, Oracle does not support column level privileges for object tables.

Type Dependencies

As with stored objects such as procedures and tables, types being referenced by other objects are called dependencies. There are some special issues for types depended upon by tables. Because a table contains data that relies on the type definition for access, any change to the type causes all stored data to become inaccessible. Changes that can cause this effect are when necessary privileges required by the type are revoked or the type or dependent types are dropped. If either of these actions occur, then the table becomes invalid and cannot be accessed.

A table that is invalid because of missing privileges can automatically become valid and accessible if the required privileges are granted again. A table that is invalid because a dependent type has been dropped can never be accessed again, and the only permissible action is to drop the table.

Because of the severe effects which revoking a privilege on a type or dropping a type can cause, the SQL statements REVOKE and DROP TYPE by default implement a restrict semantics. This means that if the named type in either statement has table or

type dependents, then an error is received and the statement aborts. However, if the FORCE clause for either statement is used, the statement always succeeds, and if there are depended-upon tables, they are invalidated.

> **See Also:** Oracle9i Database Reference for details about using the REVOKE, DROP TYPE, and FORCE clauses

Introduction to Roles

Oracle provides for easy and controlled privilege management through roles. Roles are named groups of related privileges that you grant to users or other roles. Roles are designed to ease the administration of end-user system and schema object privileges. However, roles are not meant to be used for application developers, because the privileges to access schema objects within stored programmatic constructs need to be granted directly.

These following properties of roles enable easier privilege management within a database:

Property	Description
Reduced privilege administration	Rather than granting the same set of privileges explicitly to several users, you can grant the privileges for a group of related users to a role, and then only the role needs to be granted to each member of the group.
Dynamic privilege management	If the privileges of a group must change, only the privileges of the role need to be modified. The security domains of all users granted the group's role automatically reflect the changes made to the role.
Selective availability of privileges	You can selectively enable or disable the roles granted to a user. This allows specific control of a user's privileges in any given situation.
Application awareness	The data dictionary records which roles exist, so you can design applications to query the dictionary and automatically enable (or disable) selective roles when a user attempts to execute the application by way of a given username.
Application-specific security	You can protect role use with a password. Applications can be created specifically to enable a role when supplied the correct password. Users cannot enable the role if they do not know the password.

Database administrators often create roles for a database application. The DBA grants a secure application role all privileges necessary to run the application. The DBA then grants the secure application role to other roles or users. An application can have several different roles, each granted a different set of privileges that allow for more or less data access while using the application.

The DBA can create a role with a password to prevent unauthorized use of the privileges granted to the role. Typically, an application is designed so that when it starts, it enables the proper role. As a result, an application user does not need to know the password for an application's role.

See Also:

- "Data Definition Language Statements and Roles" on page 23-22 for information about restrictions for procedures
- Oracle9i Application Developer's Guide - Fundamentals for instructions for enabling roles from an application

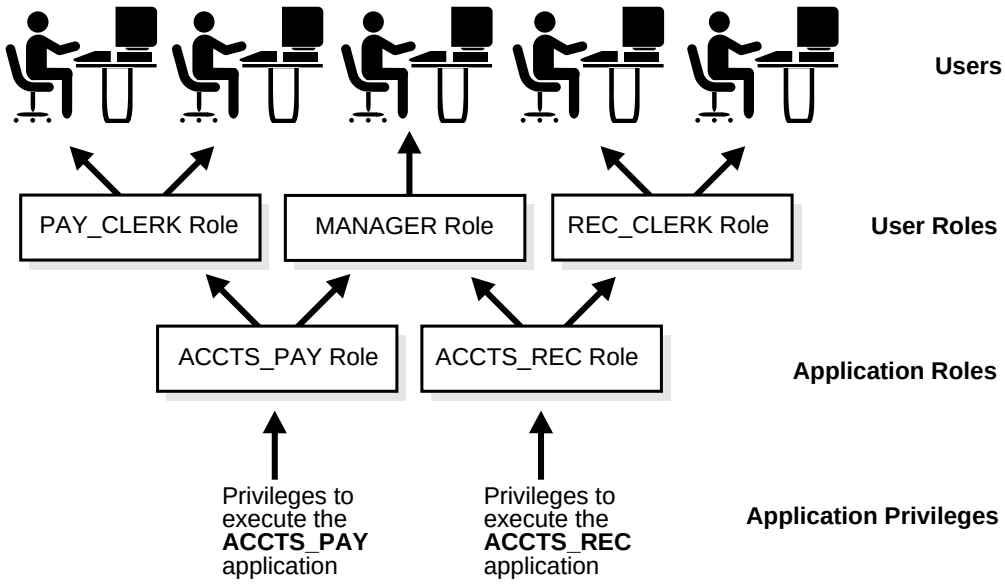
Common Uses for Roles

In general, you create a role to serve one of two purposes:

- To manage the privileges for a database application
- To manage the privileges for a user group

Figure 23- 1 and the sections that follow describe the two uses of roles.

Figure 23–1 Common Uses for Roles



Application Roles

You grant an application role all privileges necessary to run a given database application. Then, you grant the secure application role to other roles or to specific users. An application can have several different roles, with each role assigned a different set of privileges that allow for more or less data access while using the application.

User Roles

You create a user role for a group of database users with common privilege requirements. You manage user privileges by granting secure application roles and privileges to the user role and then granting the user role to appropriate users.

The Mechanisms of Roles

Database roles have the following functionality:

- A role can be granted system or schema object privileges.
- A role can be granted to other roles. However, a role cannot be granted to itself and cannot be granted circularly. For example, role `A` cannot be granted to role `B` if role `B` has previously been granted to role `A`.

- Any role can be granted to any database user.
- Each role granted to a user is, at a given time, either enabled or disabled. A user's security domain includes the privileges of all roles currently enabled for the user and excludes the privileges of any roles currently disabled for the user. Oracle allows database applications and users to enable and disable roles to provide selective availability of privileges.
- An indirectly granted role is a role granted to a role. It can be explicitly enabled or disabled for a user. However, by enabling a role that contains other roles, you implicitly enable all indirectly granted roles of the directly granted role.

Grant and Revoke Roles

You grant or revoke roles from users or other roles using the following options:

- The Grant System Privileges / Roles dialog box and Revoke System Privileges / Roles dialog box of Oracle Enterprise Manager
- The SQL statements `GRANT` and `REVOKE`

Privileges are granted to and revoked from roles using the same options. Roles can also be granted to and revoked from users using the operating system that executes Oracle, or through network services.

> **See Also:** *Oracle9i Database Administrator's Guide* for detailed instructions about role management

Who Can Grant or Revoke Roles?

Any user with the `GRANT ANY ROLE` system privilege can grant or revoke any role except a global role to or from other users or roles of the database. You should grant this system privilege conservatively because it is very powerful.

Any user granted a role with the `ADMIN OPTION` can grant or revoke that role to or from other users or roles of the database. This option allows administrative powers for roles on a selective basis.

> **See Also:** *Oracle9i Database Administrator's Guide* for information about global roles

Role Names

Within a database, each role name must be unique, and no username and role name can be the same. Unlike schema objects, roles are not contained in any schema. Therefore, a user who creates a role can be dropped with no effect on the role.

Security Domains of Roles and Users

Each role and user has its own unique security domain. A role's security domain includes the privileges granted to the role plus those privileges granted to any roles that are granted to the role.

A user's security domain includes privileges on all schema objects in the corresponding schema, the privileges granted to the user, and the privileges of roles granted to the user that are currently enabled. (A role can be simultaneously enabled for one user and disabled for another.) A user's security domain also includes the privileges and roles granted to the user group `PUBLIC`.

PL/SQL Blocks and Roles

The use of roles in a PL/SQL block depends on whether it is an anonymous block or a named block (stored procedure, function, or trigger), and whether it executes with definer rights or invoker rights.

Named Blocks with Definer Rights

All roles are disabled in any named PL/SQL block (stored procedure, function, or trigger) that executes with definer rights. Roles are not used for privilege checking and you cannot set roles within a definer-rights procedure.

The `SESSION_ROLES` view shows all roles that are currently enabled. If a named PL/SQL block that executes with definer rights queries `SESSION_ROLES`, the query does not return any rows.

See Also: *Oracle9i Database Reference*

Anonymous Blocks with Invoker Rights

Named PL/SQL blocks that execute with invoker rights and anonymous PL/SQL blocks are executed based on privileges granted through enabled roles. Current roles are used for privilege checking within an invoker-rights PL/SQL block, and you can use dynamic SQL to set a role in the session.

See Also:

- *PL/SQL User's Guide and Reference* for an explanation of invoker and definer rights
- "Dynamic SQL in PL/SQL" on page 14-20

Data Definition Language Statements and Roles

A user requires one or more privileges to successfully execute a data definition language (DDL) statement, depending on the statement. For example, to create a table, the user must have the CREATE TABLE or CREATE ANY TABLE system privilege. To create a view of another user's table, the creator requires the CREATE VIEW or CREATE ANY VIEW system privilege and either the SELECT *object* privilege for the table or the SELECT ANY TABLE system privilege.

Oracle avoids the dependencies on privileges received by way of roles by restricting the use of specific privileges in certain DDL statements. The following rules outline these privilege restrictions concerning DDL statements:

- All system privileges and schema object privileges that permit a user to perform a DDL operation are usable when received through a role. For example:
  - System Privileges: the CREATE TABLE, CREATE VIEW and CREATE PROCEDURE privileges.
  - Schema Object Privileges: the ALTER and INDEX privileges for a table.

  Exception: The REFERENCES object privilege for a table cannot be used to define a table's foreign key if the privilege is received through a role.
- All system privileges and object privileges that allow a user to perform a DML operation that is required to issue a DDL statement are *not* usable when received through a role. For example:
  - A user who receives the SELECT ANY TABLE system privilege or the SELECT *object* privilege for a table through a role can use neither privilege to create a view on another user's table.

The following example further clarifies the permitted and restricted uses of privileges received through roles:

Assume that a user is:

- Granted a role that has the CREATE VIEW system privilege

- Granted a role that has the SELECT *object* privilege for the employees table, but the user is indirectly granted the SELECT *object* privilege for the employees table
- Directly granted the SELECT *object* privilege for the departments table

Given these directly and indirectly granted privileges:

- The user can issue SELECT statements on both the employees and departments tables.
- Although the user has both the CREATE VIEW and SELECT privilege for the employees table through a role, the user cannot create a usable view on the employees table, because the SELECT *object* privilege for the employees table was granted through a role. Any views created will produce errors when accessed.
- The user can create a view on the departments table, because the user has the CREATE VIEW privilege through a role and the SELECT privilege for the departments table directly.

Predefined Roles

The following roles are defined automatically for Oracle databases:

- CONNECT
- RESOURCE
- DBA
- EXP_FULL_DATABASE
- IMP_FULL_DATABASE

These roles are provided for backward compatibility to earlier versions of Oracle and can be modified in the same manner as any other role in an Oracle database.

The Operating System and Roles

In some environments, you can administer database security using the operating system. The operating system can be used to manage the granting (and revoking) of database roles and to manage their password authentication. This capability is not available on all operating systems.

> **See Also:** Your operating system specific Oracle documentation for details on managing roles through the operating system

Roles in a Distributed Environment

When you use roles in a distributed database environment, you must ensure that all needed roles are set as the default roles for a distributed (remote) session. You cannot enable roles when connecting to a remote database from within a local database session. For example, you cannot execute a remote procedure that attempts to enable a role at the remote site.

See Also: *Oracle9i Heterogeneous Connectivity Administrator's Guide*

Fine-Grained Access Control

Fine-grained access control lets you implement security policies with functions and then associate those security policies with tables, views, or synonyms. The database server automatically enforces those security policies, no matter how the data is accessed (for example, by ad hoc queries).

You can:

- Use different policies for `SELECT`, `INSERT`, `UPDATE`, and `DELETE`.
- Use security policies only where you need them (for example, on salary information).
- Use more than one policy for each table, including building on top of base policies in packaged applications.
- Distinguish policies between different applications, by using *policy groups*. Each policy group indicates a set of policies that belong to an application.

The database administrator designates an application context, called a *driving context*, to indicate the policy group in effect. When tables, views, or synonyms are accessed, the fine-grained access control engine looks up the driving context to determine the policy group in effect and enforces all the associated policies that belong to that policy group.

The PL/SQL package `DBMS_RLS` let you administer your security policies. Using this package, you can add, drop, enable, disable, and refresh the policies you create.

See Also:

- *Oracle9i Supplied PL/SQL Packages and Types Reference* for information about package implementation
- *Oracle9i Application Developer's Guide - Fundamentals* for information and examples on establishing security policies

Dynamic Predicates

The function or package that implements the security policy you create returns a predicate (a `WHERE` condition). This predicate controls access as set out by the policy. Rewritten queries are fully optimized and shareable.

Application Context

Application context facilitates the implementation of fine-grained access control. It lets you implement security policies with functions and then associate those security policies with applications. Each application can have its own application-specific context. Users are not allowed to arbitrarily change their context (for example, through SQL*Plus).

Application contexts permit flexible, parameter-based access control, based on attributes of interest to an application. For example, context attributes for a human resources application could include "position," "organizational unit," and "country," whereas attributes for an order-entry control might be "customer number" and "sales region".

You can:

- Base predicates on context values
- Use context values within predicates, as bind variables
- Set user attributes
- Access user attributes

To define an application context:

1. Create a PL/SQL package with functions that validate and set the context for your application. You may want to use an event trigger on login to set the initial context for logged-in users.
2. Use `CREATE CONTEXT` to specify a unique context name and associate it with the PL/SQL package you created.
3. Do one of the following:
   - Reference the application context in a policy function implementing fine-grained access control.
   - Create an event trigger on login to set the initial context for a user. For example, you could query a user's employee number and set this as an "employee number" context value.

4. Reference the application context.

See Also:

- *PL/SQL User's Guide and Reference*
- *Oracle9i Supplied PL/SQL Packages and Types Reference*
- *Oracle9i Application Developer's Guide - Fundamentals*

Secure Application Roles

Oracle provides secure application roles, which are roles that can be enabled only by authorized PL/SQL packages. This mechanism restricts the enabling of roles to the invoking application.

In previous releases, passwords were either embedded in the source code or stored in a table. Application developers no longer need to secure a role by embedding passwords inside applications. Instead, they create a secure application role and specify which PL/SQL package is authorized to enable the role. Package identity is used to determine whether there are sufficient privileges to enable the roles. The application performs authentication before enabling the role.

The application can perform customized authorization, such as checking whether the user has connected through a proxy, before enabling the role.

Note: Because of the restriction that users cannot change security domain inside definer's right procedures, secure application roles can only be enabled inside invoker's right procedures.

Creation of Secure Application Roles

Secure application roles are created by using the `CREATE ROLE ... IDENTIFIED USING` statement. Here is an example:

```
CREATE ROLE admin_role IDENTIFIED USING hr.admin;
```

This indicates the following:

- The role `admin_role` to be created is a secure application role.
- The role can only be enabled by any module defined inside the PL/SQL package `hr.admin`.

You must have the system privilege `CREATE ROLE` to execute this statement.

Roles that are enabled inside an Invoker's Right procedure remain in effect even after the procedure exits. Therefore, you can have a dedicated procedure that deals with enabling the role for the rest of the session to use.

See Also:

- *Oracle9i SQL Reference*
- *PL/SQL User's Guide and Reference*
- *Oracle9i Supplied PL/SQL Packages and Types Reference*
- *Oracle9i Application Developer's Guide - Fundamentals*

24

Auditing

This chapter discusses the auditing feature of Oracle. It includes:

- Introduction to Auditing
- Statement Auditing
- Privilege Auditing
- Schema Object Auditing
- Fine-Grained Auditing
- Focus Statement, Privilege, and Schema Object Auditing
- Audit in a Multitier Environment

Introduction to Auditing

Auditing is the monitoring and recording of selected user database actions. Auditing is normally used to:

- Investigate suspicious activity. For example, if an unauthorized user is deleting data from tables, the security administrator might decide to audit all connections to the database and all successful and unsuccessful deletions of rows from all tables in the database.
- Monitor and gather data about specific database activities. For example, the database administrator can gather statistics about which tables are being updated, how many logical I/Os are performed, or how many concurrent users connect at peak times.

Features of Auditing

This section outlines the features of the Oracle auditing mechanism.

Types of Auditing

Oracle supports three general types of auditing:

Type of Auditing	Description
Statement auditing	The selective auditing of SQL statements with respect to only the type of statement, not the specific schema objects on which it operates. Statement auditing options are typically broad, auditing the use of several types of related actions for each option. For example, `AUDIT TABLE` tracks several DDL statements regardless of the table on which they are issued. You can set statement auditing to audit selected users or every user in the database.
Privilege auditing	The selective auditing of the use of powerful system privileges to perform corresponding actions, such as `AUDIT CREATE TABLE`. Privilege auditing is more focused than statement auditing because it audits only the use of the target privilege. You can set privilege auditing to audit a selected user or every user in the database.
Schema object auditing	The selective auditing of specific statements on a particular schema object, such as `AUDIT SELECT ON employees`. Schema object auditing is very focused, auditing only a specific statement on a specific schema object. Schema object auditing always applies to all users of the database.

Type of Auditing	Description
Fine-grained auditing	Fine-grained auditing allows the monitoring of data access based on content.

Focus of Auditing

Oracle allows audit options to be focused or broad. You can audit:

- Successful statement executions, unsuccessful statement executions, or both
- Statement executions once in each user session or once every time the statement is executed
- Activities of all users or of a specific user

Audit Records and the Audit Trail

Audit records include information such as the operation that was audited, the user performing the operation, and the date and time of the operation. Audit records can be stored in either a data dictionary table, called the database audit trail, or an operating system audit trail.

The database audit trail is a single table named `SYS.AUD$` in the `SYS` schema of each Oracle database's data dictionary. Several predefined views are provided to help you use the information in this table.

The audit trail records can contain different types of information, depending on the events audited and the auditing options set. The following information is always included in each audit trail record, if the information is meaningful to the particular audit action:

- The user name
- The session identifier
- The terminal identifier
- The name of the schema object accessed
- The operation performed or attempted
- The completion code of the operation
- The date and time stamp
- The system privileges used

The operating system audit trail is encoded and not readable, but it is decoded in data dictionary files and error messages.

- Action code describes the operation performed or attempted. The AUDIT_ACTIONS data dictionary table contains a list of these codes and their descriptions.
- Privileges used describes any system privileges used to perform the operation. The SYSTEM_PRIVILEGE_MAP table lists all of these codes and their descriptions.
- Completion code describes the result of the attempted operation. Successful operations return a value of zero, and unsuccessful operations return the Oracle error code describing why the operation was unsuccessful.

See Also:

- *Oracle9i Database Administrator's Guide* for instructions for creating and using predefined views
- *Oracle9i Database Error Messages* for a list of completion codes

Mechanisms for Auditing

This section explains the mechanisms used by the Oracle auditing features.

When Are Audit Records Generated?

The recording of audit information can be enabled or disabled. This functionality allows any authorized database user to set audit options at any time but reserves control of recording audit information for the security administrator.

When auditing is enabled in the database, an audit record is generated during the execute phase of statement execution.

SQL statements inside PL/SQL program units are individually audited, as necessary, when the program unit is executed.

The generation and insertion of an audit trail record is independent of a user's transaction. Therefore, even if a user's transaction is rolled back, the audit trail record remains committed.

Note: Operations by the SYS user and by users connected through SYSDBA or SYSOPER can be fully audited with the AUDIT_SYS_OPERATIONS initialization parameter. Successful SQL statements from SYS are audited indiscriminately.

The audit records for sessions established by the user SYS or connections with administrative privileges are sent to an operating system location. Sending them to a location separate from the usual database audit trail in the SYS schema provides for greater auditing security.

See Also:

- *Oracle9i Database Administrator's Guide* for instructions on enabling and disabling auditing
- Chapter 14, "SQL, PL/SQL, and Java" for information about the different phases of SQL statement processing and shared SQL

Events Always Audited to the Operating System Audit Trail

Regardless of whether database auditing is enabled, Oracle always records some database-related actions into the operating system audit trail:

- At instance startup, an audit record is generated that details the operating system user starting the instance, the user's terminal identifier, the date and time stamp, and whether database auditing was enabled or disabled. This information is recorded into the operating system audit trail because the database audit trail is not available until after startup has successfully completed. Recording the state of database auditing at startup further prevents an administrator from restarting a database with database auditing disabled so that they are able to perform unaudited actions.
- At instance shutdown, an audit record is generated that details the operating system user shutting down the instance, the user's terminal identifier, the date and time stamp.
- During connections with administrator privileges, an audit record is generated that details the operating system user connecting to Oracle with administrator privileges. This provides accountability of users connected with administrator privileges.

On operating systems that do not make an audit trail accessible to Oracle, these audit trail records are placed in an Oracle audit trail file in the same directory as background process trace files.

> **See Also:** Your operating system specific Oracle documentation for more information about the operating system audit trail

When Do Audit Options Take Effect?

Statement and privilege audit options in effect at the time a database user connects to the database remain in effect for the duration of the session. A session does not see the effects of statement or privilege audit options being set or changed. The modified statement or privilege audit options take effect only when the current session is ended and a new session is created. In contrast, changes to schema object audit options become effective for current sessions immediately.

Audit in a Distributed Database

Auditing is site autonomous. An instance audits only the statements issued by directly connected users. A local Oracle node cannot audit actions that take place in a remote database. Because remote connections are established through the user account of a database link, the remote Oracle node audits the statements issued through the database link's connection.

> **See Also:** *Oracle9i Database Administrator's Guide*

Audit to the Operating System Audit Trail

Oracle allows audit trail records to be directed to an operating system audit trail if the operating system makes such an audit trail available to Oracle. On some other operating systems, these audit records are written to a file outside the database, with a format similar to other Oracle trace files.

> **See Also:** Your operating system specific Oracle documentation, to see if this feature has been implemented on your operating system

Oracle allows certain actions that are *always* audited to continue, even when the operating system audit trail (or the operating system file containing audit records) is unable to record the audit record. The usual cause of this is that the operating system audit trail or the file system is full and unable to accept new records.

System administrators configuring operating system auditing should ensure that the audit trail or the file system does not fill completely. Most operating systems

provide administrators with sufficient information and warning to ensure this does not occur. Note, however, that configuring auditing to use the database audit trail removes this vulnerability, because the Oracle server prevents audited events from occurring if the audit trail is unable to accept the database audit record for the statement.

Statement Auditing

Statement auditing is the selective auditing of related groups of statements that fall into two categories:

- DDL statements, regarding a particular type of database structure or schema object, but not a specifically named structure or schema object (for example, `AUDIT TABLE` audits all `CREATE` and `DROP TABLE` statements)
- DML statements, regarding a particular type of database structure or schema object, but not a specifically named structure or schema object (for example, `AUDIT SELECT TABLE` audits all `SELECT ... FROM TABLE/VIEW` statements, regardless of the table or view)

Statement auditing can be broad or focused, auditing the activities of all database users or the activities of only a select list of database users.

Privilege Auditing

Privilege auditing is the selective auditing of the statements allowed using a system privilege. For example, auditing of the `SELECT ANY TABLE` system privilege audits users' statements that are executed using the `SELECT ANY TABLE` system privilege. You can audit the use of any system privilege.

In all cases of privilege auditing, owner privileges and schema object privileges are checked before system privileges. If the owner and schema object privileges suffice to permit the action, the action is not audited.

If similar statement and privilege audit options are both set, only a single audit record is generated. For example, if the statement clause `TABLE` and the system privilege `CREATE TABLE` are both audited, only a single audit record is generated each time a table is created.

Privilege auditing is more focused than statement auditing because each option audits only specific types of statements, not a related list of statements. For example, the statement auditing clause `TABLE` audits `CREATE TABLE`, `ALTER TABLE`, and `DROP TABLE` statements, while the privilege auditing option `CREATE`

TABLE audits only CREATE TABLE statements. This is because only the CREATE TABLE statement requires the CREATE TABLE privilege.

Like statement auditing, privilege auditing can audit the activities of all database users or the activities of a select list of database users.

Schema Object Auditing

Schema object auditing is the selective auditing of specific DML statements (including queries) and GRANT and REVOKE statements for specific schema objects. Schema object auditing audits the operations permitted by schema object privileges, such as SELECT or DELETE statements on a given table, as well as the GRANT and REVOKE statements that control those privileges.

You can audit statements that reference tables, views, sequences, **standalone** stored procedures and functions, and packages. Procedures in packages cannot be audited individually.

Statements that reference clusters, database links, indexes, or synonyms are not audited directly. However, you can audit access to these schema objects indirectly by auditing the operations that affect the base table.

Schema object audit options are always set for all users of the database. These options cannot be set for a specific list of users. You can set default schema object audit options for all auditable schema objects.

> **See Also:** *Oracle9i SQL Reference* for information about auditable schema objects

Schema Object Audit Options for Views and Procedures

Views and procedures (including stored functions, packages, and triggers) reference underlying schema objects in their definition. Therefore, auditing with respect to views and procedures has several unique characteristics. Multiple audit records can be generated as the result of using a view or a procedure: The use of the view or procedure is subject to enabled audit options, and the SQL statements issued as a result of using the view or procedure are subject to the enabled audit options of the base schema objects (including default audit options).

Consider the following series of SQL statements:

```
AUDIT SELECT ON employees;

CREATE VIEW employees_departments AS
  SELECT employee_id, last_name, department_id
```

```
    FROM employees, departments
    WHERE employees.department_id = departments.department_id;

AUDIT SELECT ON employees_departments;

SELECT * FROM employees_departments;
```

As a result of the query on `employees_departments`, two audit records are generated: one for the query on the `employees_departments` view and one for the query on the base table `employees` (indirectly through the `employees_departments` view). The query on the base table `departments` does not generate an audit record because the `SELECT` audit option for this table is not enabled. All audit records pertain to the user that queried the `employees_departments` view.

The audit options for a view or procedure are determined when the view or procedure is first used and placed in the shared pool. These audit options remain set until the view or procedure is flushed from, and subsequently replaced in, the shared pool. Auditing a schema object invalidates that schema object in the cache and causes it to be reloaded. Any changes to the audit options of base schema objects are not observed by views and procedures in the shared pool.

Continuing with the previous example, if auditing of `SELECT` statements is turned off for the `employees` table, use of the `employees_departments` view no longer generates an audit record for the `employees` table.

Fine-Grained Auditing

Fine-grained auditing allows the monitoring of data access based on content. A built-in audit mechanism in the database prevents users from by-passing the audit. Oracle triggers can potentially monitor DML actions such as `INSERT`, `UPDATE`, and `DELETE`. However, monitoring on `SELECT` is costly and might not work for certain cases. In addition, users might want to define their own alert action in addition to just inserting an audit record into the audit trail. This feature provides an extensible interface to audit `SELECT` statements on tables and views.

The `DBMS_FGA` package administers these value-based audit policies. Using `DBMS_FGA`, the security administrator creates an audit policy on the target table. If any of the rows returned from a query block matches the audit condition (these rows are referred to as interested rows*)*, then an audit event entry, including username, SQL text, bind variable, policy name, session ID, time stamp, and other attributes, is inserted into the audit trail. As part of the extensibility framework, administrators can also optionally define an appropriate event handler, an audit event handler, to

process the event; for example, the audit event handler could send an alert page to the administrator.

See Also: *Oracle9i Application Developer's Guide - Fundamentals*

Focus Statement, Privilege, and Schema Object Auditing

Oracle lets you focus statement, privilege, and schema object auditing in three areas:

- Successful and unsuccessful executions of the audited SQL statement
- `BY SESSION` and `BY ACCESS` auditing
- For specific users or for all users in the database (statement and privilege auditing only)

Successful and Unsuccessful Statement Executions Auditing

For statement, privilege, and schema object auditing, Oracle allows the selective auditing of successful executions of statements, unsuccessful attempts to execute statements, or both. Therefore, you can monitor actions even if the audited statements do not complete successfully.

You can audit an unsuccessful statement execution only if a valid SQL statement is issued but fails because of lack of proper authorization or because it references a nonexistent schema object. Statements that failed to execute because they simply were not valid cannot be audited. For example, an enabled privilege auditing option set to audit unsuccessful statement executions audits statements that use the target system privilege but have failed for other reasons (such as when `CREATE TABLE` is set but a `CREATE TABLE` statement fails due to lack of quota for the specified tablespace).

Using either form of the `AUDIT` statement, you can include:

- The `WHENEVER SUCCESSFUL` clause, to audit only successful executions of the audited statement
- The `WHENEVER NOT SUCCESSFUL` clause, to audit only unsuccessful executions of the audited statement
- Neither of the previous clauses, to audit both successful and unsuccessful executions of the audited statement

BY SESSION and BY ACCESS Clauses of Audit Statement

Most auditing options can be set to indicate how audit records should be generated if the audited statement is issued multiple times in a single user session. This section describes the distinction between the BY SESSION and BY ACCESS clauses of the AUDIT statement.

See Also: *Oracle9i SQL Reference*

BY SESSION

For any type of audit (schema object, statement, or privilege), BY SESSION inserts only one audit record in the audit trail, for each user and schema object, during the session that includes an audited action.

A session is the time between when a user connects to and disconnects from an Oracle database.

BY SESSION Example 1 Assume the following:

- The SELECT TABLE statement auditing option is set BY SESSION.
- JWARD connects to the database and issues five SELECT statements against the table named departments and then disconnects from the database.
- SWILLIAMS connects to the database and issues three SELECT statements against the table employees and then disconnects from the database.

In this case, the audit trail contains two audit records for the eight SELECT statements- one for each session that issued a SELECT statement.

BY SESSION Example 2 Alternatively, assume the following:

- The SELECT TABLE statement auditing option is set BY SESSION.
- JWARD connects to the database and issues five SELECT statements against the table named departments, and three SELECT statements against the table employees, and then disconnects from the database.

In this case, the audit trail contains two records- one for each schema object against which the user issued a SELECT statement in a session.

Note: If you use the BY SESSION clause when directing audit records to the operating system audit trail, Oracle generates and stores an audit record each time an access is made. Therefore, in this auditing configuration, BY SESSION is equivalent to BY ACCESS.

BY ACCESS

Setting audit BY ACCESS inserts one audit record into the audit trail for each execution of an auditable operation within a cursor. Events that cause cursors to be reused include the following:

- An application, such as Oracle Forms, holding a cursor open for reuse
- Subsequent execution of a cursor using new bind variables
- Statements executed within PL/SQL loops where the PL/SQL engine optimizes the statements to reuse a single cursor

Note that auditing is not affected by whether a cursor is shared. Each user creates her or his own audit trail records on first execution of the cursor.

For example, assume that:

- The SELECT TABLE statement auditing option is set BY ACCESS.
- JWARD connects to the database and issues five SELECT statements against the table named departments and then disconnects from the database.
- SWILLIAMS connects to the database and issues three SELECT statements against the table departments and then disconnects from the database.

The single audit trail contains eight records for the eight SELECT statements.

Defaults and Excluded Operations

The AUDIT statement lets you specify either BY SESSION or BY ACCESS. However, several audit options can be set only BY ACCESS, including:

- All statement audit options that audit DDL statements
- All privilege audit options that audit DDL statements

For all other audit options, BY SESSION is used by default.

Audit By User

Statement and privilege audit options can audit statements issued by any user or statements issued by a specific list of users. By focusing on specific users, you can minimize the number of audit records generated.

Audit By User Example To audit statements by the users SCOTT and BLAKE that query or update a table or view, issue the following statements:

```
AUDIT SELECT TABLE, UPDATE TABLE
     BY scott, blake;
```

> **See Also:** *Oracle9i SQL Reference* for more information about auditing by user

Audit in a Multitier Environment

In a multitier environment, Oracle preserves the identity of a client through all tiers. This enables auditing of actions taken on behalf of the client. To do so, use the BY proxy clause in your AUDIT statement.

This clause allows you a few options. You can:

- Audit SQL statements issued by the specific proxy on its own behalf
- Audit statements executed on behalf of a specified user or users
- Audit all statements executed on behalf of any user

The middle tier can set the light-weight user identity in a database session so that it will show up in audit trail. You use OCI or PL/SQL to set the client identifier.

> **See Also:**
>
> - *Oracle9i Application Developer's Guide - Fundamentals*
> - *Oracle Call Interface Programmer's Guide*
> - *PL/SQL User's Guide and Reference*