Part III
Oracle Transaction Management

Transaction management is the use of transactions to ensure data concurrency and consistency.

This part contains the following chapters:

9

Data Concurrency and Consistency

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

This chapter contains the following sections:

Introduction to Data Concurrency and Consistency

In a single-user database, a user can modify data without concern for other users modifying the same data at the same time. However, in a multiuser database, statements within multiple simultaneous transactions may update the same data. Transactions executing simultaneously must produce meaningful and consistent results.

A multiuser database must provide the following:

- The assurance that users can access data at the same time (data concurrency)
- The assurance that each user sees a consistent view of the data (data consistency), including visible changes made by the user's own transactions and committed transactions of other users

To describe consistent transaction behavior when transactions run concurrently, database researchers have defined a transaction isolation model called serializability. A serializable transaction operates in an environment that makes it appear as if no other users were modifying data in the database.

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

Oracle Database maintains data consistency by using a multiversion consistency model and various types of locks and transactions. In this way, the database can present a view of data to multiple concurrent users, with each view consistent to a point in time. Because different versions of data blocks can exist simultaneously, transactions can read the version of data committed at the point in time required by a query and return results that are consistent to a single point in time.

> **See Also:**
>
> "Data Integrity" and "Transactions "

Multiversion Read Consistency

In Oracle Database, multiversioning is the ability to simultaneously materialize multiple versions of data. Oracle Database maintains multiversion read consistency.

Queries of an Oracle database have the following characteristics:

- Read-consistent queries

  The data returned by a query is committed and consistent for a single point in time.

  > **Note:**
  >
  > Oracle Database *never* permits a dirty read, which occurs when a transaction reads uncommitted data in another transaction.

  To illustrate the problem with dirty reads, suppose one transaction updates a column value without committing. A second transaction reads the updated and dirty (uncommitted) value. The first session rolls back the transaction so that the column has its old value, but the second transaction proceeds using the updated value, corrupting the database. Dirty reads compromise data integrity, violate foreign keys, and ignore unique constraints.
- Nonblocking queries

  Readers and writers of data do not block one another.

> **See Also:**
>
> "Summary of Locking Behavior"

Statement-Level Read Consistency

Oracle Database always enforces **statement-level read consistency**, which guarantees that data returned by a single query is committed and consistent for a single point in time.

The point in time to which a single SQL statement is consistent depends on the transaction isolation level and the nature of the query:

- In the read committed isolation level, this point is the time at which the *statement* was opened. For example, if a `SELECT` statement opens at SCN 1000, then this statement is consistent to SCN 1000.

- In a serializable or read-only transaction, this point is the time the *transaction* began. For example, if a transaction begins at SCN 1000, and if multiple `SELECT` statements occur in this transaction, then each statement is consistent to SCN 1000.
- In a Flashback Query operation (`SELECT ... AS OF`), the `SELECT` statement explicitly specifies the point in time. For example, you can query a table as it appeared last Thursday at 2 p.m.

> **See Also:**
>
> *Oracle Database Development Guide* to learn about Flashback Query

Transaction-Level Read Consistency

Oracle Database can also provide read consistency to all queries in a transaction, known as **transaction-level read consistency**.

In this case, each statement in a transaction sees data from the *same* point in time. This is the time at which the transaction began.

Queries made by a serializable transaction see changes made by the transaction itself. For example, a transaction that updates `employees` and then queries `employees` will see the updates. Transaction-level read consistency produces repeatable reads and does not expose a query to phantom reads.

Read Consistency and Undo Segments

To manage the multiversion read consistency model, the database must create a read-consistent set of data when a table is simultaneously queried and updated.

Oracle Database achieves read consistency through undo data.

Whenever a user modifies data, Oracle Database creates undo entries, which it writes to undo segments. The undo segments contain the old values of data that have been changed by uncommitted or recently committed transactions. Thus, multiple versions of the same data, all at different points in time, can exist in the database. The database can use snapshots of data at different points in time to provide read-consistent views of the data and enable nonblocking queries.

Read consistency is guaranteed in single-instance and Oracle Real Application Clusters (Oracle RAC) environments. Oracle RAC uses a cache-to-cache block transfer mechanism known as cache fusion to transfer read-consistent images of data blocks from one database instance to another.

See Also:

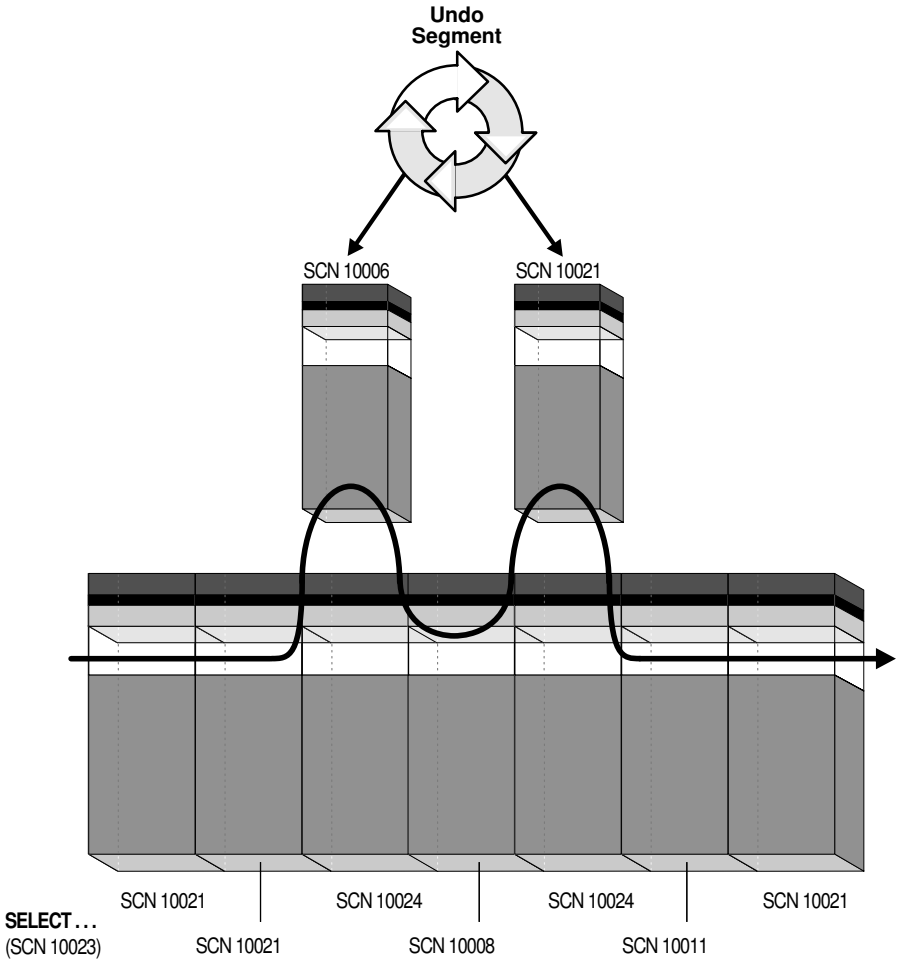
- "Undo Segments"

  to learn about undo storage
- "Internal LOBs" to learn about read consistency mechanisms for LOBs
- *Oracle Database 2 Day + Real Application Clusters Guide* to learn about cache fusion

Read Consistency: Example

This example shows a query that uses undo data to provide statement-level read consistency in the read committed isolation level.

Figure 9-1 Read Consistency in the Read Committed Isolation Level



As the database retrieves data blocks on behalf of a query, the database ensures that the data in each block reflects the contents of the block when the query began. The

database rolls back changes to the block as needed to reconstruct the block to the point in time the query started processing.

The database uses an internal ordering mechanism called an SCN to guarantee the order of transactions. As the `SELECT` statement enters the execution phase, the database determines the SCN recorded at the time the query began executing. In Figure 9-1, this SCN is 10023. The query only sees committed data with respect to SCN 10023.

In Figure 9-1, blocks with SCNs *after* 10023 indicate changed data, as shown by the two blocks with SCN 10024. The `SELECT` statement requires a version of the block that is consistent with committed changes. The database copies current data blocks to a new buffer and applies undo data to reconstruct previous versions of the blocks. These reconstructed data blocks are called *consistent read (CR) clones*.

In Figure 9-1, the database creates two CR clones: one block consistent to SCN 10006 and the other block consistent to SCN 10021. The database returns the reconstructed data for the query. In this way, Oracle Database prevents dirty reads.

> **See Also:**
>
> "Database Buffer Cache" and "System Change Numbers (SCNs)"

Read Consistency and Interested Transaction Lists

The **block header** of every segment block contains an **interested transaction list (ITL)**.

The database uses the ITL to determine whether a transaction was uncommitted when the database began modifying the block.

Entries in the ITL describe which transactions have rows locked and which rows in the block contain committed and uncommitted changes. The ITL points to the transaction table in the undo segment, which provides information about the timing of changes made to the database.

In a sense, the block header contains a recent history of transactions that affected each row in the block. The `INITRANS` parameter of the `CREATE TABLE` and `ALTER TABLE` statements controls the amount of transaction history that is kept.

> **See Also:**
>
> *Oracle Database SQL Language Reference* to learn about the `INITRANS` parameter

Locking Mechanisms

In general, multiuser databases use some form of data locking to solve the problems associated with data concurrency, consistency, and integrity.

A lock is a mechanism that prevent destructive interaction between transactions accessing the same resource.

> **See Also:**
>
> "Overview of the Oracle Database Locking Mechanism"

ANSI/ISO Transaction Isolation Levels

The SQL standard, which has been adopted by both ANSI and ISO/IEC, defines four levels of transaction isolation. These levels have differing degrees of impact on transaction processing throughput.

These isolation levels are defined in terms of phenomena that must be prevented between concurrently executing transactions. The 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. For example, a user queries a row and then later queries the same row, only to discover that the data has changed.

- Phantom reads

  A transaction reruns 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.

  For example, a transaction queries the number of employees. Five minutes later it performs the same query, but now the number has increased by one because another user inserted a record for a new hire. More data satisfies the query criteria than before, but unlike in a fuzzy read the previously read data is unchanged.

The SQL standard defines four levels of isolation in terms of the phenomena that a transaction running at a particular isolation level is permitted to experience. Table 9-1 shows the levels.

Table 9-1 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 Database offers the read committed (default) and serializable isolation levels. Also, the database offers a read-only mode.

> **See Also:**
>
> - "Overview of Oracle Database Transaction Isolation Levels" to learn about read committed, serializable, and read-only isolation levels
> - *Oracle Database SQL Language Reference* for a discussion of Oracle Database conformance to SQL standards

Overview of Oracle Database Transaction Isolation Levels

The ANSI standard for transaction isolation levels is defined in terms of the phenomena that are either permitted or prevented for each isolation level.

Oracle Database provides the transaction isolation levels:

- Read Committed Isolation Level
- Serializable Isolation Level
- Read-Only Isolation Level

> **See Also:**
>
> - *Oracle Database Development Guide* to learn more about transaction isolation levels
> - *Oracle Database SQL Language Reference* and *Oracle Database PL/SQL Language Reference* to learn about `SET TRANSACTION ISOLATION LEVEL`

Read Committed Isolation Level

In the **read committed isolation level**, every query executed by a transaction sees only data committed before the query—not the transaction—began.

This isolation level is the default. It is appropriate for database environments in which few transactions are likely to conflict.

A query in a read committed transaction avoids reading data that commits while the query is in progress. For example, if a query is halfway through a scan of a million-row table, and if a different transaction commits an update to row 950,000, then the query does not see this change when it reads row 950,000. However, because the database does not prevent other transactions from modifying data read by a query, other transactions may change data *between* query executions. Thus, a transaction that runs the same query twice may experience fuzzy reads and phantoms.

Read Consistency in the Read Committed Isolation Level

The database provides a consistent result set for every query, guaranteeing data consistency, with no action by the user.

An implicit query, such as a query implied by a `WHERE` clause in an `UPDATE` statement, is guaranteed a consistent set of results. However, each statement in an implicit query does not see the changes made by the DML statement itself, but sees the data as it existed before changes were made.

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

> **See Also:**
>
> "Subqueries"

Conflicting Writes in Read Committed Transactions

In a read committed transaction, a **conflicting write** occurs when the transaction attempts to change a row updated by an uncommitted concurrent transaction.

The transaction that prevents the row modification is sometimes called a *blocking transaction*. The read committed transaction waits for the blocking transaction to end and release its row lock.

The options are as follows:

- If the blocking transaction rolls back, then the waiting transaction proceeds to change the previously locked row as if the other transaction never existed.
- If the blocking transaction commits and releases its locks, then the waiting transaction proceeds with its intended update to the newly changed row.

The following table shows how transaction 1, which can be either serializable or read committed, interacts with read committed transaction 2. It shows a classic situation known as a lost update. The update made by transaction 1 is not in the table *even though transaction 1 committed it*. Devising a strategy to handle lost updates is an important part of application development.

Table 9-2 Conflicting Writes and Lost Updates in a READ COMMITTED Transaction

Session 1	Session 2	Explanation

`SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda', 'Greene', 'Hintz');`
 	
`LAST_NAME SALARY`
`------------- ----------`
`Banda 6200`
`Greene 9500`
	No action.	Session 1 queries the salaries for Banda, Greene, and Hintz. No employee named Hintz is found.

Table 9-2 (Cont.) Conflicting Writes and Lost Updates in a READ COMMITTED Transaction

Session 1	Session 2	Explanation
`SQL> UPDATE employees SET salary = 7000 WHERE last_name = 'Banda';`	No action.	Session 1 begins a transaction by updating the Banda salary. The default isolation level for transaction 1 is `READ COMMITTED`.
No action.	`SQL> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;`	Session 2 begins transaction 2 and sets the isolation level explicitly to `READ COMMITTED`.

No action.	`SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda', 'Greene', 'Hintz');`
	
	`LAST_NAME SALARY`
	`------------ ----------`
	`Banda 6200`
	`Greene 9500`
		Transaction 2 queries the salaries for Banda, Greene, and Hintz. Oracle Database uses read consistency to show the salary for Banda before the uncommitted update made by transaction 1.

No action.	`SQL> UPDATE employees SET salary = 9900 WHERE last_name='Greene';`	Transaction 2 updates the salary for Greene successfully because transaction 1 locked only the Banda row.
`SQL> INSERT INTO employees (employee_id, last_name, email, hire_date, job_id) VALUES (210, 'Hintz', 'JHINTZ', SYSDATE, 'SH_CLERK');`	No action.	Transaction 1 inserts a row for employee Hintz, but does not commit.

No action.	`SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda', 'Greene', 'Hintz');`
	
	`LAST_NAME SALARY`
	`------------ ----------`
	`Banda 6200`
	`Greene 9900`
		Transaction 2 queries the salaries for employees Banda, Greene, and Hintz.
		Transaction 2 sees its own update to the salary for Greene. Transaction 2 does not see the uncommitted update to the salary for Banda or the insertion for Hintz made by transaction 1.

No action.	`SQL> UPDATE employees SET salary = 6300 WHERE last_name = 'Banda';`
	
	`-- prompt does not return`
		Transaction 2 attempts to update the row for Banda, which is currently locked by transaction 1, creating a conflicting write. Transaction 2 waits until transaction 1 ends.

`SQL> COMMIT;`	No action.	Transaction 1 commits its work, ending the transaction.

Table 9-2 (Cont.) Conflicting Writes and Lost Updates in a READ COMMITTED Transaction

Session 1	Session 2	Explanation

No action.	`1 row updated.`
	
	`SQL>`
		The lock on the Banda row is now released, so transaction 2 proceeds with its update to the salary for Banda.

No action.	`SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda', 'Greene', 'Hintz');`
	
	`LAST_NAME SALARY`
	`------------ ----------`
	`Banda 6300`
	`Greene 9900`
	`Hintz`
		Transaction 2 queries the salaries for employees Banda, Greene, and Hintz. The Hintz insert committed by transaction 1 is now visible to transaction 2. Transaction 2 sees its own update to the Banda salary.

No action.	`COMMIT;`	Transaction 2 commits its work, ending the transaction.

`SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda', 'Greene', 'Hintz');`
 	
`LAST_NAME SALARY`
`------------ ----------`
`Banda 6300`
`Greene 9900`
`Hintz`
	No action.	Session 1 queries the rows for Banda, Greene, and Hintz. The salary for Banda is 6300, which is the update made by transaction 2. The update of Banda's salary to 7000 made by transaction 1 is now "lost."

See Also:

- "Use of Locks" to learn about lost updates
- "Row Locks (TX)" to learn when and why the database obtains row locks

Serializable Isolation Level

In the **serializable isolation level**, a transaction sees only changes committed at the time the transaction—not the query—began and changes made by the transaction itself.

A serializable transaction operates in an environment that makes it appear as if no other users were modifying data in the database. 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

In serializable isolation, the read consistency normally obtained at the statement level extends to the entire transaction. Any row read by the transaction is assured to be the same when reread. Any query is guaranteed to return the same results for the duration of the transaction, so changes made by other transactions are not visible to the query regardless of how long it has been running. Serializable transactions do not experience dirty reads, fuzzy reads, or phantom reads.

Oracle Database permits a serializable transaction to modify a row only if changes to the row made by other transactions were *already* committed when the serializable transaction began. The database generates an error when a serializable transaction tries to update or delete data changed by a different transaction that committed *after* the serializable transaction began:

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

When a serializable transaction fails with the `ORA-08177` error, an application can take several actions, including the following:

- 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

The following table shows how a serializable transaction interacts with other transactions. If the serializable transaction does not try to change a row committed by another transaction after the serializable transaction began, then a serialized access problem is avoided.

Table 9-3 Serializable Transaction

Session 1	Session 2	Explanation

`SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda', 'Greene', 'Hintz');`
 	
`LAST_NAME SALARY`
`------------ ----------`
`Banda 6200`
`Greene 9500`
	No action.	Session 1 queries the salaries for Banda, Greene, and Hintz. No employee named Hintz is found.

`SQL> UPDATE employees SET salary = 7000 WHERE last_name='Banda';`	No action.	Session 1 begins transaction 1 by updating the Banda salary. The default isolation level is `READ COMMITTED`.
No action.	`SQL> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;`	Session 2 begins transaction 2 and sets it to the `SERIALIZABLE` isolation level.

Table 9-3 (Cont.) Serializable Transaction

Session 1	Session 2	Explanation

No action.	`SQL> SELECT last_name, salary`
	`FROM employees`
	`WHERE last_name IN`
	`('Banda', 'Greene', 'Hintz');`
	
	`LAST_NAME SALARY`
	`---------- ----------`
	`Banda 6200`
	`Greene 9500`
		Transaction 2 queries the salaries for Banda, Greene, and Hintz. Oracle Database uses read consistency to show the salary for Banda *before* the uncommitted update made by transaction 1.

No action.	`SQL> UPDATE employees`
	`SET salary = 9900`
	`WHERE last_name = 'Greene';`
		Transaction 2 updates the Greene salary successfully because only the Banda row is locked.

`SQL> INSERT INTO employees`
`(employee_id, last_name,`
`email, hire_date, job_id)`
`VALUES (210, 'Hintz',`
`'JHINTZ', SYSDATE,'SH_CLERK');`
	No action.	Transaction 1 inserts a row for employee Hintz.

`SQL> COMMIT;`	No action.	Transaction 1 commits its work, ending the transaction.

`SQL> SELECT last_name, salary`
`FROM employees`
`WHERE last_name IN`
`('Banda', 'Greene', 'Hintz');`
 	
`LAST_NAME SALARY`
`---------- ----------`
`Banda 7000`
`Greene 9500`
`Hintz`
	`SQL> SELECT last_name, salary`
	`FROM employees`
	`WHERE last_name IN`
	`('Banda', 'Greene', 'Hintz');`
	
	`LAST_NAME SALARY`
	`---------- ----------`
	`Banda 6200`
	`Greene 9900`
		Session 1 queries the salaries for employees Banda, Greene, and Hintz and sees changes committed by transaction 1. Session 1 does not see the uncommitted Greene update made by transaction 2.
		
		Transaction 2 queries the salaries for employees Banda, Greene, and Hintz. Oracle Database read consistency ensures that the Hintz insert and Banda update committed by transaction 1 are *not* visible to transaction 2. Transaction 2 sees its own update to the Greene salary.

No action.	`COMMIT;`	Transaction 2 commits its work, ending the transaction.

Table 9-3 (Cont.) Serializable Transaction

Session 1	Session 2	Explanation

`SQL> SELECT last_name, salary`
`FROM employees`
`WHERE last_name IN`
`('Banda', 'Greene', 'Hintz');`
 	
`LAST_NAME SALARY`
`---------- ----------`
`Banda 7000`
`Greene 9900`
`Hintz`
	`SQL> SELECT last_name, salary`
	`FROM employees`
	`WHERE last_name IN`
	`('Banda', 'Greene', 'Hintz');`
	
	`LAST_NAME SALARY`
	`---------- ---------`
	`Banda 7000`
	`Greene 9900`
	`Hintz`
		Both sessions query the salaries for Banda, Greene, and Hintz. Each session sees all committed changes made by transaction 1 and transaction 2.

`SQL> UPDATE employees`
`SET salary = 7100`
`WHERE last_name = 'Hintz';`
	No action.	Session 1 begins transaction 3 by updating the Hintz salary. The default isolation level for transaction 3 is `READ COMMITTED`.

No action.	`SQL> SET TRANSACTION` `ISOLATION LEVEL SERIALIZABLE;`	Session 2 begins transaction 4 and sets it to the `SERIALIZABLE` isolation level.

No action.	`SQL> UPDATE employees`
	`SET salary = 7200`
	`WHERE last_name = 'Hintz';`
	
	`-- prompt does not return`
		Transaction 4 attempts to update the salary for Hintz, but is blocked because transaction 3 locked the Hintz row. Transaction 4 queues behind transaction 3.

`SQL> COMMIT;`	No action.	Transaction 3 commits its update of the Hintz salary, ending the transaction.

No action.	`UPDATE employees`
	`SET salary = 7200`
	`WHERE last_name = 'Hintz'`
	`*`
	`ERROR at line 1:`
	`ORA-08177: can't`
	`serialize access for`
	`this transaction`
		The commit that ends transaction 3 causes the Hintz update in transaction 4 to fail with the `ORA-08177` error. The problem error occurs because transaction 3 committed the Hintz update *after* transaction 4 began.

No action.	`SQL> ROLLBACK;`	Session 2 rolls back transaction 4, which ends the transaction.
No action.	`SQL> SET TRANSACTION` `ISOLATION LEVEL SERIALIZABLE;`	Session 2 begins transaction 5 and sets it to the `SERIALIZABLE` isolation level.

Table 9-3 (Cont.) Serializable Transaction

Session 1	Session 2	Explanation

No action.	`SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda', 'Greene', 'Hintz');`
	
	`LAST_NAME SALARY`
	`------------ ----------`
	`Banda 7000`
	`Greene 9500`
	`Hintz 7100`
		Transaction 5 queries the salaries for Banda, Greene, and Hintz. The Hintz salary update committed by transaction 3 is visible.

No action.	`SQL> UPDATE employees SET salary = 7200 WHERE last_name='Hintz';`
	
	`1 row updated.`
		Transaction 5 updates the Hintz salary to a different value. Because the Hintz update made by transaction 3 committed *before* the start of transaction 5, the serialized access problem is avoided.
		Note: If a different transaction updated and committed the Hintz row after transaction 5 began, then the serialized access problem would occur again.

No action.	`SQL> COMMIT;`	Session 2 commits the update without any problems, ending the transaction.

See Also:

- "Row Locks (TX)"
- "Overview of Transaction Control"

Read-Only Isolation Level

The **read-only isolation level** is similar to the serializable isolation level, but read-only transactions do not permit data to be modified in the transaction unless the user is `SYS`.

Read-only transactions are not susceptible to the `ORA-08177` error. Read-only transactions are useful for generating reports in which the contents must be consistent with respect to the time when the transaction began.

Oracle Database achieves read consistency by reconstructing data as needed from the undo segments. Because undo segments are used in a circular fashion, the database can overwrite undo data. Long-running reports run the risk that undo data required for read consistency may have been reused by a different transaction, raising a `snapshot too old` error. Setting an undo retention period, which is the minimum amount of time that the database attempts to retain old undo data before overwriting it, appropriately avoids this problem.

> **See Also:**
>
> - "Undo Segments"
> - *Oracle Database Administrator's Guide* to learn how to set the undo retention period

Overview of the Oracle Database Locking Mechanism

A **lock** is a mechanism that prevents destructive interactions.

Interactions are destructive when they incorrectly update data or incorrectly alter underlying data structures, between transactions accessing shared data. Locks play a crucial role in maintaining database concurrency and consistency.

Summary of Locking Behavior

The database maintains several different types of locks, depending on the operation that acquired the lock.

In general, the database uses two types of locks: exclusive locks and share locks. Only one exclusive lock can be obtained on a resource such as a row or a table, but many share locks can be obtained on a single resource.

Locks affect the interaction of readers and writers. A reader is a query of a resource, whereas a writer is a statement modifying a resource. The following rules summarize the locking behavior of Oracle Database for readers and writers:

- A row is locked only when modified by a writer.

  When a statement updates one row, the transaction acquires a lock for this row only. By locking table data at the row level, the database minimizes contention for the same data. Under normal circumstances[1] the database does not escalate a row lock to the block or table level.

- A writer of a row blocks a concurrent writer of the same row.

  If one transaction is modifying a row, then a row lock prevents a different transaction from modifying the same row simultaneously.

- A reader never blocks a writer.

  Because a reader of a row does not lock it, a writer can modify this row. The only exception is a `SELECT ... FOR UPDATE` statement, which is a special type of `SELECT` statement that *does* lock the row that it is reading.

- A writer never blocks a reader.

  When a row is being changed by a writer, the database uses undo data to provide readers with a consistent view of the row.

[1] When processing a distributed two-phase commit, the database may briefly prevent read access in special circumstances. Specifically, if a query starts between the prepare and commit phases and attempts to read the data before the commit, then the database may escalate a lock from row-level to block-level to guarantee read consistency.

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

> **See Also:**
>
> - *Oracle Database SQL Language Reference* to learn about `SELECT ... FOR UPDATE`
> - *Oracle Database Administrator's Guide* to learn about waits associated with in-doubt distributed transactions

Use of Locks

In a single-user database, locks are not necessary because only one user is modifying information. However, when multiple users are accessing and modifying data, the database must provide a way to prevent concurrent modification of the same data.

Locks achieve the following important database requirements:

- Consistency

  The data a session is viewing or changing must not be changed by other sessions until the user is finished.

- Integrity

  The data and structures must reflect all changes made to them in the correct sequence.

Oracle Database provides data concurrency, consistency, and integrity among transactions through its locking mechanisms. Locking occurs automatically and requires no user action.

The need for locks can be illustrated by a concurrent update of a single row. In the following example, a simple web-based application presents the end user with an employee email and phone number. The application uses an `UPDATE` statement such as the following to modify the data:

```
UPDATE employees
SET    email = ?, phone_number = ?
WHERE  employee_id = ?
AND    email = ?
AND    phone_number = ?
```

In the preceding `UPDATE` statement, the email and phone number values in the `WHERE` clause are the original, unmodified values for the specified employee. This update ensures that the row that the application modifies was not changed after the application last read and displayed it to the user. In this way, the application avoids the lost update problem in which one user overwrites changes made by another user, effectively losing the update by the second user (Table 9-2 shows an example of a lost update).

Table 9-4 shows the sequence of events when two sessions attempt to modify the same row in the `employees` table at roughly the same time.

Table 9-4 Row Locking Example

T	Session 1	Session 2	Description

t0	`SELECT employee_id as ID,`
	`email, phone_number`
	`FROM hr.employees`
	`WHERE last_name='Himuro';`
	
	`ID EMAIL PHONE_NUMBER`
	`--- ------- ------------`
	`118 GHIMURO 515.127.4565`
			In session 1, the `hr1` user queries `hr.employees` for the Himuro record and displays the employee_id (`118`), email (`GHIMURO`), and phone number (`515.127.4565`) attributes.

t1		`SELECT employee_id as ID,`
		`email, phone_number`
		`FROM hr.employees`
		`WHERE last_name='Himuro';`
		
		`ID EMAIL PHONE_NUMBER`
		`--- ------- ------------`
		`118 GHIMURO 515.127.4565`
			In session 2, the `hr2` user queries `hr.employees` for the Himuro record and displays the employee_id (`118`), email (`GHIMURO`), and phone number (`515.127.4565`) attributes.

t2	`UPDATE hr.employees SET`
	`phone_number='515.555.1234'`
	`WHERE employee_id=118`
	`AND email='GHIMURO'`
	`AND phone_number =`
	`'515.127.4565';`
	
	`1 row updated.`
			In session 1, the `hr1` user updates the phone number in the row to `515.555.1234`, which acquires a lock on the `GHIMURO` row.

t3		`UPDATE hr.employees SET`
		`phone_number='515.555.1235'`
		`WHERE employee_id=118`
		`AND email='GHIMURO'`
		`AND phone_number =`
		`'515.127.4565';`
		
		`-- SQL*Plus does not show`
		`-- a row updated message or`
		`-- return the prompt.`
			In session 2, the `hr2` user attempts to update the same row, but is blocked because `hr1` is currently processing the row.
			The attempted update by `hr2` occurs almost simultaneously with the `hr1` update.

t4	`COMMIT;`
	
	`Commit complete.`
			In session 1, the `hr1` user commits the transaction.
			The commit makes the change for Himuro permanent and unblocks session 2, which has been waiting.

t5		`0 rows updated.`	In session 2, the `hr2` user discovers that the `GHIMURO` row was modified in such a way that it no longer matches its predicate. Because the predicates do not match, session 2 updates no records.

Table 9-4 (Cont.) Row Locking Example

T	Session 1	Session 2	Description

t6	`UPDATE hr.employees SET phone_number='515.555.1235' WHERE employee_id=118 AND email='GHIMURO' AND phone_number='515.555.1234';`
	
	`1 row updated.`
			In session 1, the `hr1` user realizes that it updated the `GHIMURO` row with the wrong phone number. The user starts a new transaction and updates the phone number in the row to `515.555.1235`, which locks the `GHIMURO` row.

t7		`SELECT employee_id as ID, email, phone_number FROM hr.employees WHERE last_name='Himuro';`
		
		`ID EMAIL PHONE_NUMBER`
		`--- ------- ------------`
		`118 GHIMURO 515.555.1234`
			In session 2, the `hr2` user queries `hr.employees` for the Himuro record. The record shows the phone number update committed by session 1 at t4. Oracle Database read consistency ensures that session 2 does not see the uncommitted change made at t6.

t8		`UPDATE hr.employees SET phone_number='515.555.1235 WHERE employee_id=118 AND email='GHIMURO' AND phone_number = '515.555.1234';`
		
		`-- SQL*Plus does not show -- a row updated message or -- return the prompt.`
			In session 2, the `hr2` user attempts to update the same row, but is blocked because `hr1` is currently processing the row.

t9	`ROLLBACK;`
	
	`Rollback complete.`
			In session 1, the `hr1` user rolls back the transaction, which ends it.

t10		`1 row updated.`	In session 2, the update of the phone number succeeds because the session 1 update was rolled back. The `GHIMURO` row matches its predicate, so the update succeeds.

t11		`COMMIT;`
		
		`Commit complete.`
			Session 2 commits the update, ending the transaction.

Oracle Database automatically obtains necessary locks when executing SQL statements. For example, before the database permits a session to modify data, the session must first lock the data. The lock gives the session exclusive control over the data so that no other transaction can modify the locked data until the lock is released.

Because the locking mechanisms of Oracle Database are tied closely to transaction control, application designers need only define transactions properly, and Oracle Database automatically manages locking. Users never need to lock any resource explicitly, although Oracle Database also enables users to lock data manually.

The following sections explain concepts that are important for understanding how Oracle Database achieves data concurrency.

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* to learn about the `OWA_OPT_LOCK` package, which contains subprograms that can help prevent lost updates

Lock Modes

Oracle Database automatically uses the lowest applicable level of restrictiveness to provide the highest degree of data concurrency yet also provide fail-safe data integrity.

The less restrictive the level, the more available the data is for access by other users. Conversely, the more restrictive the level, the more limited other transactions are in the types of locks that they can acquire.

Oracle Database uses two modes of locking in a multiuser database:

- Exclusive lock mode

  This mode prevents the associated resource from being shared. A transaction obtains an exclusive lock when it modifies 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

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

Assume that a transaction uses a `SELECT ... FOR UPDATE` statement to select a single table row. The transaction acquires an exclusive row lock and a row share table lock. The row lock allows other sessions to modify any rows *other than* the locked row, while the table lock prevents sessions from altering the structure of the table. Thus, the database permits as many statements as possible to execute.

Lock Conversion and Escalation

Oracle Database performs **lock conversion** as necessary.

In lock conversion, the database automatically converts a table lock of lower restrictiveness to one of higher restrictiveness. For example, suppose a transaction issues a `SELECT ... FOR UPDATE` for an employee and later updates the locked row. In this case, the database automatically converts the row share table lock to a row exclusive table lock. 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.

Lock conversion is different from lock escalation, which 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). If a session locks many rows in a

table, then some databases automatically escalate the row locks to a single table. The number of locks decreases, but the restrictiveness of what is locked increases.

Oracle Database never escalates locks. Lock escalation greatly increases the probability of deadlocks. Assume that a system is trying to escalate locks on behalf of transaction 1 but cannot because of the locks held by transaction 2. A deadlock is created if transaction 2 also requires lock escalation of the same data before it can proceed.

Lock Duration

Oracle Database automatically releases a lock when some event occurs so that the transaction no longer requires the resource.

Usually, the database holds locks acquired by statements within a transaction for the duration of the transaction. These locks prevent destructive interference such as dirty reads, lost updates, and destructive DDL from concurrent transactions.

> **Note:**
>
> A table lock taken on a child table because of an unindexed foreign key is held for the duration of the statement, not the transaction. Also, the `DBMS_LOCK` package enables user-defined locks to be released and allocated at will and even held over transaction boundaries.

Oracle Database releases all locks acquired by the statements within a transaction when it commits or rolls back. Oracle Database 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 continue to wait until after the original transaction commits or rolls back completely.

> **See Also:**
>
> - "Table 10-3" that shows transaction waiting behavior
> - "Overview of User-Defined Locks" to learn more about `DBMS_LOCK`

Locks and Deadlocks

A **deadlock** is a situation in which two or more users are waiting for data locked by each other. Deadlocks prevent some transactions from continuing to work.

Oracle Database automatically detects deadlocks and resolves them by rolling back one statement involved in the deadlock, releasing one set of the conflicting row locks. The database returns a corresponding message to the transaction that undergoes statement-level rollback. The statement rolled back belongs to the transaction that detects the deadlock. Usually, the signaled transaction should be rolled back explicitly, but it can retry the rolled-back statement after waiting.

Table 9-5 illustrates two transactions in a deadlock.

Table 9-5 Deadlocked Transactions

T	Session 1	Session 2	Explanation

t0	`SQL> UPDATE employees`
	`  SET salary = salary*1.1`
	`  WHERE employee_id = 100;`
	
	`1 row updated.`
		`SQL> UPDATE employees`
		`  SET  salary = salary*1.1`
		`  WHERE employee_id = 200;`
		
		`1 row updated.`
			Session 1 starts transaction 1 and updates the salary for employee 100. Session 2 starts transaction 2 and updates the salary for employee 200. No problem exists because each transaction locks only the row that it attempts to update.

t1	`SQL> UPDATE employees`
	`  SET salary = salary*1.1`
	`  WHERE employee_id = 200;`
	
	`-- prompt does not return`
		`SQL> UPDATE employees`
		`  SET salary = salary*1.1`
		`  WHERE employee_id = 100;`
		
		`-- prompt does not return`
			Transaction 1 attempts to update the employee 200 row, which is currently locked by transaction 2. Transaction 2 attempts to update the employee 100 row, which is currently locked by transaction 1.
			A deadlock results because neither transaction can obtain the resource it needs to proceed or terminate. No matter how long each transaction waits, the conflicting locks are held.

t2	`UPDATE employees`
	`       *`
	`ERROR at line 1:`
	`ORA-00060: deadlock detected`
	`while waiting for resource`
	
	`SQL>`
			Transaction 1 signals the deadlock and rolls back the `UPDATE` statement issued at t1. However, the update made at t0 is not rolled back. The prompt is returned in session 1.
			Note: Only one session in the deadlock actually gets the deadlock error, but either session could get the error.

t3	`SQL> COMMIT;`
	
	`Commit complete.`
			Session 1 commits the update made at t0, ending transaction 1. The update unsuccessfully attempted at t1 is not committed.

t4		`1 row updated.`
		
		`SQL>`
			The update at t1 in transaction 2, which was being blocked by transaction 1, is executed. The prompt is returned.

t5		`SQL> COMMIT;`
		
		`Commit complete.`
			Session 2 commits the updates made at t0 and t1, which ends transaction 2.

Deadlocks most often occur when transactions explicitly override the default locking of Oracle Database. Because Oracle Database does not escalate locks and does not use read locks for queries, but does use row-level (rather than page-level) locking, deadlocks occur infrequently.

> **See Also:**
>
> - "Overview of Manual Data Locks"
> - *Oracle Database Development Guide* to learn how to handle deadlocks when you lock tables explicitly

Overview of Automatic Locks

Oracle Database automatically locks a resource on behalf of a transaction to prevent other transactions from doing something that requires exclusive access to the same resource.

The database automatically acquires different types of locks at different levels of restrictiveness depending on the resource and the operation being performed.

> **Note:**
>
> The database never locks rows when performing simple reads.

Oracle Database locks are divided into the categories show in the following table.

Table 9-6 Lock Categories

Lock	Description	To Learn More
DML Locks	Protect data. For example, table locks lock entire tables, while row locks lock selected rows.	"DML Locks"
DDL Locks	Protect the structure of schema objects—for example, the dictionary definitions of tables and views.	"DDL Locks"
System Locks	Protect internal database structures such as data files. Latches, mutexes, and internal locks are entirely automatic.	"System Locks"

DML Locks

A **DML lock**, also called a *data lock*, guarantees the integrity of data accessed concurrently by multiple users.

For example, a DML lock prevents two customers from buying the last copy of a book available from an online bookseller. DML locks prevent destructive interference of simultaneous conflicting DML or DDL operations.

DML statements automatically acquire the following types of locks:

- Row Locks (TX)
- Table Locks (TM)

In the following sections, the acronym in parentheses after each type of lock or lock mode is the abbreviation used in the Locks Monitor of Oracle Enterprise Manager (Enterprise Manager). Enterprise Manager might display TM for any table lock, rather than indicate the mode of table lock (such as RS or SRX).

> **See Also:**
>
> "Oracle Enterprise Manager"

Row Locks (TX)

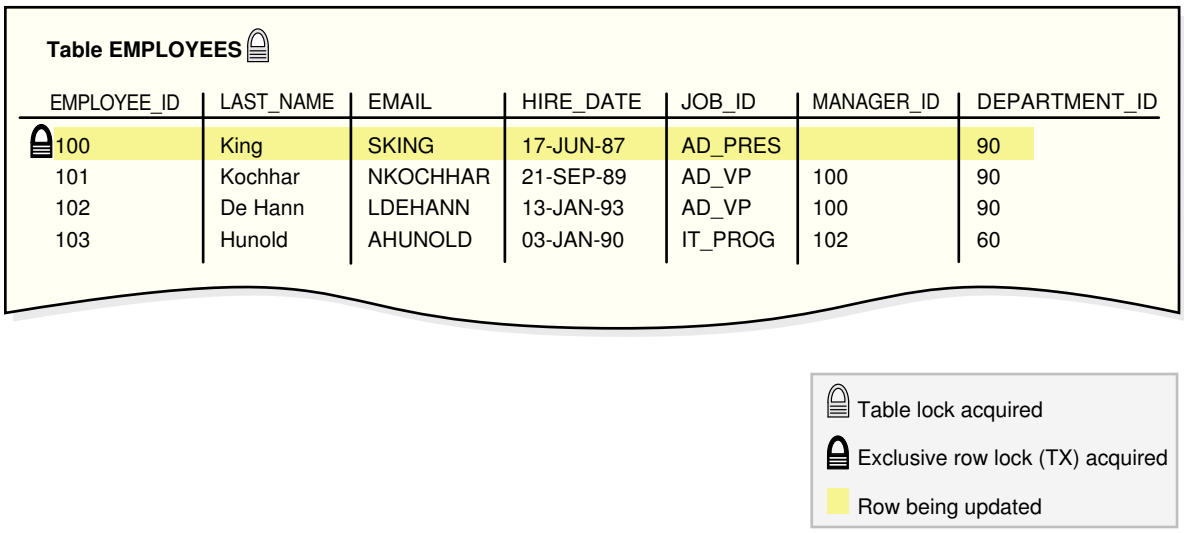
A **row lock**, also called a *TX lock*, is a lock on a single row of table. A transaction acquires a row lock for each row modified by an `INSERT`, `UPDATE`, `DELETE`, `MERGE`, or `SELECT ... FOR UPDATE` statement. The row lock exists until the transaction commits or rolls back.

Row locks primarily serve as a queuing mechanism to prevent two transactions from modifying the same row. The database always locks a modified row in exclusive mode so that other transactions cannot modify the row until the transaction holding the lock commits or rolls back. Row locking provides the finest grain locking possible and so provides the best possible concurrency and throughput.

> **Note:**
>
> If a transaction terminates because of database instance failure, then block-level recovery makes a row available before the entire transaction is recovered.

If a transaction obtains a lock for a row, then the transaction also acquires a lock for the table containing the row. The table lock prevents conflicting DDL operations that would override data changes in a current transaction. Figure 9-2 illustrates an update of the third row in a table. Oracle Database automatically places an exclusive lock on the updated row and a subexclusive lock on the table.

Figure 9-2 Row and Table Locks



Row Locks and Concurrency

This scenario illustrates how Oracle Database uses row locks for concurrency.

Three sessions query the same rows simultaneously. Session 1 and 2 proceed to make uncommitted updates to different rows, while session 3 makes no updates. Each session sees its own uncommitted updates but not the uncommitted updates of any other session.

Table 9-7 Data Concurrency Example

T	Session 1	Session 2	Session 3	Explanation

t0	`SELECT employee_id, salary FROM employees WHERE employee_id IN ( 100, 101 );`
	
	`EMPLOYEE_ID SALARY`
	`----------- ------`
	`100 512`
	`101 600`
		`SELECT employee_id, salary FROM employees WHERE employee_id IN ( 100, 101 );`
		
		`EMPLOYEE_ID SALARY`
		`----------- ------`
		`100 512`
		`101 600`
			`SELECT employee_id, salary FROM employees WHERE employee_id IN ( 100, 101 );`
			
			`EMPLOYEE_ID SALARY`
			`----------- ------`
			`100 512`
			`101 600`
				Three different sessions simultaneously query the ID and salary of employees 100 and 101. The results returned by each query are identical.

t1	`UPDATE hr.employees SET salary = salary+100 WHERE employee_id=100;`			Session 1 updates the salary of employee 100, but does not commit. In the update, the writer acquires a row-level lock for the updated row only, thereby preventing other writers from modifying this row.

Table 9-7 (Cont.) Data Concurrency Example

T	Session 1	Session 2	Session 3	Explanation

t2	`SELECT employee_id,`
	`  salary`
	`FROM  employees`
	`WHERE  employee_id`
	`IN ( 100, 101 );`
	
	`EMPLOYEE_ID  SALARY`
	`-----------  ------`
	`100  **612**`
	`101  600`
		`SELECT employee_id,`
		`  salary`
		`FROM  employees`
		`WHERE  employee_id`
		`IN ( 100, 101 );`
		
		`EMPLOYEE_ID  SALARY`
		`-----------  ------`
		`100  512`
		`101  600`
			`SELECT employee_id,`
			`  salary`
			`FROM  employees`
			`WHERE  employee_id`
			`IN ( 100, 101 );`
			
			`EMPLOYEE_ID  SALARY`
			`-----------  ------`
			`100  512`
			`101  600`
				Each session simultaneously issues the original query. Session 1 shows the salary of 612 resulting from the t1 update. The readers in session 2 and 3 return rows immediately and do not wait for session 1 to end its transaction. The database uses multiversion read consistency to show the salary as it existed before the update in session 1.

t3		`UPDATE hr.employee`
		`  SET  salary =`
		`  salary+100`
		`  WHERE`
		`  employee_id=101;`
				Session 2 updates the salary of employee 101, but does not commit the transaction. In the update, the writer acquires a row-level lock for the updated row only, preventing other writers from modifying this row.

t4	`SELECT employee_id,`
	`  salary`
	`FROM  employees`
	`WHERE  employee_id`
	`IN ( 100, 101 );`
	
	`EMPLOYEE_ID  SALARY`
	`-----------  ------`
	`100  **612**`
	`101  600`
		`SELECT employee_id,`
		`  salary`
		`FROM  employees`
		`WHERE  employee_id`
		`IN ( 100, 101 );`
		
		`EMPLOYEE_ID  SALARY`
		`-----------  ------`
		`100  512`
		`101  **700**`
			`SELECT employee_id,`
			`  salary`
			`FROM  employees`
			`WHERE  employee_id`
			`IN ( 100, 101 );`
			
			`EMPLOYEE_ID  SALARY`
			`-----------  ------`
			`100  512`
			`101  600`
				Each session simultaneously issues the original query. Session 1 shows the salary of 612 resulting from the t1 update, but not the salary update for employee 101 made in session 2. The reader in session 2 shows the salary update made in session 2, but not the salary update made in session 1. The reader in session 3 uses read consistency to show the salaries before modification by session 1 and 2.

> **See Also:**
>
> - *Oracle Database SQL Language Reference*
> - *Oracle Database Reference* to learn about `V$LOCK`

Storage of Row Locks

Unlike some databases, which use a lock manager to maintain a list of locks in memory, Oracle Database stores lock information in the data block that contains the locked row.

The database uses a queuing mechanism for acquisition of row locks. If a transaction requires a lock for an unlocked row, then the transaction places a lock in the data block. Each row modified by this transaction points to a copy of the transaction ID stored in the block header.

When a transaction ends, the transaction ID remains in the block header. If a different transaction wants to modify a row, then it uses the transaction ID to determine whether the lock is active. If the lock is active, then the session asks to be notified when the lock is released. Otherwise, the transaction acquires the lock.

> **See Also:**
>
> - "Overview of Data Blocks" to learn more about the data block header
> - *Oracle Database Reference* to learn about `V$TRANSACTION`

Table Locks (TM)

A **table lock**, also called a *TM lock*, is acquired by a transaction when a table is modified by an `INSERT`, `UPDATE`, `DELETE`, `MERGE`, `SELECT` with the `FOR UPDATE` clause, or `LOCK TABLE` statement.

DML operations require table locks to reserve DML access to the table on behalf of a transaction and to prevent DDL operations that would conflict with the transaction.

A table lock can be held in any of the following modes:

- Row Share (RS)

  This lock, also 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 lock is the least restrictive mode of table lock, offering the highest degree of concurrency for a table.

- Row Exclusive Table Lock (RX)

  This lock, also called a *subexclusive table lock (SX)*, generally indicates that the transaction holding the lock has updated table rows or issued `SELECT ... FOR UPDATE`. An SX lock allows other transactions to query, insert, update, delete, or lock rows concurrently in the same table. Therefore, SX locks allow multiple

transactions to obtain simultaneous SX and subshare table locks for the same table.

- Share Table Lock (S)

  A share table lock held by a transaction allows other transactions to query the table (without using `SELECT ... FOR UPDATE`), but updates are allowed only if a single transaction holds the share table lock. Because multiple transactions may hold a share table lock concurrently, holding this lock is not sufficient to ensure that a transaction can modify the table.

- Share Row Exclusive Table Lock (SRX)

  This lock, also called a *share-subexclusive table lock (SSX)*, is more restrictive than a share table lock. Only one transaction at a time can acquire an SSX lock on a given table. An SSX lock held by a transaction allows other transactions to query the table (except for `SELECT ... FOR UPDATE`) but not to update the table.

- Exclusive Table Lock (X)

  This lock is the most restrictive, prohibiting other transactions from performing any type of DML statement or placing any type of lock on the table.

See Also:

- *Oracle Database SQL Language Reference*
- *Oracle Database Development Guide* to learn more about table locks

Locks and Foreign Keys

Oracle Database maximizes the concurrency control of parent keys in relation to dependent foreign keys.

Locking behavior depends on whether foreign key columns are indexed. If foreign keys are not indexed, then the child table will probably be locked more frequently, deadlocks will occur, and concurrency will be decreased. For this reason foreign keys should almost always be indexed. The only exception is when the matching unique or primary key is never updated or deleted.

Locks and Unindexed Foreign Keys

The database acquires a full table lock on the child table when no index exists on the foreign key column of the child table, and a session modifies a primary key in the parent table (for example, deletes a row or modifies primary key attributes) or merges rows into the parent table.

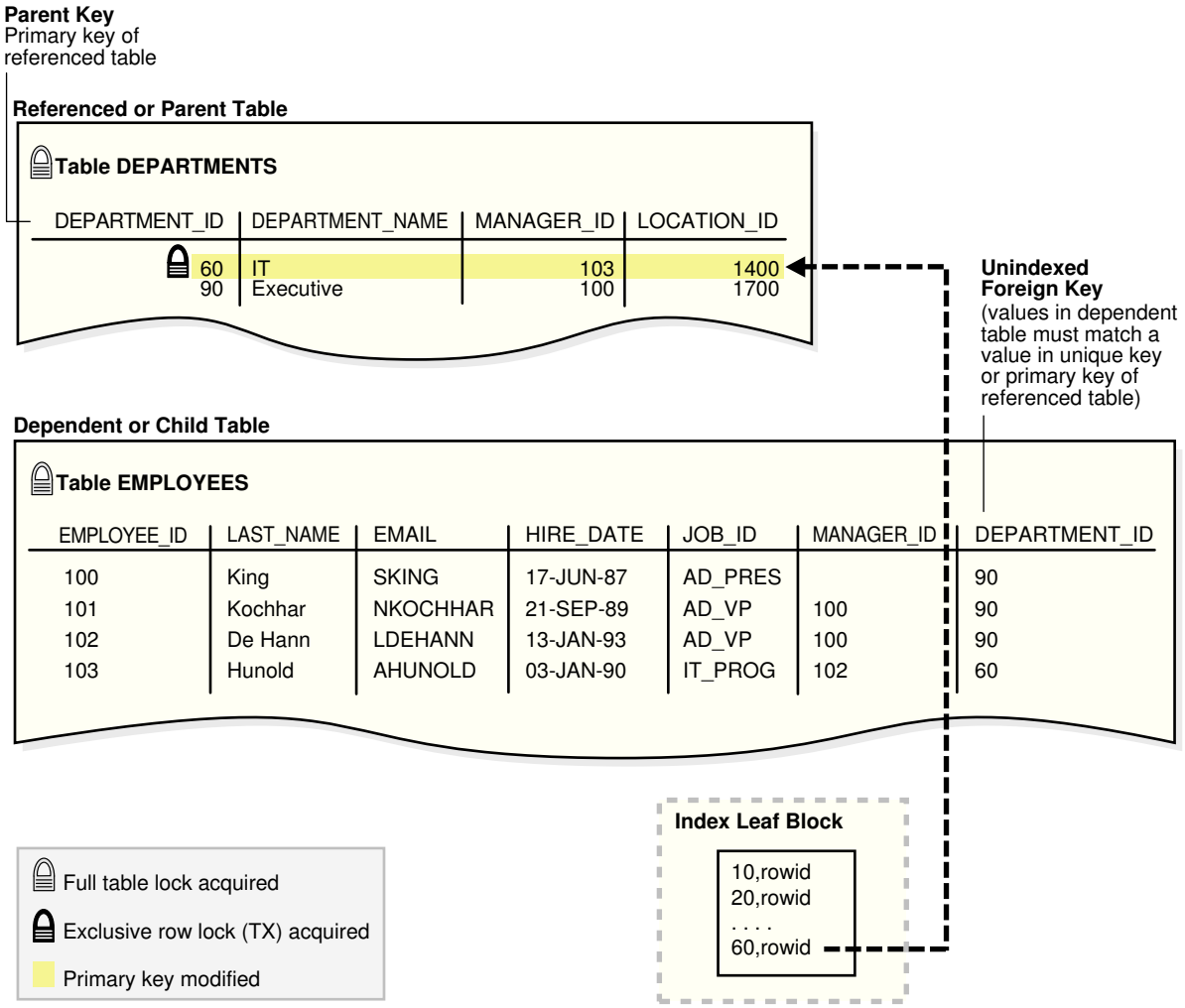
When both of the following conditions are true, the database acquires a full table lock on the child table:

- No index exists on the foreign key column of the child table.
- A session modifies a primary key in the parent table (for example, deletes a row or modifies primary key attributes) or merges rows into the parent table.

> **Note:**
>
> Inserts into the parent table do *not* acquire blocking table locks that prevent DML on the child table. In the case of inserts, the database acquires a lock on the child table that prevents structural changes, but not modifications of existing or newly added rows.

Suppose that `hr.departments` table is a parent of `hr.employees`, which contains the unindexed foreign key `employees.department_id`. The following figure shows a session modifying the primary key attributes of department 60 in the `departments` table.

Figure 9-3 Locking Mechanisms with Unindexed Foreign Key



In Figure 9-3, the database acquires a full table lock on `employees` during the primary key modification of department 60. This lock enables other sessions to query but not update the `employees` table. For example, sessions cannot update employee phone numbers. The table lock on `employees` releases immediately after the primary key modification on the `departments` table completes. If multiple rows in `departments`

undergo primary key modifications, then a table lock on `employees` is obtained and released once for each row that is modified in `departments`.

> **Note:**
>
> DML on a child table does not acquire a table lock on the parent table.

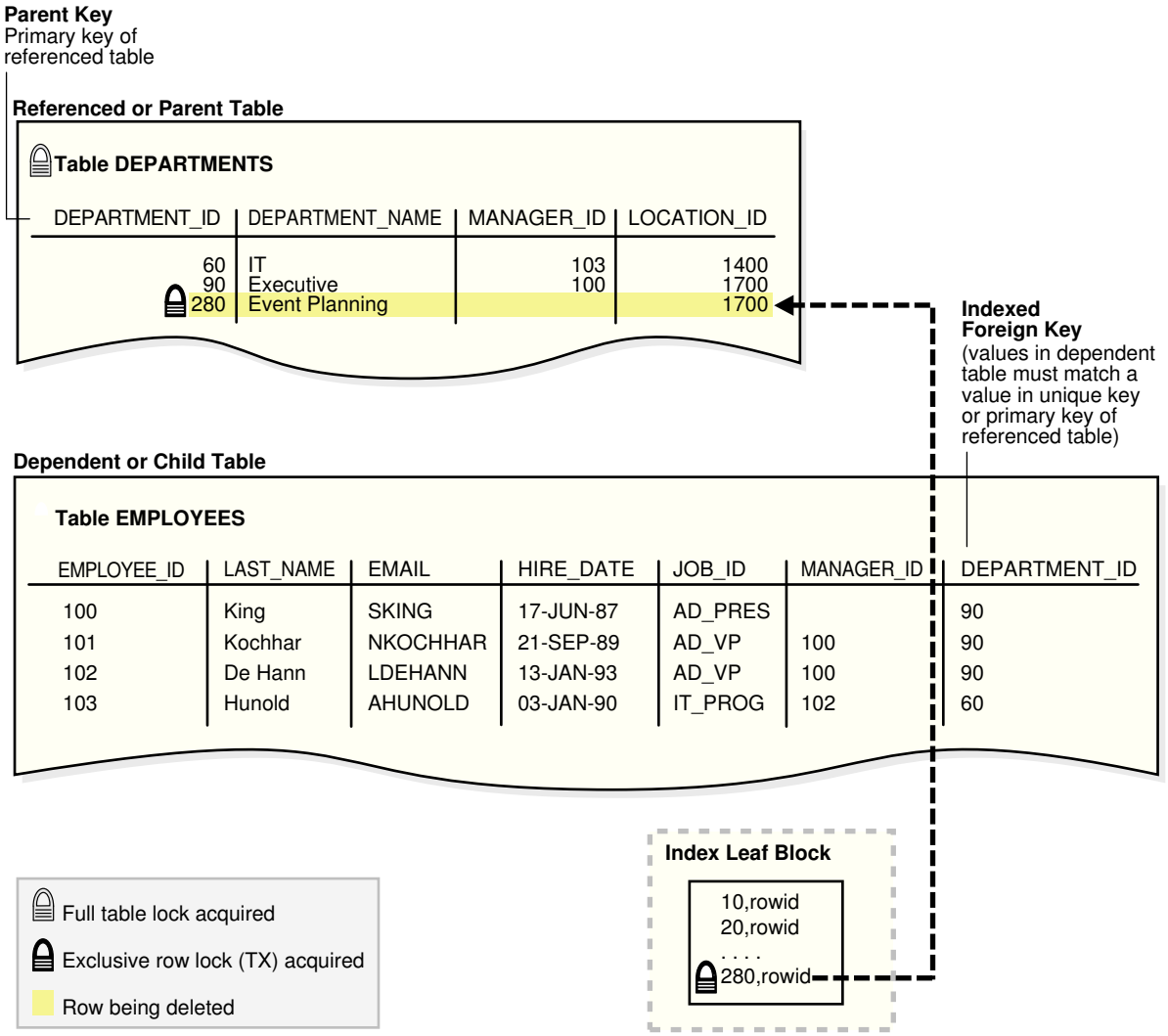
Locks and Indexed Foreign Keys

The database does *not* acquire a full table lock on the child table when a foreign key column in the child table *is* indexed, and a session modifies a primary key in the parent table (for example, deletes a row or modifies primary key attributes) or merges rows into the parent table.

A lock on the parent table prevents transactions from acquiring exclusive table locks, but does not prevent DML on the parent *or* child table during the primary key modification. This situation is preferable if primary key modifications occur on the parent table while updates occur on the child table.

Figure 9-4 shows child table `employees` with an indexed `department_id` column. A transaction deletes department 280 from `departments`. This deletion does not cause the database to acquire a full table lock on the `employees` table as in the scenario described in "Locks and Unindexed Foreign Keys".

Figure 9-4 Locking Mechanisms with Indexed Foreign Key



If the child table specifies `ON DELETE CASCADE`, then deletions from the parent table can result in deletions from the child table. For example, the deletion of department 280 can cause the deletion of records from `employees` for employees in the deleted department. In this case, waiting and locking rules are the same as if you deleted rows from the child table after deleting rows from the parent table.

> **See Also:**
>
> - "Foreign Key Constraints"
> - "Introduction to Indexes"

DDL Locks

A **data dictionary (DDL) lock** protects the definition of a schema object while an ongoing DDL operation acts on or refers to the object.

Only individual schema objects that are modified or referenced are locked during DDL operations. The database never locks the whole data dictionary.

Oracle Database acquires a DDL lock automatically on behalf of any DDL transaction requiring it. Users cannot explicitly request DDL locks. For example, if a user creates a stored procedure, then Oracle Database automatically acquires DDL locks for all schema objects referenced in the procedure definition. The DDL locks prevent these objects from being altered or dropped before procedure compilation is complete.

Exclusive DDL Locks

An exclusive DDL lock prevents other sessions from obtaining a DDL or DML lock.

Most DDL operations 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, `DROP TABLE` is not allowed to drop a table while `ALTER TABLE` is adding a column to it, and vice versa.

Exclusive DDL locks last for the duration of DDL statement execution and automatic commit. During the acquisition of an exclusive DDL lock, if another DDL lock is held on the schema object by another operation, then the acquisition waits until the older DDL lock is released and then proceeds.

> **See Also:**
>
> "Share DDL Locks" describes situations where exclusive locks are not required to prevent destructive interference

Share DDL Locks

A share DDL lock for a resource prevents destructive interference with conflicting DDL operations, but allows data concurrency for similar DDL operations.

For example, when a `CREATE PROCEDURE` statement is run, the containing transaction acquires share DDL locks for all referenced tables. Other transactions can concurrently create procedures that reference the same tables and acquire concurrent share DDL locks on the same tables, but no transaction can acquire an exclusive DDL lock on any referenced table.

A share DDL lock lasts for the duration of DDL statement execution and automatic commit. Thus, a transaction holding a share DDL lock is guaranteed that the definition of the referenced schema object remains constant during the transaction.

Breakable Parse Locks

A **parse lock** is held by a SQL statement or PL/SQL program unit for each schema object that 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 is called a *breakable parse lock* because it does not disallow any DDL operation and can be broken to allow conflicting DDL operations.

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

> **See Also:**
>
> "Shared Pool"

System Locks

Oracle Database uses various types of system locks to protect internal database and memory structures. These mechanisms are inaccessible to users because users have no control over their occurrence or duration.

Latches

A **latch** is a simple, low-level serialization mechanism that coordinates multiuser access to shared data structures, objects, and files.

Latches protect shared memory resources from corruption when accessed by multiple processes. Specifically, latches protect data structures from the following situations:

- Concurrent modification by multiple sessions
- Being read by one session while being modified by another session
- Deallocation (aging out) of memory while being accessed

Typically, a single latch protects multiple objects in the SGA. For example, background processes such as DBW and LGWR allocate memory from the shared pool to create data structures. To allocate this memory, these processes use a shared pool latch that serializes access to prevent two processes from trying to inspect or modify the shared pool simultaneously. After the memory is allocated, other processes may need to access shared pool areas such as the library cache, which is required for parsing. In this case, processes latch only the library cache, not the entire shared pool.

Unlike enqueue latches such as row locks, latches do not permit sessions to queue. When a latch becomes available, the first session to request the latch obtains exclusive access to it. The phenomenon of latch spinning occurs when a process repeatedly requests a latch in a loop, whereas latch sleeping occurs when a process releases the CPU before renewing the latch request.

Typically, an Oracle process acquires a latch for an extremely short time while manipulating or looking at a data structure. For example, while processing a salary update of a single employee, the database may obtain and release thousands of latches. The implementation of latches is operating system-dependent, especially in respect to whether and how long a process waits for a latch.

An increase in latching means a decrease in concurrency. For example, excessive hard parse operations create contention for the library cache latch. The `V$LATCH` view

contains detailed latch usage statistics for each latch, including the number of times each latch was requested and waited for.

> **See Also:**
>
> - "SQL Parsing"
> - *Oracle Database Reference* to learn about `V$LATCH`
> - *Oracle Database Performance Tuning Guide* to learn about wait event statistics

Mutexes

A **mutual exclusion object (mutex)** is a low-level mechanism that prevents an object in memory from aging out or from being corrupted when accessed by concurrent processes. A mutex is similar to a latch, but whereas a latch typically protects a group of objects, a mutex protects a single object.

Mutexes provide several benefits:

- A mutex can reduce the possibility of contention.

  Because a latch protects multiple objects, it can become a bottleneck when processes attempt to access any of these objects concurrently. By serializing access to an individual object rather than a group, a mutex increases availability.
- A mutex consumes less memory than a latch.
- When in shared mode, a mutex permits concurrent reference by multiple sessions.

Internal Locks

Internal locks are higher-level, more complex mechanisms than latches and mutexes and serve various purposes.

The database uses the following types of internal locks:

- 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, whereas exclusive locks are released when the DDL operation is complete.
- File and log management locks

  These locks protect various files. For example, an internal 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 online redo log files. Data files 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 undo segment locks

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

> **See Also:**
>
> "Data Dictionary Cache"

Overview of Manual Data Locks

You can manually override the Oracle Database default locking mechanisms.

Oracle Database performs locking automatically to ensure data concurrency, data integrity, and statement-level read consistency. However, overriding the default locking is useful in situations such as the following:

- Applications require transaction-level read consistency or repeatable reads.

  In this case, queries 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.

You can override Oracle Database automatic locking at the session or transaction level. At the session level, a session can set the required transaction isolation level with the `ALTER SESSION` statement. At the transaction level, transactions that include the following SQL statements override Oracle Database default locking:

- The `SET TRANSACTION ISOLATION LEVEL` statement
- The `LOCK TABLE` statement (which locks either a table or, when used with views, the base tables)
- The `SELECT ... FOR UPDATE` statement

Locks acquired by the preceding statements are released after the transaction ends or a rollback to savepoint releases them.

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

> **See Also:**
>
> - *Oracle Database SQL Language Reference* for descriptions of `LOCK TABLE` and `SELECT`
> - *Oracle Database Development Guide* to learn how to manually lock tables

Overview of User-Defined Locks

With Oracle Database Lock Management services, you can define your own locks for a specific application.

For example, you might create a lock to serialize access to a message log on the file system. Because a reserved user lock is the same as an Oracle Database lock, it has all the Oracle Database lock functionality including deadlock detection. User locks never conflict with Oracle Database locks, because they are identified with the prefix `UL`.

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

> **See Also:**
>
> - *Oracle Database Development Guide* for more information about Oracle Database Lock Management services
> - *Oracle Database PL/SQL Packages and Types Reference* for information about `DBMS_LOCK`

10
Transactions

This chapter defines a transaction and describes how the database processes transactions.

This chapter contains the following sections:

Introduction to Transactions

A **transaction** is a logical, atomic unit of work that contains one or more SQL statements.

A transaction groups SQL statements so that they are either all committed, which means they are applied to the database, or all rolled back, which means they are undone from the database. Oracle Database assigns every transaction a unique identifier called a transaction ID.

All Oracle transactions obey the basic properties of a database transaction, known as ACID properties. ACID is an acronym for the following:

- Atomicity

  All tasks of a transaction are performed or none of them are. There are no partial transactions. For example, if a transaction starts updating 100 rows, but the system fails after 20 updates, then the database rolls back the changes to these 20 rows.

- Consistency

  The transaction takes the database from one consistent state to another consistent state. For example, in a banking transaction that debits a savings account and credits a checking account, a failure must not cause the database to credit only one account, which would lead to inconsistent data.

- Isolation

  The effect of a transaction is not visible to other transactions until the transaction is committed. For example, one user updating the `hr.employees` table does not see the uncommitted changes to `employees` made concurrently by another user. Thus, it appears to users as if transactions are executing serially.

- Durability

Changes made by committed transactions are permanent. After a transaction completes, the database ensures through its recovery mechanisms that changes from the transaction are not lost.

The use of transactions is one of the most important ways that a database management system differs from a file system.

Sample Transaction: Account Debit and Credit

To illustrate the concept of a transaction, consider a banking database.

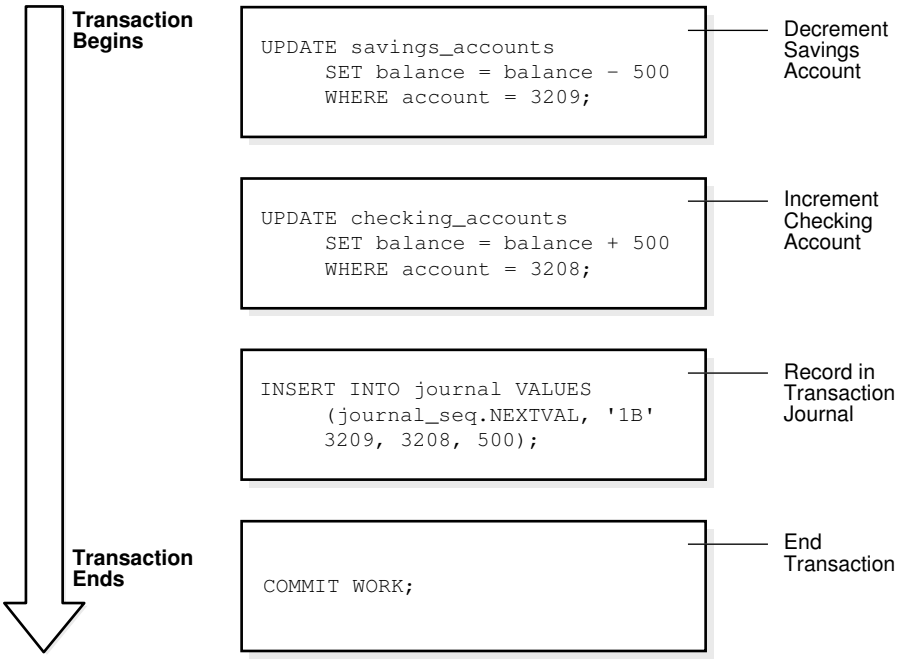
When a customer transfers money from a savings account to a checking account, the transaction must consist of three separate operations:

- Decrement the savings account
- Increment the checking account
- Record the transaction in the transaction journal

Oracle Database must allow for two situations. If all three SQL statements maintain the accounts in proper balance, then the effects of the transaction can be applied to the database. However, if a problem such as insufficient funds, invalid account number, or a hardware failure prevents one or two of the statements in the transaction from completing, then the database must roll back the entire transaction so that the balance of all accounts is correct.

The following graphic illustrates a banking transaction. The first statement subtracts $500 from savings account 3209. The second statement adds $500 to checking account 3208. The third statement inserts a record of the transfer into the journal table. The final statement commits the transaction.

Figure 10-1 A Banking Transaction



Structure of a Transaction

A database transaction consists of one or more statements.

Specifically, a transaction consists of one of the following:

- One or more data manipulation language (DML) statements that together constitute an atomic change to the database
- One data definition language (DDL) statement

A transaction has a beginning and an end.

> **See Also:**
>
> - "Overview of SQL Statements"
> - *Oracle Database SQL Language Reference* for an account of the types of SQL statements

Beginning of a Transaction

A transaction begins when the first executable SQL statement is encountered.

An executable SQL statement is a SQL statement that generates calls to a database instance, including DML and DDL statements and the `SET TRANSACTION` statement.

When a transaction begins, Oracle Database assigns the transaction to an available undo data segment to record the undo entries for the new transaction. A transaction ID is not allocated until an undo segment and transaction table slot are allocated, which occurs during the first DML statement. A transaction ID is unique to a transaction and represents the undo segment number, slot, and sequence number.

The following example execute an `UPDATE` statement to begin a transaction and queries `V$TRANSACTION` for details about the transaction:

```
SQL> UPDATE hr.employees SET salary=salary;

107 rows updated.

SQL> SELECT XID AS "txn id", XIDUSN AS "undo seg", XIDSLOT AS "slot",
  2  XIDSQN AS "seq", STATUS AS "txn status"
  3  FROM V$TRANSACTION;

txn id             undo seg       slot        seq txn status
---------------- ---------- ---------- ---------- ----------------
0600060037000000          6          6         55 ACTIVE
```

> **See Also:**
>
> "Undo Segments"

End of a Transaction

A transaction can end under different circumstances.

A transaction ends when any of the following actions occurs:

- A user issues a `COMMIT` or `ROLLBACK` statement *without* a `SAVEPOINT` clause.

  In a commit, a user explicitly or implicitly requested that the changes in the transaction be made permanent. Changes made by the transaction are permanent and visible to other users only after a transaction commits.

- A user runs a DDL command such as `CREATE`, `DROP`, `RENAME`, or `ALTER`.

  The database issues an implicit `COMMIT` statement before and after every DDL statement. If the current transaction contains DML statements, then Oracle Database first commits the transaction and then runs and commits the DDL statement as a new, single-statement transaction.

- A user exits normally from most Oracle Database utilities and tools, causing the current transaction to be implicitly committed. The commit behavior when a user disconnects is application-dependent and configurable.

  > **Note:**
  >
  > Applications should always explicitly commit or undo transactions before program termination.

- A client process terminates abnormally, causing the transaction to be implicitly rolled back using metadata stored in the transaction table and the undo segment.

After one transaction ends, the next executable SQL statement automatically starts the following transaction. The following example executes an `UPDATE` to start a transaction, ends the transaction with a `ROLLBACK` statement, and then executes an `UPDATE` to start a new transaction (note that the transaction IDs are different):

```
SQL> UPDATE hr.employees SET salary=salary;
107 rows updated.

SQL> SELECT XID, STATUS FROM V$TRANSACTION;

XID              STATUS
---------------- ----------------
0800090033000000 ACTIVE

SQL> ROLLBACK;

Rollback complete.

SQL> SELECT XID FROM V$TRANSACTION;

no rows selected

SQL> UPDATE hr.employees SET last_name=last_name;

107 rows updated.

SQL> SELECT XID, STATUS FROM V$TRANSACTION;
```

```
XID              STATUS
---------------- ----------------
0900050033000000 ACTIVE
```

> **See Also:**
>
> - "Sample Transaction: Account Debit and Credit" for an example of a transaction that ends with a commit.
> - *Oracle Database SQL Language Reference* to learn about `COMMIT`

Statement-Level Atomicity

Oracle Database supports **statement-level atomicity**, which means that a SQL statement is an atomic unit of work and either completely succeeds or completely fails.

A successful statement is different from a committed transaction. A single SQL statement executes successfully if the database parses and runs it without error as an atomic unit, as when all rows are changed in a multirow update.

If a SQL statement causes an error during execution, then it is not successful and so all effects of the statement are rolled back. This operation is a statement-level rollback. This operation has the following characteristics:

- A SQL statement that does not succeed causes the loss only of work it would have performed itself.

  The unsuccessful statement does not cause the loss of any work that preceded it in the current transaction. For example, if the execution of the second `UPDATE` statement in "Sample Transaction: Account Debit and Credit" causes an error and is rolled back, then the work performed by the first `UPDATE` statement is not rolled back. The first `UPDATE` statement can be committed or rolled back explicitly by the user.
- The effect of the rollback is as if the statement had never been run.

  Any side effects of an atomic statement, for example, triggers invoked upon execution of the statement, are considered part of the atomic statement. Either all work generated as part of the atomic statement succeeds or none does.

An example of an error causing a statement-level rollback is an attempt to insert a duplicate primary key. Single SQL statements involved in a deadlock, which is competition for the same data, can also cause a statement-level rollback. However, errors discovered during SQL statement parsing, such as a syntax error, have not yet been run and so do not cause a statement-level rollback.

> **See Also:**
>
> - "SQL Parsing"
> - "Locks and Deadlocks"
> - "Overview of Triggers"

System Change Numbers (SCNs)

A **system change number (SCN)** is a logical, internal time stamp used by Oracle Database.

SCNs order events that occur within the database, which is necessary to satisfy the ACID properties of a transaction. Oracle Database uses SCNs to mark the SCN before which all changes are known to be on disk so that recovery avoids applying unnecessary redo. The database also uses SCNs to mark the point at which no redo exists for a set of data so that recovery can stop.

SCNs occur in a monotonically increasing sequence. Oracle Database can use an SCN like a clock because an observed SCN indicates a logical point in time, and repeated observations return equal or greater values. If one event has a lower SCN than another event, then it occurred at an earlier time in the database. Several events may share the same SCN, which means that they occurred at the same time in the database.

Every transaction has an SCN. For example, if a transaction updates a row, then the database records the SCN at which this update occurred. Other modifications in this transaction have the same SCN. When a transaction commits, the database records an SCN for this commit.

Oracle Database increments SCNs in the system global area (SGA). When a transaction modifies data, the database writes a new SCN to the undo data segment assigned to the transaction. The log writer process then writes the commit record of the transaction immediately to the online redo log. The commit record has the unique SCN of the transaction. Oracle Database also uses SCNs as part of its instance recovery and media recovery mechanisms.

> **See Also:**
>
> - "Overview of Instance Recovery"
> - "Backup and Recovery"

Overview of Transaction Control

Transaction control is the management of changes made by DML statements and the grouping of DML statements into transactions.

In general, application designers are concerned with transaction control so that work is accomplished in logical units and data is kept consistent.

Transaction control involves using the following statements, as described in "Transaction Control Statements":

- The `COMMIT` statement ends the current transaction and makes all changes performed in the transaction permanent. `COMMIT` also erases all savepoints in the transaction and releases transaction locks.
- The `ROLLBACK` statement reverses the work done in the current transaction; it causes all data changes since the last `COMMIT` or `ROLLBACK` to be discarded. The

`ROLLBACK TO SAVEPOINT` statement undoes the changes since the last savepoint but does not end the entire transaction.

- The `SAVEPOINT` statement identifies a point in a transaction to which you can later roll back.

The session in Table 10-1 illustrates the basic concepts of transaction control.

Table 10-1 Transaction Control

T	Session	Explanation
t0	`COMMIT;`	This statement ends any existing transaction in the session.
t1	`SET TRANSACTION NAME 'sal_update';`	This statement begins a transaction and names it `sal_update`.

t2	`UPDATE employees`
	`  SET salary = 7000`
	`  WHERE last_name = 'Banda';`
		This statement updates the salary for Banda to 7000.

t3	`SAVEPOINT after_banda_sal;`	This statement creates a savepoint named `after_banda_sal`, enabling changes in this transaction to be rolled back to this point.

t4	`UPDATE employees`
	`  SET salary = 12000`
	`  WHERE last_name = 'Greene';`
		This statement updates the salary for Greene to 12000.

t5	`SAVEPOINT after_greene_sal;`	This statement creates a savepoint named `after_greene_sal`, enabling changes in this transaction to be rolled back to this point.
t6	`ROLLBACK TO SAVEPOINT` `  after_banda_sal;`	This statement rolls back the transaction to t3, undoing the update to Greene's salary at t4. The `sal_update` transaction has *not* ended.

t7	`UPDATE employees`
	`  SET salary = 11000`
	`  WHERE last_name = 'Greene';`
		This statement updates the salary for Greene to 11000 in transaction `sal_update`.

t8	`ROLLBACK;`	This statement rolls back all changes in transaction `sal_update`, ending the transaction.
t9	`SET TRANSACTION NAME 'sal_update2';`	This statement begins a new transaction in the session and names it `sal_update2`.

t10	`UPDATE employees`
	`  SET salary = 7050`
	`  WHERE last_name = 'Banda';`
		This statement updates the salary for Banda to 7050.

t11	`UPDATE employees`
	`  SET salary = 10950`
	`  WHERE last_name = 'Greene';`
		This statement updates the salary for Greene to 10950.

Table 10-1 (Cont.) Transaction Control

T	Session	Explanation
t12	`COMMIT;`	This statement commits all changes made in transaction `sal_update2`, ending the transaction. The commit guarantees that the changes are saved in the online redo log files.

> **See Also:**
>
> *Oracle Database SQL Language Reference* to learn about transaction control statements

Transaction Names

A transaction name is an optional, user-specified tag that serves as a reminder of the work that the transaction is performing. You name a transaction with the `SET TRANSACTION ... NAME` statement, which if used must be first statement of the transaction.

In Table 10-1, the first transaction was named `sal_update` and the second was named `sal_update2`.

Transaction names provide the following advantages:

- It is easier to monitor long-running transactions and to resolve in-doubt distributed transactions.
- You can view transaction names along with transaction IDs in applications. For example, a database administrator can view transaction names in Oracle Enterprise Manager (Enterprise Manager) when monitoring system activity.
- The database writes transaction names to the transaction auditing redo record, so you can use LogMiner to search for a specific transaction in the redo log.
- You can use transaction names to find a specific transaction in data dictionary views such as `V$TRANSACTION`.

> **See Also:**
>
> - "Oracle Enterprise Manager"
> - *Oracle Database Reference* to learn about `V$TRANSACTION`
> - *Oracle Database SQL Language Reference* to learn about `SET TRANSACTION`

Active Transactions

An **active transaction** is one that has started but not yet committed or rolled back.

In Table 10-1, the first statement to modify data in the `sal_update` transaction is the update to Banda's salary. From the successful execution of this update until the `ROLLBACK` statement ends the transaction, the `sal_update` transaction is active.

Data changes made by a transaction are temporary until the transaction is committed or rolled back. Before the transaction ends, the state of the data is as shown in the following table.

Table 10-2 State of the Data Before the Transaction Ends

State	Description	To Learn More
Oracle Database has generated undo information in the SGA.	The undo data contains the old data values changed by the SQL statements of the transaction.	"Read Consistency in the Read Committed Isolation Level"
Oracle Database has generated redo in the online redo log buffer of the SGA.	The redo log record contains the change to the data block and the change to the undo block.	"Redo Log Buffer "
Changes have been made to the database buffers of the SGA.	The data changes for a committed transaction, stored in the database buffers of the SGA, are not necessarily written immediately to the data files by the database writer (DBW). The disk write can happen before or after the commit.	"Database Buffer Cache"
The rows affected by the data change are locked.	Other users cannot change the data in the affected rows, nor can they see the uncommitted changes.	"Summary of Locking Behavior"

Savepoints

A **savepoint** is a user-declared intermediate marker within the context of a transaction.

Internally, the savepoint marker resolves to an SCN. Savepoints divide a long transaction into smaller parts.

If you use savepoints in a long transaction, then you have the option later of rolling back work performed before the current point in the transaction but after a declared savepoint within the transaction. Thus, if you make an error, you do not need to resubmit every statement. Table 10-1 creates savepoint `after_banda_sal` so that the update to the Greene salary can be rolled back to this savepoint.

Rollback to Savepoint

A rollback to a savepoint in an uncommitted transaction means undoing any changes made after the specified savepoint, but it does not mean a rollback of the transaction itself.

When a transaction is rolled back to a savepoint, as when the `ROLLBACK TO SAVEPOINT after_banda_sal` is run in Table 10-1, the following occurs:

1. Oracle Database rolls back only the statements run after the savepoint.

   In Table 10-1, the `ROLLBACK TO SAVEPOINT` causes the `UPDATE` for Greene to be rolled back, but not the `UPDATE` for Banda.

2. Oracle Database preserves the savepoint specified in the `ROLLBACK TO SAVEPOINT` statement, but all subsequent savepoints are lost.

   In Table 10-1, the `ROLLBACK TO SAVEPOINT` causes the `after_greene_sal` savepoint to be lost.

3. Oracle Database releases all table and row locks acquired after the specified savepoint but retains all data locks acquired before the savepoint.

The transaction remains active and can be continued.

See Also:

- *Oracle Database SQL Language Reference* to learn about the `ROLLBACK` and `SAVEPOINT` statements
- *Oracle Database PL/SQL Language Reference* to learn about transaction processing and control

Enqueued Transactions

Depending on the scenario, transactions waiting for previously locked resources may still be blocked after a rollback to savepoint.

When a transaction is blocked by another transaction it enqueues on the blocking transaction itself, so that the entire blocking transaction must commit or roll back for the blocked transaction to continue.

In the scenario shown in the following table, session 1 rolls back to a savepoint created before it executed a DML statement. However, session 2 is still blocked because it is waiting for the session 1 transaction to complete.

Table 10-3 Rollback to Savepoint Example

T	Session 1	Session 2	Session 3	Explanation
t0	`UPDATE employees SET salary=7000 WHERE last_name= 'Banda';`			Session 1 begins a transaction. The session places an exclusive lock on the `Banda` row (TX) and a subexclusive table lock (SX) on the table.
t1	`SAVEPOINT after_banda_sal;`			Session 1 creates a savepoint named `after_banda_sal`.
t2	`UPDATE employees SET salary=12000 WHERE last_name= 'Greene';`			Session 1 locks the `Greene` row.

Table 10-3 (Cont.) Rollback to Savepoint Example

T	Session 1	Session 2	Session 3	Explanation
t3		`UPDATE employees SET salary=14000 WHERE last_name= 'Greene';`		Session 2 attempts to update the `Greene` row, but fails to acquire a lock because session 1 has a lock on this row. No transaction has begun in session 2.
t4	`ROLLBACK TO SAVEPOINT after_banda_sal;`			Session 1 rolls back the update to the salary for `Greene`, which releases the row lock for `Greene`. The table lock acquired at t0 is not released. At this point, session 2 is *still* blocked by session 1 because session 2 enqueues on the session 1 *transaction*, which has not yet completed.
t5			`UPDATE employees SET salary=11000 WHERE last_name= 'Greene';`	The `Greene` row is currently unlocked, so session 3 acquires a lock for an update to the `Greene` row. This statement begins a transaction in session 3.
t6	`COMMIT;`			Session 1 commits, ending its transaction. Session 2 is now enqueued for its update to the `Greene` row behind the transaction in session 3.

> **See Also:**
>
> "Lock Duration" to learn more about when Oracle Database releases locks

Rollback of Transactions

A rollback of an uncommitted transaction undoes any changes to data that have been performed by SQL statements within the transaction.

After a transaction has been rolled back, the effects of the work done in the transaction no longer exist. In rolling back an entire transaction, without referencing any savepoints, Oracle Database performs the following actions:

- Undoes all changes made by all the SQL statements in the transaction by using the corresponding undo segments

  The transaction table entry for every active transaction contains a pointer to all the undo data (in reverse order of application) for the transaction. The database reads the data from the undo segment, reverses the operation, and then marks the undo entry as applied. Thus, if a transaction inserts a row, then a rollback deletes it. If a transaction updates a row, then a rollback reverses the update. If a transaction deletes a row, then a rollback reinserts it. In Table 10-1, the `ROLLBACK` reverses the updates to the salaries of Greene and Banda.
- Releases all the locks of data held by the transaction
- Erases all savepoints in the transaction

  In Table 10-1, the `ROLLBACK` deletes the savepoint `after_banda_sal`. The `after_greene_sal` savepoint was removed by the `ROLLBACK TO SAVEPOINT` statement.
- Ends the transaction

  In Table 10-1, the `ROLLBACK` leaves the database in the same state as it was after the initial `COMMIT` was executed.

The duration of a rollback is a function of the amount of data modified.

> **See Also:**
>
> - "Undo Segments"
> - *Oracle Database SQL Language Reference* to learn about the `ROLLBACK` command

Commits of Transactions

A commit ends the current transaction and makes permanent all changes performed in the transaction.

In Table 10-1, a second transaction begins with `sal_update2` and ends with an explicit `COMMIT` statement. The changes that resulted from the two `UPDATE` statements are now made permanent.

When a transaction commits, the following actions occur:

- The database generates an SCN for the `COMMIT`.

  The internal transaction table for the associated undo tablespace records that the transaction has committed. The corresponding unique SCN of the transaction is assigned and recorded in the transaction table.
- The log writer process (LGWR) process writes remaining redo log entries in the redo log buffers to the online redo log and writes the transaction SCN to the online redo log. *This atomic event constitutes the commit of the transaction.*
- Oracle Database releases locks held on rows and tables.

  Users who were enqueued waiting on locks held by the uncommitted transaction are allowed to proceed with their work.
- Oracle Database deletes savepoints.

In Table 10-1, no savepoints existed in the `sal_update` transaction so no savepoints were erased.

- Oracle Database performs a commit cleanout.

  If modified blocks containing data from the committed transaction are still in the SGA, and if no other session is modifying them, then the database removes lock-related transaction information (the ITL entry) from the blocks.

  Ideally, the `COMMIT` cleans the blocks so that a subsequent `SELECT` does not have to perform this task. If no ITL entry exists for a specific row, then it is not locked. If an ITL entry exists for a specific row, then it is possibly locked, so a session must check the undo segment header to determine whether this interested transaction has committed. If the interested transaction has committed, then the session cleans out the block, which generates redo. However, if the `COMMIT` cleaned out the ITL previously, then the check and cleanout are unnecessary.

  > **Note:**
  >
  > Because a block cleanout generates redo, a query may generate redo and thus cause blocks to be written during the next checkpoint.

- Oracle Database marks the transaction complete.

After a transaction commits, users can view the changes.

Typically, a commit is a fast operation, regardless of the transaction size. The speed of a commit does not increase with the size of the data modified in the transaction. The lengthiest part of the commit is the physical disk I/O performed by LGWR. However, the amount of time spent by LGWR is reduced because it has been incrementally writing the contents of the redo log buffer in the background.

The default behavior is for LGWR to write redo to the online redo log synchronously and for transactions to wait for the buffered redo to be on disk before returning a commit to the user. However, for lower transaction commit latency, application developers can specify that redo be written asynchronously so that transactions need not wait for the redo to be on disk and can return from the `COMMIT` call immediately.

> **See Also:**
>
> - "Serializable Isolation Level"
> - "Locking Mechanisms"
> - "Overview of Background Processes" for more information about LGWR
> - *Oracle Database PL/SQL Packages and Types Reference* for more information on asynchronous commit

Overview of Transaction Guard

Transaction Guard is an API that applications can use to provide **transaction idempotence**, which is the ability of the database to preserve a guaranteed commit

outcome that indicates whether a transaction committed and completed. Oracle Database provides the API for JDBC thin, OCI, OCCI, and ODP.Net.

A recoverable error is caused by an external system failure, independent of the application session logic that is executing. Recoverable errors occur following planned and unplanned outages of foreground processes, networks, nodes, storage, and databases. If an outage breaks the connection between a client application and the database, then the application receives a disconnection error message. The transaction that was running when the connection broke is called an in-flight transaction.

To decide whether to resubmit the transaction or to return the result (committed or uncommitted) to the client, the application must determine the outcome of the in-flight transaction. Before Oracle Database 12c, commit messages returned to the client were not persistent. Checking a transaction was no guarantee that it would not commit after being checked, permitting duplicate transactions and other forms of logical corruption. For example, a user might refresh a web browser when purchasing a book online and be charged twice for the same book.

See Also:

- "Introduction to Transactions "
- *Oracle Database Development Guide* to learn about Transaction Guard
- *Oracle Real Application Clusters Administration and Deployment Guide* to learn how to configure services for Transaction Guard

Benefits of Transaction Guard

Starting in Oracle Database 12c, Transaction Guard provides applications with a tool for determining the status of an in-flight transaction following a recoverable outage.

Using Transaction Guard, an application can ensure that a transaction executes no more than once. For example, if an online bookstore application determines that the previously submitted commit failed, then the application can safely resubmit.

Transaction Guard provides a tool for at-most-once execution to avoid the application executing duplicate submissions. Transaction Guard provides a known outcome for every transaction.

Transaction Guard is a core Oracle Database capability. Application Continuity uses Transaction Guard when masking outages from end users. Without Transaction Guard, an application retrying after an error may cause duplicate transactions to be committed.

See Also:

- "Overview of Application Continuity" to learn about Application Continuity, which works with Transaction Guard to help developers achieve high application availability
- *Oracle Database Development Guide* to learn about Transaction Guard, including the types of supported and included transactions

How Transaction Guard Works

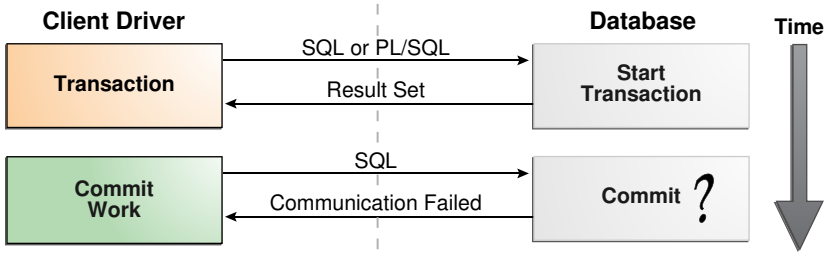
This section explains the problem of lost commit messages and how Transaction Guard uses logical transaction IDs to solve the problem.

Lost Commit Messages

When designing for idempotence, developers must address the problem of communication failures after submission of commit statements. Commit messages do not persist in the database and so cannot be retrieved after a failure.

The following graphic is a high-level representation of an interaction between a client application and a database.

Figure 10-2 Lost Commit Message



In the standard commit case, the database commits a transaction and returns a success message to the client. In Figure 10-2, the client submits a commit statement and receives a message stating that communication failed. This type of failure can occur for several reasons, including a database instance failure or network outage. In this scenario, the client does not know the state of the transaction.

Following a communication failure, the database may still be running the submission and be unaware that the client disconnected. Checking the transaction state does not guarantee that an active transaction will not commit after being checked. If the client resends the commit because of this out-of-date information, then the database may repeat the transaction, resulting in logical corruption.

Logical Transaction ID

Oracle Database solves the communication failure by using a globally unique identifier called a **logical transaction ID**.

This ID contains the logical session number allocated when a session first connects, and a running commit number that is updated each time the session commits or rolls back.[1] From the application perspective, the logical transaction ID uniquely identifies the last database transaction submitted on the session that failed.

For each round trip from the client in which one or more transactions are committed, the database stores a logical transaction ID. This ID can provide transaction idempotence for interactions between the application and the database for each round trip that commits data.

The at-most-once protocol enables access to the commit outcome by requiring the database to do the following:

- Maintain the logical transaction ID for the retention period agreed for retry
- Persist the logical transaction ID on commit

While a transaction is running, both the database and client hold the logical transaction ID. The database gives the client a logical transaction ID at authentication, when borrowing from a connection pool, and at each round trip from the client driver that executes one or more commit operations.

Before the application can determine the outcome of the last transaction following a recoverable error, the application obtains the logical transaction ID held at the client using Java, OCI, OCCI, or ODP.Net APIs. The application then invokes the PL/SQL procedure `DBMS_APP_CONT.GET_LTXID_OUTCOME` with the logical transaction ID to determine the outcome of the last submission: committed (`true` or `false`) and user call completed (`true` or `false`).

When using Transaction Guard, the application can replay transactions when the error is recoverable and the last transaction on the session has not committed. The application can continue when the last transaction has committed and the user call has completed. The application can use Transaction Guard to return the known outcome to the client so that the client can decide the next action to take.

See Also:

- *Oracle Database Development Guide* to learn about logical transaction IDs
- *Oracle Database PL/SQL Packages and Types Reference* to learn more about the `DBMS_APP_CONT.GET_LTXID_OUTCOME` procedure

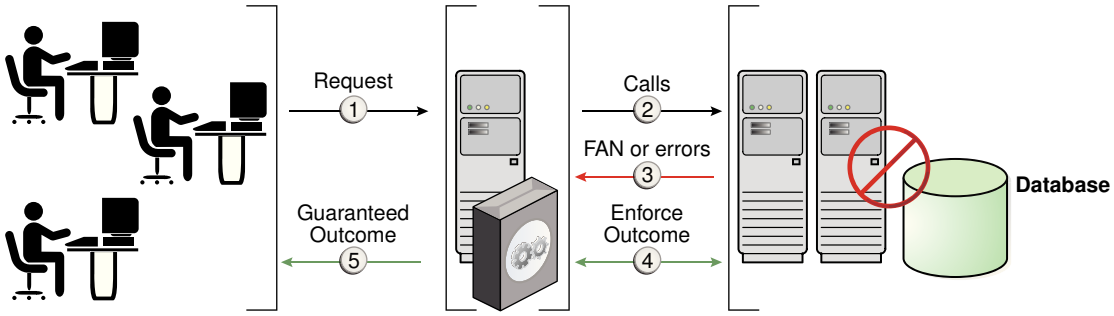
Transaction Guard: Example

In this scenario, the commit message is lost because of a recoverable error.

Transaction Guard uses the logical transaction ID to preserve the outcome of the `COMMIT` statement, ensuring that there is a known outcome for the transaction.

[1] For Oracle Real Application Clusters (Oracle RAC), the logical transaction ID includes the database instance number as a prefix.

Figure 10-3 Check of Logical Transaction Status



In Figure 10-3, the database informs the application whether the transaction committed and whether the last user call completed. The application can then return the result to the end user. The possibilities are:

- If the transaction committed and the user call completed, then the application can return the result to the end user and continue.
- If the transaction committed but the user call did not complete, then the application can return the result to the end user with warnings. Examples include a lost out bind or lost number of rows processed. Some applications depend on the extra information, whereas others do not.
- If the user call was not committed, then the application can return this information to the end user, or safely resubmit. The protocol is guaranteed. When the commit status returns false, the last submission is blocked from committing.

> **See Also:**
>
> *Oracle Database Development Guide* to learn how to use Transaction Guard

Overview of Application Continuity

Application Continuity attempts to mask outages from applications by replaying incomplete application requests after unplanned and planned outages. In this context, a request is a unit of work from the application.

Typically, a request corresponds to the DML statements and other database calls of a single web request on a single database connection. In general, a request is demarcated by the calls made between check-out and check-in of a database connection from a connection pool.

This section contains the following topics:

- Benefits of Application Continuity
- Application Continuity Architecture

Benefits of Application Continuity

A basic problem for developers is how to mask a lost database session from end users.

Application Continuity attempts to solve the lost session problem by restoring the database session when any component disrupts the conversation between database and client. The restored database session includes all states, cursors, variables, and the most recent transaction when one exists.

Use Case for Application Continuity

In a typical case, a client has submitted a request to the database, which has built up both transactional and nontransactional states.

The state at the client remains current, potentially with entered data, returned data, and cached data and variables. However, the database session state, which the application needs to operate within, is lost.

If the client request has initiated one or more transactions, then the application is faced with the following possibilities:

- If a commit *has* been issued, then the commit message returned to the client is not durable. The client does not know whether the request committed, and where in the nontransactional processing state it reached.
- If a commit has *not* been issued, or if it was issued but did not execute, then the in-flight transaction is rolled back and must be resubmitted using a session in the correct state.

If the replay is successful, then database user service for planned and unplanned outages is not interrupted. If the database detects changes in the data seen and potentially acted on by the application, then the replay is rejected. Replay is not attempted when the time allowed for starting replay is exceeded, the application uses a restricted call, or the application has explicitly disabled replay using the `disableReplay` method.

> **See Also:**
>
> *Oracle Real Application Clusters Administration and Deployment Guide* to learn more about how Application Continuity works for database sessions

Application Continuity for Planned Maintenance

Application Continuity for planned outages enables applications to continue operations for database sessions that can be reliably drained or migrated.

Scheduled maintenance need not disrupt application work. Application Continuity gives active work time to drain from its current location to a new location currently unaffected by maintenance. At the end of the drain interval, sessions may remain on the database instance where maintenance is planned. Instead of forcibly disconnecting these sessions, Application Continuity can fail over these sessions to a surviving site, and resubmit any in-flight transactions.

With Application Continuity enabled the database can do the following:

- Report no errors for either incoming or existing work during maintenance
- Redirect active database sessions to other functional services
- Rebalance database sessions, as needed, during and after the maintenance

Control the drain behavior during planned maintenance using the `drain_timeout` and `stop_option` service attributes of the SRVCTL utility, Global Data Services Control Utility (GDSCTL), and Oracle Data Guard Broker. The `DBMS_SERVICE` package provides the underlying infrastructure.

> **See Also:**
>
> - *Oracle Real Application Clusters Administration and Deployment Guide* to learn more Application Continuity
> - *Oracle Database PL/SQL Packages and Types Reference* to learn more about `DBMS_SERVICE`
> - *Oracle Real Application Clusters Administration and Deployment Guide* for the SRVCTL command reference
> - *Oracle Database Global Data Services Concepts and Administration Guide* for the GDSCTL command reference

Application Continuity Architecture

The key components of Application Continuity are runtime, reconnection, and replay.

The phases are as follows:

1. Normal runtime

   In this phase, Application Continuity performs the following tasks:

   - Identifies database requests
   - Decides whether local and database calls are replayable
   - Builds proxy objects to enable replay, if necessary, and to manage queues
   - Holds original calls and validation of these calls until the end of the database request or replay is disabled

2. Reconnection

   This phase is triggered by a recoverable error. Application Continuity performs the following tasks:

   - Ensures that replay is enabled for the database requests
   - Manages timeouts
   - Obtains a new connection to the database, and then validates that this is a valid database target
   - Uses Transaction Guard to determine whether the last transaction committed successfully (committed transactions are not resubmitted)

3. Replay

Application Continuity performs the following tasks:

- Replays calls that are held in the queue
- Disables replay if user-visible changes in results appear during the replay
- Does not allow a commit, but does allow the last recall (which encountered the error) to commit

Following a successful replay, the request continues from the point of failure.

See Also:

- "Overview of Transaction Guard"
- *Oracle Real Application Clusters Administration and Deployment Guide* to learn more about Application Continuity
- *Oracle Database JDBC Developer's Guide* and *Oracle Database JDBC Java API Reference* to learn more about JDBC and Application Continuity

Overview of Autonomous Transactions

An **autonomous transaction** is an independent transaction that can be called from another transaction, which is the main transaction. You can suspend the calling transaction, perform SQL operations and commit or undo them in the autonomous transaction, and then resume the calling transaction.

Autonomous transactions are useful for actions that must be performed independently, regardless of whether the calling transaction commits or rolls back. For example, in a stock purchase transaction, you want to commit customer data regardless of whether the overall stock purchase goes through. Additionally, you want to log error messages to a debug table even if the overall transaction rolls back.

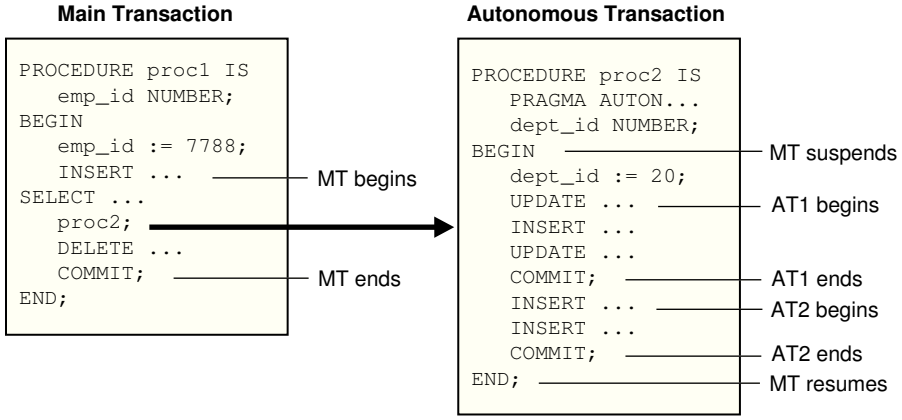
Autonomous transactions have the following characteristics:

- The autonomous transaction does not see uncommitted changes made by the main transaction and does not share locks or resources with the main transaction.
- Changes in an autonomous transaction are visible to other transactions upon commit of the autonomous transactions. Thus, users can access the updated information without having to wait for the main transaction to commit.
- Autonomous transactions can start other autonomous transactions. There are no limits, other than resource limits, on how many levels of autonomous transactions can be called.

In PL/SQL, an autonomous transaction executes within an *autonomous scope*, which is a routine marked with the pragma `AUTONOMOUS_TRANSACTION`. In this context, routines include top-level anonymous PL/SQL blocks and PL/SQL subprograms and triggers. A pragma is a directive that instructs the compiler to perform a compilation option. The pragma `AUTONOMOUS_TRANSACTION` instructs the database that this procedure, when executed, is to be executed as a new autonomous transaction that is independent of its parent transaction.

The following graphic shows how control flows from the main routine (MT) to an autonomous routine and back again. The main routine is `proc1` and the autonomous routine is `proc2`. The autonomous routine can commit multiple transactions (AT1 and AT2) before control returns to the main routine.

Figure 10-4 Transaction Control Flow



When you enter the executable section of an autonomous routine, the main routine suspends. When you exit the autonomous routine, the main routine resumes.

In Figure 10-4, the `COMMIT` inside `proc1` makes permanent not only its own work but any outstanding work performed in its session. However, a `COMMIT` in `proc2` makes permanent only the work performed in the `proc2` transaction. Thus, the `COMMIT` statements in transactions AT1 and AT2 have no effect on the MT transaction.

> **See Also:**
>
> *Oracle Database Development Guide* and *Oracle Database PL/SQL Language Reference* to learn how to use autonomous transactions

Overview of Distributed Transactions

A **distributed transaction** is a transaction that includes one or more statements that update data on two or more distinct nodes of a distributed database, using a schema object called a database link.

A distributed database is a set of databases in a distributed system that can appear to applications as a single data source. A database link describes how one database instance can log in to another database instance.

Unlike a transaction on a local database, a distributed transaction alters data on multiple databases. Consequently, distributed transaction processing is more complicated because the database must coordinate the committing or rolling back of the changes in a transaction as an atomic unit. The entire transaction must commit or roll back. Oracle Database must coordinate transaction control over a network and maintain data consistency, even if a network or system failure occurs.

> **See Also:**
>
> *Oracle Database Administrator's Guide* to learn how to manage distributed transactions

Two-Phase Commit

The **two-phase commit mechanism** guarantees that *all* databases participating in a distributed transaction either all commit or all undo the statements in the transaction. The mechanism also protects implicit DML performed by integrity constraints, remote procedure calls, and triggers.

In a two-phase commit among multiple databases, one database coordinates the distributed transaction. The initiating node is called the *global coordinator*. The coordinator asks the other databases if they are prepared to commit. If any database responds with a no, then the entire transaction is rolled back. If all databases vote yes, then the coordinator broadcasts a message to make the commit permanent on each of the databases.

The two-phase commit mechanism is transparent to users who issue distributed transactions. In fact, users need not even know the transaction is distributed. A `COMMIT` statement denoting the end of a transaction automatically triggers the two-phase commit mechanism. No coding or complex statement syntax is required to include distributed transactions within the body of a database application.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* to learn about the two-phase commit mechanism
> - *Oracle Database SQL Language Reference*

In-Doubt Transactions

An **in-doubt distributed transaction** occurs when a two-phase commit was interrupted by any type of system or network failure.

For example, two databases report to the coordinating database that they were prepared to commit, but the coordinating database instance fails immediately after receiving the messages. The two databases who are prepared to commit are now left hanging while they await notification of the outcome.

The recoverer (`RECO`) background process automatically resolves the outcome of in-doubt distributed transactions. After the failure is repaired and communication is reestablished, the `RECO` process of each local Oracle database automatically commits or rolls back any in-doubt distributed transactions consistently on all involved nodes.

In the event of a long-term failure, Oracle Database enables each local administrator to manually commit or undo any distributed transactions that are in doubt because of the failure. This option enables the local database administrator to free any locked resources that are held indefinitely because of the long-term failure.

If a database must be recovered to a past time, then database recovery facilities enable database administrators at other sites to return their databases to the earlier point in time. This operation ensures that the global database remains consistent.

See Also:

- "Recoverer Process (RECO) "
- *Oracle Database Administrator's Guide* to learn how to manage in-doubt transactions