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 comply with 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

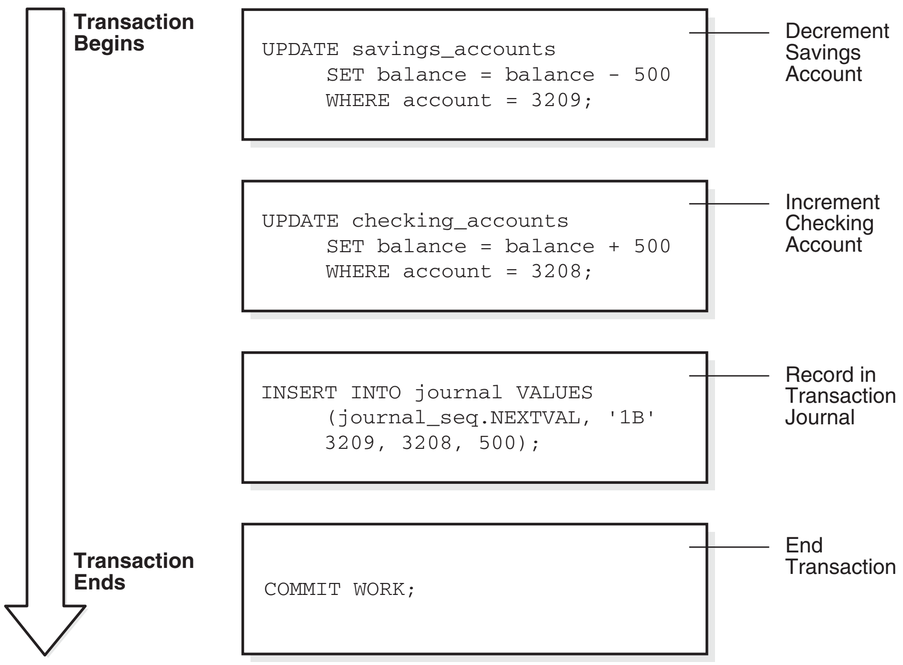
To illustrate the concept of a transaction, consider a banking database. When a customer transfers money from a savings account to a checking account, the transaction must consist of three separate operations:

- 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.

Figure 10–1 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



```
UPDATE savings_accounts
     SET balance = balance - 500
     WHERE account = 3209;
```

```
UPDATE checking_accounts
     SET balance = balance + 500
     WHERE account = 3208;
```

```
INSERT INTO journal VALUES
     (journal_seq.NEXTVAL, '1B'
     3209, 3208, 500);
```

```
COMMIT WORK;
```

Transaction Begins
Decrement Savings Account
Increment Checking Account
Record in Transaction Journal
Transaction Ends
End 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" on page 7-3

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" on page 12-24

End of a Transaction

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. The transaction shown in Figure 10–1 ends with a commit.

- 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:

- "Tools for Database Administrators" on page 18-2 and "Tools for Database Developers" on page 19-1
- *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 Figure 10–1 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" on page 7-16
- "Locks and Deadlocks" on page 9-16
- "Overview of Triggers" on page 8-16

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 with respect to the database. Several events may share the same SCN, which means that they occurred at the same time with respect to 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" on page 13-12 and "Backup and Recovery" on page 18-9

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" on page 7-8:

- 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

Time	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`.

Table 10–1 (Cont.) Transaction Control

Time	Session	Explanation

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.

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 on page 10-6, 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" on page 18-2
- *Oracle Database Reference* to learn about `V$TRANSACTION`
- *Oracle Database SQL Language Reference* to learn about `SET TRANSACTION`

Active Transactions

An **active transaction** has started but not yet committed or rolled back. In Table 10–1 on page 10-6, 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 follows:

- Oracle Database has generated **undo data** information in the **system global area (SGA)**.

  The undo data contains the old data values changed by the SQL statements of the transaction. See "Read Consistency in the Read Committed Isolation Level" on page 9-7.

- 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. See "Redo Log Buffer" on page 14-14.

- 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. See "Database Buffer Cache" on page 14-9.

- 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. See "Summary of Locking Behavior" on page 9-12.

Savepoints

A **savepoint** is a user-declared intermediate marker within the context of a transaction. Internally, this 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 on page 10-6 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 on page 10-6, the following occurs:

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

   In Table 10–1 on page 10-6, 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 on page 10-6, 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 previous to 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 Table 10–2, 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 10–2 Rollback to Savepoint Example

Time	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.
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" on page 9-16 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 on page 10-6, 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 on page 10-6, 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 on page 10-6, 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" on page 12-24

Committing Transactions

A **commit** ends the current transaction and makes permanent all changes performed in the transaction. In Table 10–1 on page 10-6, 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:

- A **system change number (SCN)** is generated 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. See "Serializable Isolation Level" on page 9-8.

- The **log writer (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 on page 10-6, 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 from the blocks. Ideally, the `COMMIT` cleans out the blocks so that a subsequent `SELECT` does not have to perform this task.

  > **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:

- *Oracle Database PL/SQL Language Reference* for more information on asynchronous commit
- "Locking Mechanisms" on page 9-5
- "Overview of Background Processes" on page 15-7 for more information about LGWR