1. Introduction to Transactions

A **transaction** is a logical unit of work that contains one or more SQL statements. A transaction is an atomic unit. The effects of all the SQL statements in a transaction can be either all **committed** (applied to the database) or all **rolled back** (undone from the database).

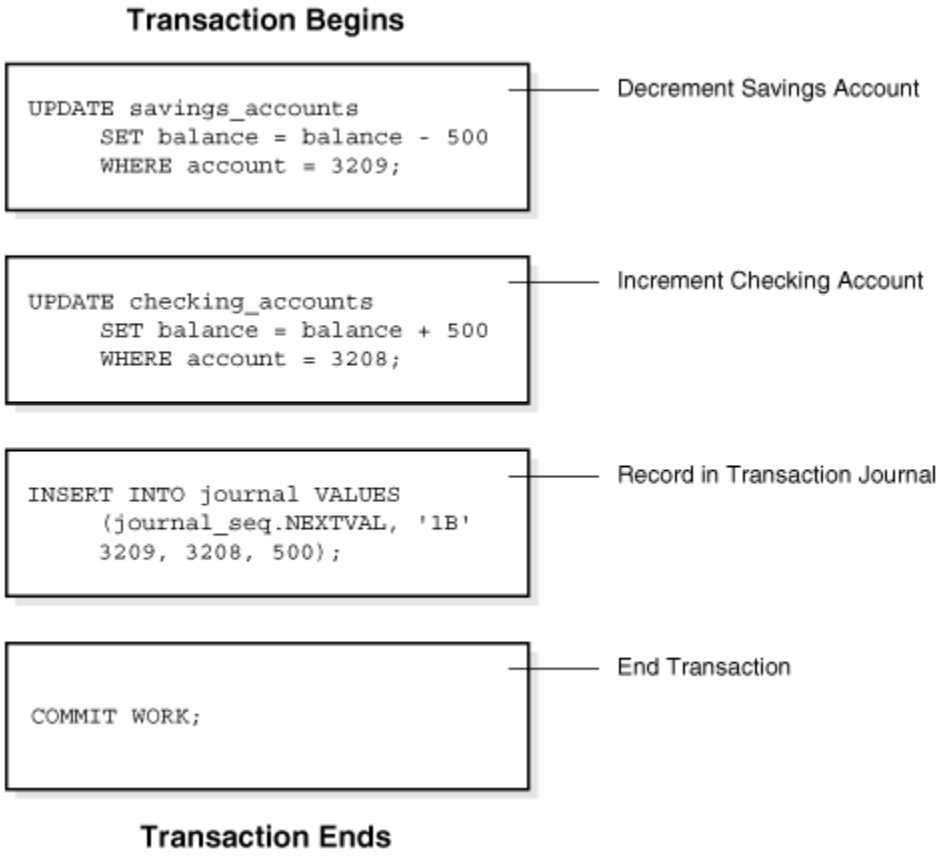
A transaction begins with the first executable SQL statement. A transaction ends when it is committed or rolled back, either explicitly with a `COMMIT` or `ROLLBACK` statement or implicitly when a DDL statement is issued.

Sample Transaction: Account Debit and Credit

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.

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.

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:

```
create table Project (id number primary key,
                      projectName varchar2(50),
                      cost    number);
insert into project values (1, 'JUPITER', 2000);
insert into project values (2, 'Saturn', 1000);
insert into project values (3, 'Mercury', 15000);

SQL> UPDATE project SET cost= cost;
-- rows updated.

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

XID                 NAME        STATUS
---------------- ----------  ----------------
09000700D5020000       (null)  ACTIVE
```

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. The transaction shown in "Sample Transaction: Account Debit and Credit" 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.
- A client process terminates abnormally, causing the transaction to be implicitly rolled back using metadata stored in the transaction table and the undo segment.

Statement Execution and Transaction Control

A SQL statement that runs successfully is different from a committed transaction. Executing successfully means that a single statement was:

- Parsed
- Found to be a valid SQL construction
- Run without error as an atomic unit. For example, all rows of a multirow update are changed.However, until the transaction that contains the statement is committed, the transaction can be rolled back, and all of the changes of the statement can be undone. A statement, rather than a transaction, runs successfully.

Commit Transactions

Committing means that a user has explicitly or implicitly requested that the changes in the transaction be made permanent. An explicit request occurs when the user issues a `COMMIT` statement. An implicit request occurs after normal termination of an application or completion of a data definition language (DDL) operation. The changes made by the SQL statement(s) of a transaction become permanent and visible to other users only after that transaction commits. Queries that are issued after the transaction commits will see the committed changes.

Rollback of Transactions

Use the `ROLLBACK` statement to undo work done in the current transaction or to manually undo the work done by an in-doubt distributed transaction.

Using `ROLLBACK` without the `TO SAVEPOINT` clause performs the following operations:

- Ends the transaction
- Undoes all changes in the current transaction
- Erases all savepoints in the transaction
- Releases any transaction locks

Savepoint in Transactions

The SAVEPOINT statement names and marks the current point in the processing of a transaction. With the ROLLBACK TO statement, savepoints undo parts of a transaction instead of the whole transaction.

A simple rollback or commit erases all savepoints. When you roll back to a savepoint, any savepoints marked after that savepoint are erased. The savepoint to which you roll back remains.

An implicit savepoint is marked before executing an INSERT, UPDATE, or DELETE statement. If the statement fails, a rollback to the implicit savepoint is done. Normally, just the failed SQL statement is rolled back, not the whole transaction; if the statement raises an unhandled exception, the host environment (such as SQL*Plus) determines what is rolled back.

Examples

The following statement rolls back your entire current transaction:

```
ROLLBACK;
```

The following statement rolls back your current transaction to savepoint `banda_sal`:

```
ROLLBACK TO SAVEPOINT banda_sal;
```

Note that a RAISE_APPLICATION_ERROR or a RAISE [exception name] statement will also automatically rollback your PL/SQL block as a single atomic unit. Which is of course a desirable effect as it doesn't leave you with uncommitted changes.

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.

Time	Session	Explanation
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.
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.

2. Overview of Transaction Processing in PL/SQL

Using COMMIT in PL/SQL

The `COMMIT` statement ends the current transaction, making any changes made during that transaction permanent, and visible to other users. Transactions are not tied to PL/SQL `BEGIN-END` blocks. A block can contain multiple transactions, and a transaction can span multiple blocks.

```
CREATE TABLE accounts (account_id NUMBER(6), balance NUMBER (10,2));

INSERT INTO accounts VALUES (7715, 6350.00);

INSERT INTO accounts VALUES (7720, 5100.50);

DECLARE

  transfer NUMBER(8,2) := 250;

BEGIN

  UPDATE accounts SET balance = balance - transfer WHERE account_id = 7715;

  UPDATE accounts SET balance = balance + transfer WHERE account_id = 7720;

  COMMIT;

END;

/
```

Using ROLLBACK in PL/SQL

The `ROLLBACK` statement ends the current transaction and undoes any changes made during that transaction. If you make a mistake, such as deleting the wrong row from a table, a rollback restores the original data. If you cannot finish a transaction because an exception is raised or a SQL statement fails, a rollback lets you take corrective action and perhaps start over.

```
CREATE TABLE emp_name AS SELECT employee_id, last_name FROM employees;
CREATE UNIQUE INDEX empname_ix ON emp_name (employee_id);
CREATE TABLE emp_sal AS SELECT employee_id, salary FROM employees;
CREATE UNIQUE INDEX empsal_ix ON emp_sal (employee_id);
CREATE TABLE emp_job AS SELECT employee_id, job_id FROM employees;
CREATE UNIQUE INDEX empjobid_ix ON emp_job (employee_id);
```

```
DECLARE
   emp_id       NUMBER(6);
   emp_lastname VARCHAR2(25);
   emp_salary   NUMBER(8,2);
   emp_jobid    VARCHAR2(10);
BEGIN
   SELECT employee_id, last_name, salary, job_id INTO emp_id, emp_lastname,
     emp_salary, emp_jobid FROM employees WHERE employee_id = 120;
   INSERT INTO emp_name VALUES (emp_id, emp_lastname);
   INSERT INTO emp_sal VALUES (emp_id, emp_salary);
   INSERT INTO emp_job VALUES (emp_id, emp_jobid);
EXCEPTION
   WHEN DUP_VAL_ON_INDEX THEN
      ROLLBACK;
      DBMS_OUTPUT.PUT_LINE('Inserts have been rolled back');
END;
/
```

Using SAVEPOINT in PL/SQL

`SAVEPOINT` names and marks the current point in the processing of a transaction. Savepoints let you roll back part of a transaction instead of the whole transaction. The number of active savepoints for each session is unlimited.

```
CREATE TABLE emp_name AS SELECT employee_id, last_name, salary FROM
employees;
CREATE UNIQUE INDEX empname_ix ON emp_name (employee_id);


DECLARE
   emp_id        employees.employee_id%TYPE;
   emp_lastname  employees.last_name%TYPE;
   emp_salary    employees.salary%TYPE;
BEGIN
   SELECT employee_id, last_name, salary INTO emp_id, emp_lastname,
     emp_salary FROM employees WHERE employee_id = 120;
   UPDATE emp_name SET salary = salary * 1.1 WHERE employee_id = emp_id;
   DELETE FROM emp_name WHERE employee_id = 130;
   SAVEPOINT do_insert;
   INSERT INTO emp_name VALUES (emp_id, emp_lastname, emp_salary);
EXCEPTION
   WHEN DUP_VAL_ON_INDEX THEN
```

```
      ROLLBACK TO do_insert;
      DBMS_OUTPUT.PUT_LINE('Insert has been rolled back');
END;
/
```

If you exit a stored subprogram with an unhandled exception, PL/SQL does not assign values to `OUT` parameters, and does not do any rollback.