DATABASE MANAGEMENT SYSTEMS LAB FILE

Submitted To: Prof. S. Sivaprasad Kumar
Mr. Anshul Arora
Submitted By: Arohan Sharma
Roll Number: 2K16/MC/22

Index

SNO.	EXPERIMENT	DATE	SIGNATURE
1.	Create a database and table in MySQL and use DDL Commands		
2.	Use DML commands on a table in MySQL		
3.	Use the SELECT, Group By, Order By and default system commands		
4.	Writing nested sub-queries and implementing joins in MySQL		

Experiment 1

Aim: Create a database and table in MySQL and use DDL Commands

Query 1: Creating a database and a table and inserting values in the table

```
Arohans-MacBook-Pro:~ Arohan$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 11
Server version: 8.0.13 MySQL Community Server - GPL

Copyright (c) 2000, 2018, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> create database employee;
Query OK, 1 row affected (0.04 sec)

mysql> use employee;
Database changed
mysql> create table emp;
ERROR 1113 (42000): A table must have at least 1 column
mysql> create table emp(first_name char(50), last_name char(50), office_code int (3));
Query OK, 0 rows affected (0.02 sec)

mysql> insert into emp values("Arohan", "Sharma", 101);
Query OK, 1 row affected (0.01 sec)

mysql> insert into emp values("Apurva", "Puri", 302);
Query OK, 1 row affected (0.05 sec)

mysql> insert into emp values("Ansh", "Agrawal", 211);
Query OK, 1 row affected (0.10 sec)

mysql> select * from emp;
+------------+-----------+-------------+
| first_name | last_name | office_code |
+------------+-----------+-------------+
| Arohan     | Sharma    |         101 |
| Apurva     | Puri      |         302 |
| Ansh       | Agrawal   |         211 |
+------------+-----------+-------------+
3 rows in set (0.00 sec)

mysql>
```

Query 2: Use Alter table to edit structure of table

i) Add a column

```
mysql> alter table emp add salary int(6);
Query OK, 0 rows affected (0.11 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> select * from emp;
+------------+-----------+-------------+--------+
| first_name | last_name | office_code | salary |
+------------+-----------+-------------+--------+
| Arohan     | Sharma    |         101 |   NULL |
| Apurva     | Puri      |         302 |   NULL |
| Ansh       | Agrawal   |         211 |   NULL |
+------------+-----------+-------------+--------+
3 rows in set (0.00 sec)
```

ii) Drop a column

```
mysql> alter table emp drop last_name;
Query OK, 0 rows affected (0.15 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> select * from emp;
+------------+-------------+--------+
| first_name | office_code | salary |
+------------+-------------+--------+
| Arohan     |         101 |   NULL |
| Apurva     |         302 |   NULL |
| Ansh       |         211 |   NULL |
+------------+-------------+--------+
3 rows in set (0.00 sec)

mysql>
```

iii) Modify a column

```
mysql> alter table emp modify column first_name char(50) NOT NULL;
Query OK, 0 rows affected (0.10 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> select * from emp;
+------------+-------------+--------+
| first_name | office_code | salary |
+------------+-------------+--------+
| Arohan     |         101 |   NULL |
| Apurva     |         302 |   NULL |
| Ansh       |         211 |   NULL |
+------------+-------------+--------+
3 rows in set (0.00 sec)
```

iv) Rename a column

```
mysql> alter table emp change first_name fname char(50);
Query OK, 0 rows affected (0.22 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> select * from emp;
+--------+-------------+--------+
| fname  | office_code | salary |
+--------+-------------+--------+
| Arohan |         101 |   NULL |
| Apurva |         302 |   NULL |
| Ansh   |         211 |   NULL |
+--------+-------------+--------+
3 rows in set (0.00 sec)
```

Query 3: Drop a table

```
mysql> drop table emp;
Query OK, 0 rows affected (0.08 sec)

mysql> select * from emp;
ERROR 1146 (42S02): Table 'employee.emp' doesn't exist
mysql>
```

Experiment 2

Aim: Use DML commands on a table in MySQL

Query 1: Insert values in a table

```
mysql> insert into emp values("Arohan", "Sharma", 101);
Query OK, 1 row affected (0.01 sec)

mysql> insert into emp values("Apurva", "Puri", 302);
Query OK, 1 row affected (0.05 sec)

mysql> insert into emp values("Ansh", "Agrawal", 211);
Query OK, 1 row affected (0.10 sec)

mysql> select * from emp;
+------------+-----------+-------------+
| first_name | last_name | office_code |
+------------+-----------+-------------+
| Arohan     | Sharma    |         101 |
| Apurva     | Puri      |         302 |
| Ansh       | Agrawal   |         211 |
+------------+-----------+-------------+
3 rows in set (0.00 sec)

mysql>
```

Query 2: Use update command to update values of an attribute

```
mysql> update emp set first_name = "Kush", last_name = "Arora"
    -> where office_code = 211;
Query OK, 1 row affected (0.10 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> select * from emp;
+------------+-----------+-------------+
| first_name | last_name | office_code |
+------------+-----------+-------------+
| Arohan     | Sharma    |         101 |
| Apurva     | Puri      |         302 |
| Kush       | Arora     |         211 |
+------------+-----------+-------------+
3 rows in set (0.00 sec)
```

Query 3: Use delete command to delete a row

```
mysql> delete from emp where office_code = 211;
Query OK, 1 row affected (0.08 sec)

mysql> select * from emp;
+------------+-----------+-------------+
| first_name | last_name | office_code |
+------------+-----------+-------------+
| Arohan     | Sharma    |         101 |
| Apurva     | Puri      |         302 |
+------------+-----------+-------------+
2 rows in set (0.00 sec)

mysql>
```

Query 4: Use truncate command to truncate a table

```
mysql> truncate emp;
Query OK, 0 rows affected (0.06 sec)

mysql> select * from emp;
Empty set (0.00 sec)
```

Experiment 4

Aim: Use the SELECT, Group By, Order By and default system commands.

Query 1: Use select *

```
mysql> select * from emp;
+------------+-----------+-------------+
| first_name | last_name | office_code |
+------------+-----------+-------------+
| Arohan     | Sharma    |         101 |
| Apurva     | Puri      |         302 |
| Ansh       | Agrawal   |         211 |
+------------+-----------+-------------+
3 rows in set (0.00 sec)

mysql>
```

Query 2: Use select command with where clause

```
mysql> select * from emp where office_code = 211;
+------------+-----------+-------------+
| first_name | last_name | office_code |
+------------+-----------+-------------+
| Ansh       | Agrawal   |         211 |
+------------+-----------+-------------+
1 row in set (0.00 sec)

mysql>
```

Query 3: Use select command with where and in clause

```
mysql> select * from emp where first_name in ("Arohan", "Ansh")
    -> ;
+------------+-----------+-------------+
| first_name | last_name | office_code |
+------------+-----------+-------------+
| Arohan     | Sharma    |         101 |
| Ansh       | Agrawal   |         211 |
+------------+-----------+-------------+
2 rows in set (0.00 sec)

mysql>
```

Query 4: Use aggregate functions on a table

```
mysql> select * from employee
    -> ;
+------------+---------+-------------+--------+
| first_name | City    | office_code | salary |
+------------+---------+-------------+--------+
| Arohan     | Delhi   |          10 |  23400 |
| Ansh       | Mumbai  |          20 |  23540 |
| Apurva     | Delhi   |          10 |  39000 |
| Kush       | Mumbai  |          20 |  32109 |
| Arjun      | Chennai |          30 |  31000 |
| Anjali     | Delhi   |          10 |  30000 |
+------------+---------+-------------+--------+
6 rows in set (0.00 sec)

mysql> select avg(salary) from employee;
+-------------+
| avg(salary) |
+-------------+
|  29841.5000 |
+-------------+
1 row in set (0.00 sec)

mysql> select count(salary) from employee;
+---------------+
| count(salary) |
+---------------+
|             6 |
+---------------+
1 row in set (0.00 sec)

mysql>
```

Query 5: Use group by command on a table

```
mysql> select city, avg(salary) from employee
    -> group by city;
+---------+-------------+
| city    | avg(salary) |
+---------+-------------+
| Delhi   |  30800.0000 |
| Mumbai  |  27824.5000 |
| Chennai |  31000.0000 |
+---------+-------------+
3 rows in set (0.00 sec)

mysql>
```

Query 6: Use group by and having command on a table

```
mysql> select city, avg(salary) from employee
    -> group by city
    -> having count(salary) > 1;
+--------+-------------+
| city   | avg(salary) |
+--------+-------------+
| Delhi  |  30800.0000 |
| Mumbai |  27824.5000 |
+--------+-------------+
2 rows in set (0.00 sec)

mysql>
```

Query 7: Use order by on a table

```
mysql> select * from employee
    -> order by salary desc;
+------------+---------+-------------+--------+
| first_name | City    | office_code | salary |
+------------+---------+-------------+--------+
| Apurva     | Delhi   |          10 |  39000 |
| Kush       | Mumbai  |          20 |  32109 |
| Arjun      | Chennai |          30 |  31000 |
| Anjali     | Delhi   |          10 |  30000 |
| Ansh       | Mumbai  |          20 |  23540 |
| Arohan     | Delhi   |          10 |  23400 |
+------------+---------+-------------+--------+
6 rows in set (0.00 sec)

mysql>
```

Query 8: Create a “view” and display it

```
mysql> create view Office as select city, office_code from employee;
Query OK, 0 rows affected (0.01 sec)

mysql> select * from Office;
+---------+-------------+
| city    | office_code |
+---------+-------------+
| Delhi   |          10 |
| Mumbai  |          20 |
| Delhi   |          10 |
| Mumbai  |          20 |
| Chennai |          30 |
| Delhi   |          10 |
+---------+-------------+
6 rows in set (0.00 sec)

mysql>
```

Query 9: Using in-built functions: current_timestamp and sysdate()

```
mysql> select current_timestamp;
+---------------------+
| current_timestamp   |
+---------------------+
| 2019-04-10 01:08:08 |
+---------------------+
1 row in set (0.00 sec)
```

```
mysql> select sysdate();
+---------------------+
| sysdate()           |
+---------------------+
| 2019-04-10 01:08:29 |
+---------------------+
1 row in set (0.00 sec)

mysql>
```

Experiment 5

Aim: Writing nested sub-queries and implementing joins in MySQL

Query 1: Nested Query 1

```
mysql> select * from employees;
+------------+-----------+-------------+
| first_name | last_name | office_code |
+------------+-----------+-------------+
| Arohan     | Sharma    |          10 |
| Apurva     | Puri      |          20 |
| Ansh       | Agrawal   |          30 |
+------------+-----------+-------------+
3 rows in set (0.00 sec)

mysql> select * from offices;
+-------------+---------+
| office_code | country |
+-------------+---------+
|          10 | INDIA   |
|          20 | USA     |
+-------------+---------+
2 rows in set (0.00 sec)

mysql> SELECT first_name, last_name FROM employees
    -> WHERE office_code IN (SELECT office_code FROM offices WHERE country = 'INDIA');
+------------+-----------+
| first_name | last_name |
+------------+-----------+
| Arohan     | Sharma    |
+------------+-----------+
1 row in set (0.00 sec)

mysql>
```

Query 2: Nested Query 2

```
mysql> SELECT first_name FROM employees WHERE office_code NOT IN (SELECT office_code FROM offices WHERE country = 'USA');
+------------+
| first_name |
+------------+
| Arohan     |
| Ansh       |
+------------+
2 rows in set (0.00 sec)

mysql>
```

Query 3: Nested Query 3

```
mysql> SELECT country FROM offices
    -> WHERE office_code < (SELECT office_code FROM employees WHERE office_code > 20);
+---------+
| country |
+---------+
| INDIA   |
| USA     |
+---------+
2 rows in set (0.01 sec)

mysql>
```

Query 4: Implement Natural Join using both the Tables

```
mysql> SELECT employees.first_name, offices.country FROM employees INNER JOIN offices ON employees.office_code = offices.office_code;
+------------+---------+
| first_name | country |
+------------+---------+
| Arohan     | INDIA   |
| Apurva     | USA     |
| Ansh       | UK      |
+------------+---------+
3 rows in set (0.00 sec)

mysql>
```

EXPERIMENT 6
INTODUCTION TO PL/SQL

PL/SQL is a combination of SQL along with the procedural features of programming languages. It was developed by Oracle Corporation in the early 90's to enhance the capabilities of SQL.

VARIABLES:

Variable Declaration :

PL/SQL variables must be declared in the declaration section or in a package as a global variable. When you declare a variable, PL/SQL allocates memory for the variable's value and the storage location is identified by the variable name.

The syntax for declaring a variable is –

```
variable_name [CONSTANT] datatype [NOT NULL] [:= | DEFAULT initial_value]
```

Where, *variable_name* is a valid identifier in PL/SQL, *datatype* must be a valid PL/SQL data type or any user defined data type which we already have discussed in the last chapter. Some valid variable declarations along with their definition are shown below –

```
sales number(10, 2);
pi CONSTANT double precision := 3.1415;
name varchar2(25);
address varchar2(100);
```

When you provide a size, scale or precision limit with the data type, it is called a **constrained declaration**. Constrained declarations require less memory than unconstrained declarations. For example –

```
sales number(10, 2);
name varchar2(25);
address varchar2(100);
```

Variable Initialisation:

Whenever you declare a variable, PL/SQL assigns it a default value of NULL. If you want to initialize a variable with a value other than the NULL value, you can do so during the declaration, using either of the following –

- The **DEFAULT** keyword
- The **assignment** operator

For example –

```
counter binary_integer := 0;
greetings varchar2(20) DEFAULT 'Have a Good Day';
```

You can also specify that a variable should not have a **NULL** value using the **NOT NULL** constraint. If you use the NOT NULL constraint, you must explicitly assign an initial value for that variable.

It is a good programming practice to initialize variables properly otherwise, sometimes programs would produce unexpected results. Try the following example which makes use of various types of variables –

```
DECLARE
   a integer := 10;
   b integer := 20;
   c integer;
   f real;
BEGIN
   c := a + b;
   dbms_output.put_line('Value of c: ' || c);
   f := 70.0/3.0;
   dbms_output.put_line('Value of f: ' || f);
END;
/
```

When the above code is executed, it produces the following result –

```
Value of c: 30
Value of f: 23.333333333333333


PL/SQL procedure successfully completed.
```

INPUT OUTPUT

Oracle-supplied packages that allow PL/SQL to communicate with external processes, sessions, and files.

The packages are:

- DBMS_PIPE, to send and receive information between sessions, asynchronously.
- DBMS_OUTPUT, to send messages from a PL/SQL program to other PL/SQL programs in the same session, or to a display window running SQL*Plus.
- UTL_FILE, which allows a PL/SQL program to read information from a disk file, and write information to a file.

PACKAGES

Packages are schema objects that groups logically related PL/SQL types, variables, and subprograms.

A package will have two mandatory parts –

- Package specification
- Package body or definition

Package Specification

The specification is the interface to the package. It just **DECLARES** the types, variables, constants, exceptions, cursors, and subprograms that can be referenced from outside the package. In other words, it contains all information about the content of the package, but excludes the code for the subprograms.

All objects placed in the specification are called **public** objects. Any subprogram not in the package specification but coded in the package body is called a **private** object.

The following code snippet shows a package specification having a single procedure. You can have many global variables defined and multiple procedures or functions inside a package.

```
CREATE PACKAGE cust_sal AS
   PROCEDURE find_sal(c_id customers.id%type);
END cust_sal;
/
```

Package Body

The package body has the codes for various methods declared in the package specification and other private declarations, which are hidden from the code outside the package.

The **CREATE PACKAGE BODY** Statement is used for creating the package body. The following code snippet shows the package body declaration for the ***cust_sal*** package created above.

```
CREATE OR REPLACE PACKAGE BODY cust_sal AS

   PROCEDURE find_sal(c_id customers.id%TYPE) IS
   c_sal customers.salary%TYPE;
   BEGIN
      SELECT salary INTO c_sal
      FROM customers
      WHERE id = c_id;
      dbms_output.put_line('Salary: '|| c_sal);
   END find_sal;
END cust_sal;
/
```

PROCEDURES

A procedure is created with the **CREATE OR REPLACE PROCEDURE** statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows –

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
{IS | AS}
BEGIN
  < procedure_body >
END procedure_name;
```

Where,

- *procedure-name* specifies the name of the procedure.
- [OR REPLACE] option allows the modification of an existing procedure.
- The optional parameter list contains name, mode and types of the parameters. **IN** represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
- *procedure-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone procedure.

FUNCTIONS

A standalone function is created using the **CREATE FUNCTION** statement. The simplified syntax for the **CREATE OR REPLACE PROCEDURE** statement is as follows –

```
CREATE [OR REPLACE] FUNCTION function_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
RETURN return_datatype
{IS | AS}
BEGIN
   < function_body >
END [function_name];
```

Where,

- *function-name* specifies the name of the function.
- [OR REPLACE] option allows the modification of an existing function.
- The optional parameter list contains name, mode and types of the parameters. IN represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
- The function must contain a **return** statement.
- The *RETURN* clause specifies the data type you are going to return from the function.
- *function-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone function.

EXPERIMENT 7
EXCEPTION IN PL/SQL

An exception is an error condition during a program execution. PL/SQL supports programmers to catch such conditions using **EXCEPTION** block in the program and an appropriate action is taken against the error condition. There are two types of exceptions –

- System-defined exceptions
- User-defined exceptions

Syntax for Exception Handling

The general syntax for exception handling is as follows. Here you can list down as many exceptions as you can handle. The default exception will be handled using ***WHEN others THEN*** –

```
DECLARE
   <declarations section>
BEGIN
   <executable command(s)>
EXCEPTION
   <exception handling goes here >
   WHEN exception1 THEN
      exception1-handling-statements
   WHEN exception2  THEN
      exception2-handling-statements
   WHEN exception3 THEN
      exception3-handling-statements
   ........
   WHEN others THEN
      exception3-handling-statements
END;
```

Raising Exceptions

Exceptions are raised by the database server automatically whenever there is any internal database error, but exceptions can be raised explicitly by the programmer by using the command **RAISE**. Following is the simple syntax for raising an exception –

```
DECLARE
   exception_name EXCEPTION;
BEGIN
   IF condition THEN
      RAISE exception_name;
   END IF;
EXCEPTION
   WHEN exception_name THEN
   statement;
END;
```

User-defined Exceptions

PL/SQL allows you to define your own exceptions according to the need of your program. A user-defined exception must be declared and then raised explicitly, using either a RAISE statement or the procedure **DBMS_STANDARD.RAISE_APPLICATION_ERROR**.

The syntax for declaring an exception is –

```
DECLARE
   my-exception EXCEPTION;
```

Pre-defined Exceptions

PL/SQL provides many pre-defined exceptions, which are executed when any database rule is violated by a program. For example, the predefined exception NO_DATA_FOUND is raised when a SELECT INTO statement returns no rows. The following table lists few of the important pre-defined exceptions –

Exception	Oracle Error	SQLCODE	Description
ACCESS_INTO_NULL	06530	-6530	It is raised when a null object is automatically assigned a value.
CASE_NOT_FOUND	06592	-6592	It is raised when none of the choices in the WHEN clause of a CASE statement is selected, and there is no ELSE clause.
COLLECTION_IS_NULL	06531	-6531	It is raised when a program attempts to apply collection methods other than EXISTS to an uninitialized nested table or varray, or the program attempts to assign values to the elements of an uninitialized nested table or varray.
DUP_VAL_ON_INDEX	00001	-1	It is raised when duplicate values are attempted to be stored in a column with unique index.
INVALID_CURSOR	01001	-1001	It is raised when attempts are made to make a cursor operation that is not allowed, such as closing an unopened cursor.
INVALID_NUMBER	01722	-1722	It is raised when the conversion of a character string into a number fails because the string does not represent a valid number.
LOGIN_DENIED	01017	-1017	It is raised when a program attempts to log on to the database with an invalid username or password.

NO_DATA_FOUND	01403	+100	It is raised when a SELECT INTO statement returns no rows.
NOT_LOGGED_ON	01012	-1012	It is raised when a database call is issued without being connected to the database.
PROGRAM_ERROR	06501	-6501	It is raised when PL/SQL has an internal problem.
ROWTYPE_MISMATCH	06504	-6504	It is raised when a cursor fetches value in a variable having incompatible data type.
SELF_IS_NULL	30625	-30625	It is raised when a member method is invoked, but the instance of the object type was not initialized.
STORAGE_ERROR	06500	-6500	It is raised when PL/SQL ran out of memory or memory was corrupted.
TOO_MANY_ROWS	01422	-1422	It is raised when a SELECT INTO statement returns more than one row.
VALUE_ERROR	06502	-6502	It is raised when an arithmetic, conversion, truncation, or sizeconstraint error occurs.
ZERO_DIVIDE	01476	1476	It is raised when an attempt is made to divide a number by zero.

EXPERIMENT 8
TRIGGERS IN PL/SQL

Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events:

- A **database manipulation (DML)** statement (DELETE, INSERT, or UPDATE)
- A **database definition (DDL)** statement (CREATE, ALTER, or DROP).
- A **database operation** (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Triggers can be defined on the table, view, schema, or database with which the event is associated.

Benefits of Triggers

Triggers can be written for the following purposes –

- Generating some derived column values automatically
- Enforcing referential integrity
- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables
- Imposing security authorizations
- Preventing invalid transactions

Creating Triggers

The syntax for creating a trigger is –

```
CREATE [OR REPLACE ] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF }
{INSERT [OR] | UPDATE [OR] | DELETE}
[OF col_name]
ON table_name
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
WHEN (condition)
DECLARE
   Declaration-statements
BEGIN
   Executable-statements
EXCEPTION
   Exception-handling-statements
END;
```

Where,

- CREATE [OR REPLACE] TRIGGER trigger_name – Creates or replaces an existing trigger with the *trigger_name*.
- {BEFORE | AFTER | INSTEAD OF} – This specifies when the trigger will be executed. The INSTEAD OF clause is used for creating trigger on a view.
- {INSERT [OR] | UPDATE [OR] | DELETE} – This specifies the DML operation.
- [OF col_name] – This specifies the column name that will be updated.

- [ON table_name] – This specifies the name of the table associated with the trigger.
- [REFERENCING OLD AS o NEW AS n] – This allows you to refer new and old values for various DML statements, such as INSERT, UPDATE, and DELETE.
- [FOR EACH ROW] – This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
- WHEN (condition) – This provides a condition for rows for which the trigger would fire. This clause is valid only for row-level triggers.

PRACTICAL - 10
TRANSACTIONS

A database transaction is an atomic unit of work that may consist of one or more related SQL statements. It is called atomic because the database modifications brought about by the SQL statements that constitute a transaction can collectively be either committed (made permanent to the database) or rolled back (undone) from the database.

A successfully executed SQL statement and a committed transaction are not same. Even if an SQL statement is executed successfully, unless the transaction containing the statement is committed, it can be rolled back and all changes made by the statement(s) can be undone.

Starting and Ending a Transaction:
A transaction has a beginning and an end. A transaction starts when one of the following events take place:

- The first SQL statement is performed after connecting to the database.
- At each new SQL statement issued after a transaction is completed.

A transaction ends when one of the following events take place –

- A COMMIT or a ROLLBACK statement is issued.
- A DDL statement, such as CREATE TABLE statement, is issued; because in that case a COMMIT is automatically performed.
- A DCL statement, such as a GRANT statement, is issued; because in that case a COMMIT is automatically performed.
- User disconnects from the database.
- User exits from SQL*PLUS by issuing the EXIT command, a COMMIT is automatically performed.
- SQL*Plus terminates abnormally, a ROLLBACK is automatically performed.
- A DML statement fails; in that case a ROLLBACK is automatically performed for undoing that DML statement.

Committing a Transaction:

A transaction is made permanent by issuing the SQL command COMMIT. The general syntax for the COMMIT command is –

```
COMMIT;
```

For example,

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1, 'Ramesh', 32, 'Ahmedabad', 2000.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (2, 'Khilan', 25, 'Delhi', 1500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (3, 'kaushik', 23, 'Kota', 2000.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (4, 'Chaitali', 25, 'Mumbai', 6500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (5, 'Hardik', 27, 'Bhopal', 8500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (6, 'Komal', 22, 'MP', 4500.00 );

COMMIT;
```

Rolling Back Transactions:

Changes made to the database without COMMIT could be undone using the ROLLBACK command.

The general syntax for the ROLLBACK command is –

```
ROLLBACK [TO SAVEPOINT < savepoint_name>];
```

When a transaction is aborted due to some unprecedented situation, like system failure, the entire transaction since a commit is automatically rolled back. If we are not using savepoint, then simply use the following statement to rollback all the changes –

```
ROLLBACK;
```

Save-points:

Save-points are sort of markers that help in splitting a long transaction into smaller units by setting some checkpoints. By setting save-points within a long transaction, we can roll back to a checkpoint if required. This is done by issuing the SAVEPOINT command.

The general syntax for the SAVEPOINT command is –

```
SAVEPOINT < savepoint_name >;
```

For example,

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (7, 'Rajnish', 27, 'HP', 9500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (8, 'Riddhi', 21, 'WB', 4500.00 );
SAVEPOINT sav1;

UPDATE CUSTOMERS
SET SALARY = SALARY + 1000;
ROLLBACK TO sav1;

UPDATE CUSTOMERS
SET SALARY = SALARY + 1000
WHERE ID = 7;
UPDATE CUSTOMERS
SET SALARY = SALARY + 1000
WHERE ID = 8;

COMMIT;
```

ROLLBACK TO sav1 – This statement rolls back all the changes up to the point, where we had marked savepoint sav1.