Unit 2 SQL Commands

****

Introduction of SQL

SQL (Structure Query Language) was developed at IBM by Donald D. Chamberlin and Raymond F. Boyce in the early 1970s. This version initially called SEQUEL (Structured English Query Language). It was designed for manipulate and retrieve data from database.

- It is a programming language which stores, manipulates and retrieves the stored data.
- SQL syntax is not case sensitive.
- It allows user to describe the data.
- It allows user to create and drop database and table.
- It allows user to create view, stored procedure, function in a database.
- It is very easy to provide permissions on tables, procedures, and views hence SQL give security to your data.

Characteristics of SQL

- SQL provide Scalability and Flexibility.
- SQL uses a free form syntax that gives the ability to user to structure the SQL statements in a best suited way.
- It is a high level language.
- It receives natural extensions to its functional capabilities.
- It can execute queries against the database.

Advantages of SQL

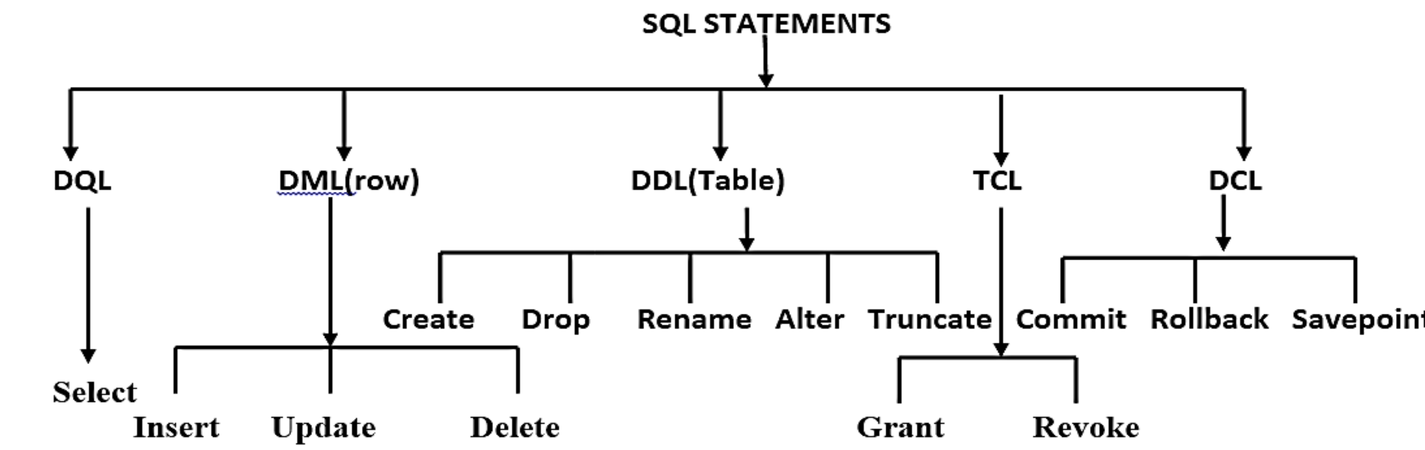
- SQL has well-defined standards.
- SQL is very simple and easy to learn.
- In SQL we can create multiple views
- SQL queries are portable
- It is an Interactive Language
- It retrieves quickly and efficiently huge amount of records from a database.
- No coding required while using standard SQL.

Rules for SQL

1. SQL starts with a verb (i.e. a SQL action word). Example: SELECT statements. This verb may have additional adjectives. Example: FROM
2. Each verb is followed by number of clauses. Example: FROM, WHERE, HAVING

3. A space separates clauses. Example: DROP TABLE EMP;
4. A comma (,) separates parameters without a clause
5. SQL statement is terminated with a semi colon.
6. Reserved words cannot be used as identifiers unless enclosed with double quotes.
7. Identifiers can contain up to 30 characters and must start With an alphabetic
8. '10d' Character and date '01-may-2001' literals must be enclosed within single quotes

Classification of SQL commands



DDL: Data Definition Language

All DDL commands are auto-committed. It saves all the changes permanently in database.

- DDL statements are used used for creating, modifying, and dropping the structure of database objects like table, view, database etc.
- It also defines indexes(keys), specifies links between tables, and forces constraints between tables.

Command	Description
create	Creates a database object like new table, a view of a table etc.
alter	Alters or modifies structure of an existing database object like table, view etc.
truncate	remove all records from a table, including all spaces allocated for the records
drop	Deletes an entire database object like table, view etc.
rename	Rename a database object like table, view etc.

DML: Data Manipulation Language

DML statements used for storing, retrieving, modifying, & deleting data within the database. DML commands are not auto-committed. Changes are not permanent to database, they can be rolled back.

Command	Description
insert	insert data (new row) into a table

update	updates existing data(row) within a table
delete	deletes all records from a table, the space for the records remain

◆ TCL: Transaction Control Language

TCL statements are used to manage the changes made by DML statements. It allows
statements to be grouped together into logical transactions. These commands can cancel
changes made by other commands by rolling back to original state. It can also make changes permanent.

Command	Description
commit	to permanently save
rollback	to undo change
savepoint	to save temporarily

◆ DCL : Data Control Language

DCL statements control access of data and database. It assign permission to user for use database object. It can also take back (revert) the permission. It allows central administration of user privileges and access.

Command	Description
grant	gives user's access privileges to database
revoke	withdraw access rights given with GRANT command (take bac permission)

◆ DQL: Data Query Language

DQL allows getting data from the database. When SELECT is fired against a table or tables, the result is compiled into a temporary table.

Command	Description
select	retrieve records from one or more table

◆ DDL: Data Definition Language

DDL (Data Definition Language) is the subset of SQL commands used to modify, create
or remove database objects.

⌖ CREATE Statement

SQL CREATE TABLE statement is used for define the structure of a table. In a table structure you should define various fields, their data types and constraints. Each column uniquely defined in the table. Each column has a minimum of three attributes, a name, data type and size (i.e. column width). Each column definition is separated from other column definition by a comma (,).

Rules for Creating Tables

1. A name can have maximum up to 30 characters
2. Alphabets from A-Z, a-z and numbers from 0-9 are allowed
3. Name should start with an alphabet
4. A special characters is allowed (Special characters like _, $, # etc.).
5. SQL word is reserved and you cannot use. For example: create, select, and so on.

Syntax	**Example**

CREATE TABLE <TableName>
 	
(<ColumnName1> <DataType><size>,
 	
<ColumnName2> <DataType><size>……..,
 	
<ColumnNameN> <DataType><size>);
	create table emp
	
	(eno number(3),
	
	ename varchar2(15),
	
	eadd varchar2(20));

Creating a Table from Previously Created Table

Syntax: create table <tablename1> <columnname1>,< columnname2> ...
<columnname n>) as select <columnname1>,<columnname2>...<columnnamen>
from <tablename2> where <condition>;

If the 'TableName2' is having records then the target table 'TableName1' will be occupied
with the same records.

Example

1. create table emp1 as select * from emp;
2. To Change column name at table creation time. For example-CURBAL to BALANCE.

create table emp2 (eno, enmae, eaddress) as select eno,enmae,eadd from emp;

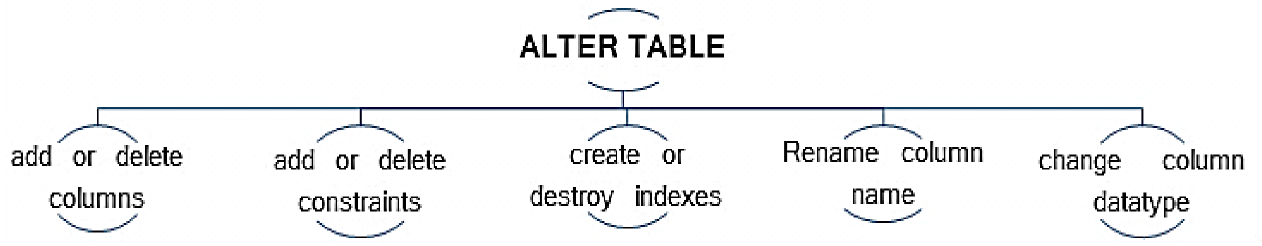
➢ **Create Table without Records (AS SELECT statement)**

To create a new table without records from the existing table (i.e. create structure only). AS SELECT statement used with WHERE clause condition that cannot be satisfied.(i.e. It will just retrieve the table structure not records. Thus, new table will be created empty.)

Syntax	**Example**
create table <new table name> as select * from <old tablename> where 1=2;	create table emp3(eno, enmae, eaddress) as select eno, enmae, eadd from emp where eno=1;

Alter Table (Modifying The Structure Of Table)

- The ALTER TABLE statement is used to add, modify or delete columns in an existing table. It is also used to rename a table.
- You can also add and drop various constraints on an existing table.



Restrictions on the ALTER table

1) User cannot change the column name
2) User cannot decrease the size of column if record in the column.

➢ **Adding New Columns**

- If you want to single add columns in table

Syntax:
```
alter table table_name add (column_name  <datatype>(size)
);
```
Example: `alter table emp add (city varchar2(20));`

- If you want to add multiple columns in table

Syntax:
```
alter table table_name   add (column1 column_definition
, column2 column_definition,………….column_n column-
definition);
```
Example: `alter table emp add (city varchar2 (20),gender char(6));`

➢ **Dropping a Column:** remove column from the table.

Syntax: `alter table <tablename> drop column <columnname>;`

Example: `alter table emp drop column city;`

➢ **Modifying Existing Columns**

- If you want to modify an existing column in table

Syntax: `alter table table_name modify (<columnname> <newdatatype> <size>);`

Example: `alter table emp modify (name char(40));`

- If you want to modify multiple columns in table

Syntax:
```
alter table table_name modify (<columnname1>
<newdatatype><size>, <columnname2> <newdatatype><size>
... <columnname n><newdatatype> <size>);
```
Example: `alter table emp1 modify (ename char(20), eadd varchar2(20));`

➢ **RENAME Existing Column:** Alter table rename column

Syntax: `alter table table_name rename column <old column_name> to <new column_name>;`

Example: `alter table emp rename column eadd to empaddress;`

➢ **Alter Table To Add / Remove Constraints**

- To add a primary key/Composite Primary Key constraint after creation of a table

Syntax: alter table <tablename> add < constraints name> (columns);
Example: alter table emp add primary key (eno);
    alter table emp add primary key (id, name); //Composite Primary Key
    alter table emp modify salary decimal(18, 2) not null;

- To add a foreign key constraint after creation of a table

  Example: alter table emp add foreign key (deptno) references dept (deptno);

- To remove primary key constraint

  Syntax: alter table <tablename> drop < constraints name>;
  Example: alter table emp drop primary key;
  It will remove primary key constraint from a table.

- To add user defined constraint

  Example: alter table emp add constraint myuniqconstraint unique(age);

- To remove user defined constraint

  Syntax: alter table <tablename> drop constraints <const_name>
  Example: alter table emp drop constraint myuniqconstraint;
  It will remove the constraint, which was defined on the table.

DROP TABLE (Drop a table from the database)

Drop command is used to remove any object such as table, index, view, package and
function, all data, indexes, triggers, constraints and permission specifications. If a table is dropped, all records within table are lost and cannot be recovered. Use very careful this
command.

Syntax: drop <object> <object_name>; or drop database <Database_Name>;
Example: drop table emp; or drop view vw_empdept;

Note: 1. DROP statement removes the table.
2. It deletes all its data.
3. It deletes constraints and indexes associated with it.
4. Any views and synonyms are remain but are invalid.
5. Any pending Transaction is committed.
6. Only the creator of the table as user with the DROP ANY TABLE privilege can
remove a table.

TRUNCATE Statement

It is used to delete the entire table data permanently and free the containing space. But table structure will be remain present but records cannot be ROLLBACK.

Syntax: truncate table <table_name>;
Example: truncate table emp;

➢ **Difference b/w DROP and TRUNCATE statements**

When you drop a table:

- Table structure will be dropped
- Relationship and Integrity constraints will be dropped
- Access privileges will also be dropped

When we TRUNCATE a table, the table structure remains the same.

- Truncate operations drop and recreate the table which much faster than deleting rows one by one.
- It is used to delete all the rows from the table and free the containing space.

RENAME Statement: It is used to change the name of a table.

Syntax: rename <old_tablename> to <new_tablename>;
OR
alter table <old_tablename> rename to <new_tablename>;
Example: rename emp to employee; or alter table emp rename to worker;

DESC (View Structure of table) : It used to view structure of a table.

Syntax: describe <Table_Name> OR desc <Table_Name >
Example: desc emp;

➢ **Difference between DROP, DELETE and TRUNCATE**

Drop	Delete	Truncate
1. Whole structure and data will drop. 2.can't Rollback 3.can't use where clause 4.All data and structure drop simultaneously	1. Structure will not drop, data will drop. 2.can Rollback 3.can use WHERE clause 4.Row by Row data Delete	1. Structure will not drop, all data will drop. 2.can't Rollback 3.can't use WHERE clause 4. Truncate operation drop and recreate the table.

DML: Data Manipulation Language

INSERT STATEMENT

It is used to insert a single or a multiple records (rows) in the table. When inserting a single row of data into the table, the insert operation:

- Creates a new empty row in the database table then
- Insert statement loads passed values into the specified columns.

In INSERT INTO statement, table columns and values have one to one relationship. i.e.
First value inserted into first column, second value inserted into second column and so on.

```
Syntax: insert into <Table_Name>
      <Columnname1>,<Columnname2>…..<ColumnnameN> values
      (<Value1>, <Value2>…………<Value N>);
Example:  insert into emp (eno, ename,
eadd) values (1, 'abhi',' pune');
```

You can specify or ignore the column names while using INSERT INTO statement.

<u>We have two methods to insert.</u>

- Ø By value method:
  1. By specifying column names
  2. Without specifying column names
- Ø By address method

➢ **Using Value Method:**

- <u>Inserting Data Into Specified Columns Using Value Method</u>

```
Syntax: insert into <table_name> (col1,col2…,Coln)
VALUES(value1,value2…,Value n);
Example: insert into student (eno, ename) values (5,
'ramesh');
```

- <u>Inserting Data Into Without specifying column names Using Value Method</u>

```
Syntax:
  insert into <table_name> values (value1, value2, valu
e 3, .... value n);
Example:    insert into emp values (1, 'nayan',
'sangola');
          insert into emp values (2, 'tara', 'solapur');
```

To insert a new record again you have to type entire insert command, if there is lot of records this will be difficult. This will be avoided by using address method.

➢ **Using Address Method**

```
  Syntax:    insert into <table_name) values (&col1,
&col2, &col3….., &coln);
```

To use forward slash (/) for repeated query execution.

```
Example:  insert into emp values (&eno, '&ename', &eadd);
          Enter value for eno: 3
          Enter value for ename: jagan
          Enter value for eadd: pune
          Old 1: insert into emp values (&eno, '&ename',
          &eadd)
          New 1: insert into emp values (3, 'jagan',
          'pune')
          sql> /
          Enter value for eno: 4
          Enter value for ename: ram
          Enter value for eadd: mumbai
          Old 1: insert into emp values (&eno, '&ename',
          &eadd)
          New 1: insert into emp values (4, 'ram',
          'mumbai')
```

➢ **Inserting Data Into A Table From Another Table**

Syntax: insert into <table_name> select <col_name, col_name> from <table_name> [where <condition>];
Example: insert into emp2 select * from emp1;

⌖ **SELECT (Retrieving Data Using Query)**

SELECT Statement used to fetch data from database table which returns data in the result
sets of table. SELECT command is used to retrieve rows selected from one or more tables.

➢ **All Rows and All Columns**

Syntax: select * from <Table_Name>; **Example:** select * from emp;

It will display data with all rows and columns from the table.

➢ **Filtering Table Data**

When we reterives specific rows and columns data from a table. SQL provides a method of filtering table data with following ways:

1. Selected columns and all rows
2. Selected rows and all columns
3. Selected columns and selected rows

1. Selected Columns and All Rows

It will display all records with specific fields that user has specified with SELECT command.

Syntax: select <col_name1>,<col_name2>,…..<col_name n> from < tablename >;
Example: select eno, ename from emp;

2. Selected Rows and All Columns

Syntax: select * from <tablename> where <condition>;
Example: select * from emp where eno=10;

Oracle engine displays only those records that satisfy the specified condition after WHERE caluse.

➢ **Where clause** applies a filter on the retrieved rows. When a WHERE clause is added
to the SQL query, the Oracle engine compares each record in the table with specified
condition. All operators such as logical, arithmetic, etc. can be used in the condition of WHERE clause.

3. Selected Columns and Selected Rows

Syntax: select <col_name1>, <col_name2>,….<col_namen> from <table_name> where <condition> ;
Example: select eno, ename from emp where eno=10;
It will display all records with specific fields after specified condition WHERE will be true.

➢ **ELIMINATING DUPLICATE ROWS**

In a table, there may be a chance to exist a duplicate value. DISTINCT command is used to retrieve only unique data. SELECT UNIQUE & SELECT DISTINCT statements are same.

Syntax: Select distinct <Col_Name1>,<Col_Name2>,…<Col_Namen> From <Tablename>;
OR select distinct * from <tablename>;

Example: select distinct eno, ename from emp; **OR** select distinct * from emp;

- **Distinct clause-** allows removing duplicates data from the result set. This clause only used with SELECT statement. It scans all the values of the specified columns and display only unique values. It eliminates rows that have exactly same contents in each column.
- **Arithmetic calculation**

Example: Select employee name, salary and compute salary * 0.05 for each row retrieved:

```
        Select ename, sal, sal * 0.05 "Salary" from emp;
```

Here the default output column sal*0.05 is renamed with Salary.

DELETE (Deleting Data)

- DELETE Statement is used to delete one or more rows from a table.
- DELETE Query use following two way,
  - Remove all rows
  - Remove only specific rows
- DELETE statement does not free containing space by the table.

Syntax: delete from <table_name> where <condition>;

- If condition is not specified then it will be delete all rows.

Example: delete from emp; // it will delete all the records from a emp table.

- If condition is specified then it will delete all rows that satisfy the specified condition.

Example: delete from emp where eno>10;
It will delete all the records for which ENO is greater than 10.

UPDATE (Updating a Contents of Table)

The UPDATE statement changes in existing rows data either adding new data or modifying existing data. User can update

- all rows from table (update all table rows)
- selected rows from table (update only specific data/rows)

Syntax: update <table_name> set <columnname1>=<expression1>, <columnname2>= <expression2>……, <columnnamen>=<expressionn> where <condtion>;

OR

update <table_name> set <column_name=expression> where <conditions>;

Example: update emp set city='sangola';

- **All rows from table:** If condition is not specified then it will be update entire table rows. It can update multiple fields at the same time.

```
      update emp set salary=500;
```

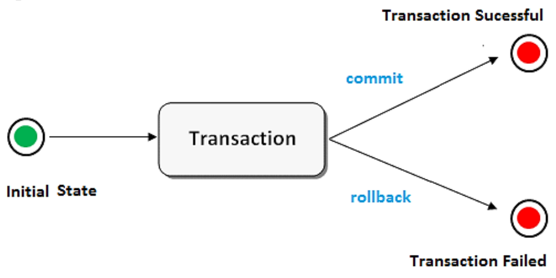
```
     update emp set address='pune', salary=1000;
```

- **Selected rows from table:** If condition is specified then it will be update only specific table rows. It can update single fields at the same time.

```
     update emp set city='sangola' where eno=10;
```

TCL: Transaction Control Language

TCL commands are used to manage transactions in database. TCL Statement manage the change which is made in the database by a DML statement will be called as a transaction. It also allows statements to be grouped together into logical transactions.



Changes will not reside permanently in the database, without auto commit option enabled
in the session. Those changes will reside in temporary memory. So, we need to handle
these changes. TCL statements are:

- COMMIT
- ROLLBACK
- SAVEPOINT

COMMIT (Permanently save any transaction into database):

COMMIT command is used to save changes invoked by a transaction in the database. COMMIT command saves all transactions in the database since the last COMMIT or ROLLBACK command. TCL are used with the DML commands INSERT, UPDATE and DELETE only. Commit cannot be used while creating or dropping tables because these operations are automatically committed in the database.

Syntax: commit;

If Auto commit option for session is enabled, then this is not required as commit takes place automatically. DDL statements will run implicit commit.

Rollback command (restores the database to last committed state)

Rollback will be used to reject the changes made by DML statements in the database. If auto commit option enabled in the session, then there is no use Rollback.

If we give Rollback then any pending changes in the database which are made by DML statements will be rejected. It is also use with savepoint command to jump to a savepoint in a transaction.

Syntax: Rollback; OR rollback to savepoint-name;

- **Savepoint command**

Savepoint command is used to temporarily save a transaction so that you can rollback to that point whenever necessary.

Syntax: savepoint savepoint-name;

Example: Savepoint and Rollback on **class table**,

ID	NAME
1	abhi
2	adam
3	alex

```
 SQL> insert into class values
(4,'Rahul');
SQL> Commit;
SQL> update class set name='abhijit'
where id='5';
SQL> savepoint A;
SQL> insert into class values
(6,'Chris');
SQL> savepoint B;
SQL> insert into class values
(7,'Bravo');
SQL> savepoint C;
SQL> select * from class;
```

ID	NAME
1	abhi
2	adam
3	alex
4	rahul
5	abhijit
6	chris
7	bravo

```
Now rollback to savepoint B
SQL> Rollback to B;
SQL> SELECT * from class;
```

ID	NAME
1	abhi
2	adam
4	alex
5	abhijit
6	chris

Now rollback to savepoint A

```
SQL> Rollback to A;
SQL> SELECT * from class;
```

ID	NAME
1	abhi
2	adam
4	alex
5	abhijit

❖ DCL : Data Control Language

DCL commands are used to enforce database security in a multiple user database environment. DBA controls the access of users to the database. Only Database Administrator's or owners of the database object can provide / remove privileges on a database object. **DCL statements: GRANT and REVOKE**

1) **GRANT:** GRANT is a command used to provide access or privileges on the database objects to the users. We can give any type of permissions to the users on the database using grant command.

Syntax: grant <privilege_name> on <object_name> to {user_name|public|role_name}
        [with grant option];

privilege_name	It is the access right or privilege granted to the user. Some of the access rights are ALL, EXECUTE, and SELECT.
object_name	It is the name of a database object like TABLE, VIEW, STORED PROCEDURE and SEQUENCE.
user_name	It is the name of the user to whom an access right is being granted.
PUBLIC	It is used to grant access rights to all users.
ROLES	They are a set of privileges grouped together.
with grant option	It allows a user to grant access rights to other users.

Example: grant select on emp to bca;

This command grants a SELECT permission on employee table to user1.

- <u>if we want to give Select permission on all the database tables to a user</u>

  Syntax: grant select any table to <user_name>;

  Example: grant select any table to bca;

- <u>If we want to give select permission on specific table then</u>

  Syntax: GRANT permission ON tab_name TO user_name;

  Example: grant select on emp to bca;

- <u>If we want to give Create table permission then,</u>

  Syntax: grant create table to <user_name>;

  Example: grant create table to bca;

  we can give any privilege to the users on the database.

2) **REVOKE:** Revoke will be used to remove previously granted access permissions to the users on the database.

Syntax: revoke <privilege_name> on <object_name> from {user_name|PUBLIC|role_name};

Example: revoke select on emp from bca;

- ➢ if we want to give Select permission on all the database tables to a user,

  Syntax: revoke select any table from <user_name>;

  Example: revoke select any table from bca;

- ➢ If we want to give select permission on specific table then

  Syntax: revoke <permission> on <table_name> from <user_name>;

  Example: revoke select on emp from bca;

- ➢ If we want to give Create table permission then,

  Syntax: revoke <permission> from <user_name>;

  Example: revoke create table from bca;

⌖ ROLES

- ▪ A role is a privilege (license) or set of privileges that allows a user to perform certain functions in the database.
- ▪ When there are many users in a database it becomes difficult to grant or revoke privileges to users.
- ▪ If you define roles, you can grant or revoke privileges to users, thereby automatically granting or revoking privileges.

  Syntax: create role role_name; **Example:** create role assistant;

- ➢ After creating role to assign permission to newly create role

  Example: grant create table, create session to assistant;

- ➢ After assigning permission to role then you can that permission to user by using role.

  Example: grant assistant to bca;

- ➢ If WITH ADMIN OPTION is used, that user can then grant roles to other users.

  Syntax: grant <role_name> to <user_name> with admin option;

  Example: grant assistant to bca with admin option;

- ➢ If role is removing then user cannot login because users have permission of the role.

  Example: drop role assistant cascade;

Three type roles:

- Connect
- Resource
- DBA (database administrator)

♦ **Connect Role:** The Connect role is the entry-level role. Connect role access can be granted permission with a database.

Example: grant connect to bca; **OR** revoke connect from bca;

♦ **Resource Role:** The Resource role gives the more database access to user. In addition to the permissions that can be granted to the Connect role, Resource roles can also be granted permission to create procedures, triggers, and indexes.

Example: grant resource to bca; **OR** revoke resource from bca;

- **DBA Role:** The DBA role includes all permission. Users with this role are able to do anything they want to the database system.

Example: grant dba to bca; **OR** revoke dba from bca;

- **Creating Users:** Create a new user in the database.

Syntax:

```
CREATE USER user_name IDENTIFIED {BY password |
EXTERNALLY}
[DEFAULT TABLESPACE tablespace_name]
[TEMPORARY TABLESPACE tablespace_name]
[QUOTA {integer [K|M] | UNLIMITED} ON tablespace]
[PROFILE profile];
```

Example:

```
create user bca identified by sangola
default tablespace system
temporary tablespace temp
quota unlimited on system;
```

OR

```
Create user bca identified by sangola;
```

- To assigning permission to user one or more permission using without role command.

Syntax: grant <permission> to <user_name>;
Example: grant connect to bca;
grant connect, create table, create session to bca;

- To change the user's password:

Example: alter user bca identified by sangola;

- To change the default table space:

Example: alter user bca default tablespace users;

- To remove a user entry from system database.

Syntax: drop user user_name [cascade];
Example: drop user bca cascade; **OR** drop user bca;

If the CASCADE option is used, all objects owned by username are dropped along with
the user's account. If CASCADE is not used then user is not dropped.

- To connect other user and another user table data and also perform DML operation.

create user bca identified by sangola; and create user bcs identified by sangola;

After creating two users then assign permission to both user.

```
create role assistant;
grant create table, create session to assistant;
grant assistant to bca;   and aslo assign role to
grant assistant to bcs;
```

After granting permission then user bca create a table stud

```
create table stud (id int, name varchar2 (10));
```

Insert one record in the table stud

```
     insert into stud values (1,'shyam');
```

Connect to another user **bcs** and use stud table for DML operation but before performing operation to grant DML permission to user **bcs**.

```
     connect bca/sangola
Syntax: grant <permission> on < table name> to
<username>;
     grant select, insert, update, delete on stud to bcs;
     connect bcs/sangola
     insert into bca.stud values (2,'ram');
     select * from bca.stud;
When revoke permission then access is denied for
specified user
     connect bca/sangola
     revoke select on stud from bcs;
```

Integrity Constraints

- Constraints are the rules. These can be specified at the time of table creation or can be added after table creation by using alter table statement.
- Constraints prevent the table from deletion, if there is any dependency

Constraints can be divided into following two types:

1) The constraints can be specified immediately after the column definition. This is called column-level definition.

2) The constraints can be specified after all columns and end of table. This is called table-level definition.

Data Integrity:

- **Entity Integrity** ensures that no duplicate rows in a table. Ex: Unique, Primary Key
- **Domain Integrity** enforces valid entries for a given column by restricting the type, the format, or the range of possible values. Ex: check, Null, Not Null
- **Referential integrity** ensures that rows cannot be deleted, which are used by other records. Ex: Foreign Key
- **User-Defined Integrity** enforces some specific business rules on entity, domain, or referential integrity categories.

Constraints are:

- **Not Null Constraint:** Ensures that a column have not NULL value.
- **Default Constraint:** Provides a default value for a column when none is specified.
- **Unique Constraint:** Ensures that all values in a column are different (unique).
- **Primary Key:** Uniquely identified each rows/records in a database table.

- **Foreign Key:** Uniquely identified a rows/records in any another database table.
- **Check Constraint:** Ensures that all values in a column must satisfy certain conditions.

➢ **Not Null Constraint**

- NOT NULL constraint prevents inserting NULL values into a column. In database,
  NULL means unknown or missing information.
- If you try to insert or update a NULL value in the column, then database engine will
  reject the change and issue an error.
- NOT NULL constraint applied only at column level not table level. You should manually define NOT NULL constraint because table column default set NULL value.
- When a column name is defined as not null, then that column becomes a mandatory column. It force to user for enter data into the column.

Principles of NULL values

- A NULL values is different from a blank or zero.
- A NULL value can be inserted into the columns of any Data type.

Example: To create an emp_info table and enforces "Name" column to not null

```
create table emp_info(no number(3) primary key,name
char(10)not null,
                    address varchar2(30));
desc emp_info;
 Name                             Null?                Type
 ----------------------------    ---------            -----------
------
 NO                                NOT NULL     NUMBER(3)
 NAME                             NOT NULL      CHAR(10)
 ADDRESS                                        VARCHAR2(30)
```

⦿ **<u>Add NOT NULL constraint in existing table column</u>**

```
alter table emp_info modify address varchar2(30) not
null;
desc emp_info;
 Name                             Null?                Type
 ----------------------------    ---------            -----------
----------
 NO                                NOT NULL      NUMBER(3)
 NAME                              NOT NULL      CHAR(10)
 ADDRESS                           NOT NULL      VARCHAR2(30)
```

⦿ **<u>Drop NOT NULL constraint from existing table column</u>**

```
alter table emp_info modify address varchar2(30) null;
desc emp_info;
 Name                             Null?                Type
 ----------------------------    ---------            -----------
-----------------
 NO                                NOT NULL      NUMBER(3)
 NAME                              NOT NULL     CHAR(10)
 ADDRESS                                        VARCHAR2(30)
```

➢ **Unique Key Constraint**

- The UNIQUE constraint is used for uniquely identify each record in a database. It requires that each value in a column should be unique.
- It provides uniqueness guarantee of a column or set of columns. It cannot allow duplicate values but **unique key allow NULL (blank) values.**
- You can assign many unique key constraints in one table.
- UNIQUE constraint can be applied at column level and table level.

unique key at column level	**Unique key at table level**
create table emp(eno number(5), name char(20), **contact_no number(10)unique,** age number(2), salary number(10));	create table emp(eno number(5), name char(20), contact_no number(10) age number(2), salary number(10), **unique (contact_no)**);

```
Now we are inserting record into this table to check
unique key how to work
Example try to insert same name as soon as fire error
for unique constraint violated.
sql> insert into emp(no,name) values(1,'om');
sql> insert into emp(no,name) values(2,'om');
*ERROR line 1:ORA-00001:unique constraint
(SYSTEM.SYS_C004081) violated.
```

⊙ **Defining a unique key constraint on multiple columns:**

Example:

```
create table students (sid int not null, name varchar2(25
) not null,
       city varchar2(25), constraint uniqcon_student
unique (sid, name));
```

⊙ **UNIQUE KEY constraint on ALTER TABLE:**

- Defining a unique key constraint on single column:
  ```
  alter table students add unique (sid);
  ```
- Defining a unique key constraint on multiple columns:
  ```
  alter table students add constraint uniqcon_student
  unique (sid, name);
  ```

➢ **Primary Key Constraints:**

- Primary Key apply on columns for uniquely identifies each record from the table.
- Primary Key constraint column cannot accept duplicate data and NULL values.
- Each table can have only ONE primary key.
- A primary key is assigned one or more columns in a table is called **composite primary key**.
- It defines a mandatory column and data through the column must be unique.
- Primary key constraint create automatic index file for identifying unique records.

- Primary key constraint can be defined at the column and table level.
- Primary key constraint combination of NOT NULL and UNIQUE Constraints

When we specify a primary key constraint for column, database engine automatically creates a unique index file for identify unique records.

at table level (primary key for single columns)	**At (single) column level**
create table stud(sid number(5), sname char(20), age number(2), contact_no number(10),**primary key(sid)**);	create table stud(**sid number(5) primary key**, sname char(20), age number(2),contact_no number(10));

- **Primary key on one column:**

```
alter table stud add primary key (sid);
```

- **Primary key on multiple columns:**

```
alter table stud add constraint pk_studentid primary
key (sid, contact_no);
```

- **<u>DROP a PRIMARY KEY constraint</u>**

```
alter table stud drop primary key;
alter table stud drop constraint pk_studentid;
```

- **<u>Enable or Disable a Primary Key</u>**

```
alter table students disable constraint pk_studentid;

alter table students enable constraint pk_studentid;
```

- **Foreign Key (referential) Constraint**
  - Foreign key represents relationship between one or more tables. FOREIGN KEY apply on columns who references values in another table column.
  - FOREIGN KEY constraints also known as relationship (referential) constraints.
  - A foreign key is a column whose values are derived from the primary key of the other table.
  - FOREIGN KEY constraint applied column must have same data type as they reference on another table column.

Foreign key/references constraints:

- Rejects an INSERT or UPDATE of a value, if a corresponding value does not exist in the primary key table.
- Rejects a DELETE, if it would invalidate a REFERENCES constraint.

Syntax: [CONSTRAINT constraint_name] FOREIGN KEY(column_name) REFERENCES
referenced_table_name(column_name);

- **<u>A FOREIGN KEY constraint specify at column level,</u>**

```
create table emp_info(no number(3) primary key,
   name char(20), address varchar(30),
   contact_no number(12));
create table emp_salary(no number(3) primary key,
   users_no number(3) references emp_info(no),
   salary number(12));
```

⊙ **A FOREIGN KEY constraint specify at table level,**

```
create table emp_info (no number(3) primary key,
    name char(20),address varchar(30),
    contact_no number(12));
create table emp_salary(no number(3) primary key,
    user_no number(3),salary number(12),
     constraint fk_user foreign key (user_no) references
emp_info(no));
```

⊙ **To add foreign key in existing table**

When you adding foreign key constraint, SQL check any existing data violate the foreign key constraint or not. If not violate constraint added successfully otherwise you have to update invalid data to prevent foreign key constraint violating.

```
alter table emp_salary add foreign key (user_no)
references emp_info(no);
```

⊙ **To drop foreign key in existing table**

```
alter table emp_salary drop constraint fk_userno;
```

⊙ **To drop a FOREIGN KEY constraint:**

```
alter table orders drop foreign key;
```

- **On Delete Cascade:** delete the dependent rows from the child table, only when parent table row is deleted.
- **On Delete Set Null:** when you want to convert dependent foreign key values to null.

Primary key	**Foreign key**
Primary key cannot be null	Foreign key can be null.
Primary key is always unique. It uniquely identify a record in a table	Foreign key can be duplicated Because foreign key is a field refer primary key in another table.
only one primary key in the table	More than one foreign key in the table.

➢ **Check Constraints:**

- Check constraint is used to check the value which is entered into a record. It is used to define condition which each row must satisfy.
- If the condition value evaluates to false, then the record violates the constraint and
  you cannot enter it into the table.
- We cannot define check constraint in SQL view, sub queries or sequences.

- It performs check on the values, before storing them into the database. It's like condition checking before saving data into a column.
- The constraint can be applied for a single column or a group of columns.

Syntax: [CONSTRAINT constraint name] CHECK (condition)

⦿ **Define CHECK constraint at column level with other column attributes**

```
create table student_info(stu_code varchar(6) primary
key check(stu_code like 'st%'),
          name varchar(30) check(name=upper(name)),
         city varchar(30) check(city
in('mumbai','pune','delhi','chennai')),
         scholarship number(5) check (scholarship
between 5000 and 20000),
         class varchar(15));
insert into student_info values
('st001','NAYANTARA','pune',8900,'ecs-II');
```

We are creating new **student_info** table name with following check constraints:

1. Values inserted into **stu_code** column must be start with the lower letter **'ST'**.
2. Values inserted into **name** column must be capitalized.
3. Values inserted into **city** column only allow 'Mumbai','Pune','Delhi','Chennai' as valid legal values.
4. Values inserted into scholarship column between 5000 and 20000.

⦿ **CHECK constraint apply in table level.**

```
create table student_info(stu_code varchar(10)
primary key,
         name varchar(30),
         city varchar (30),scholarship number(5),
class varchar(10),
         check (stu_code like 'st%'),
         check (name = upper(name)),
         check (city in
('mumbai','pune','delhi','chennai')),
         check (scholarship between 5000 and 20000));
insert into student_info values
('st002','NAYAN','delhi',9000,'bca-II');
```

⦿ **To add CHECK constraint in existing table column**

```
alter table student_info add constraint check_name
check(class=upper(class));
```

⦿ **To drop CHECK constraint in existing table column**

```
alter table student_info drop constraint check_name;
```

➢ **Default Constraint:**

- The DEFAULT constraint provides a default value to a column when the INSERT statement. You do not provide a specific value.

- The data type of default value should be match with the type of the column.
- DEFAULT constraint specified only at column level.

⦿ **Specify DEFAULT constraint at column level with other column**

```
create table stud(roll_no number(3) primary key,
   stud_name varchar(30) not null, class varchar(15)
default 'M.Sc.(CS)',
   fees_pay number(5));
```

⦿ **To add DEFAULT constraint in existing table column**

```
   alter table stu_info modify fees_pay number(5)
default 15000;
```

⦿ **To drop DEFAULT constraint in existing table column**

You need to just redefine (or modify) column attribute.

```
   alter table stu_info modify fees_pay number(5);
```

SQL operators and clauses

The symbols which are used to perform logical and mathematical operations in SQL are called SQL operators. SQL reserved words or characters are known as operators.
Three types of operators in SQL:

1. SQL Arithmetic Operators
2. SQL Comparison Operators
3. SQL Logical Operators

➢ SQL Arithmetic Operators:

Arithmetic operators in SQL are used to perform mathematical calculations like addition, subtraction, multiplication, division and modulus in SQL statements.

DUAL table: It is a system defined table which contains only one column to perform calculations on users own data.

Arithmetic Operators	**Example**
+ (Addition)	Select 10+2 from dual;
- (Subtraction)	Select 10-2 from dual;
* (multiplication)	Select 10*2 from dual;
/ (Division)	Select 10/2 from dual;
% (Modulus)	Select 10%2 from dual;

```
    select (90000*10)/100 "10% of 90000" from dual;
    select sal " Basic Sal", (0.02*sal) " TA" from emp;
```

- **SQL Comparison Operators:** used to compare values by specifying conditions on the columns.

Operators	**Example/Description**
>	x > y (x is greater than y)
<	x < y (x is less than y)
>=	x >= y (x is greater than or equal to y)
<=	x <= y (x is less than or equal to y)
=	x == y (x is equal to y)
!= or < >	x != y or x <> y (x is not equal to y)
!<	x !< y (x is not less than y)
!>	x !> y (x is not greater than y)

- **SQL Logical Operators:**

Logical operators in SQL are used to perform logical operations on the given expressions in SQL statements. There are many operators in SQL which are used in SQL statements in the WHERE clause. They are

Operator	**Description**
AND	Display output if all conditions are true. If any one condition was failed then it will not display output.
OR	Display output if anyone condition is true. If all conditions are false then it will not display output.
ALL	The **ALL** operator returns true when value matches with all values in single column set of values. It's like **AND operator** it will compare value against with all values in column.
ANY	The **Any** operator in SQL returns true when value matches with any value in single column set of values. It's like **OR operator** it will compare value against with any value in column.

LIKE	The **LIKE** operator is used to search for character string with specified pattern using wildcards in column.
IN	The **IN** operator is used to search for specified value matches any value in set of multiple values.
BETWEEN	The **BETWEEN** operator is used to get values within range.
EXISTS	The **EXISTS** operator is used to show result if subquery returns data.
NOT	The **NOT** operator is a negate operator that means it will show data for opposite of conditions what we mentioned in SQL statement.
SOME	The **SOME** operator is used to compare value with a single column set of values returned by subquery. **SOME** must match at least one value in subquery and that value must be preceded by comparison operators.

➢ **USING =, >, <, >=, <=, !=, <>**

Example:
```
select * from student where no = 2;
select * from student where no < 4;
select * from student where no > 3;
select * from student where no <= 7;
select * from student where no >= 3;
select * from student where no != 2;
select * from student where no <> 2;
```

➢ **AND Operator:** returns output when all the conditions become true.

Syntax: select * from <table_name> where <condition1> and <condi2>and<condition n>;

Example: select * from student where no=2 **and** marks>=200;

➢ **OR Operator:** returns output when either one of the conditions becomes true.

Syntax: select * from <table_name> where <condition1>and<condition2> or <conditionn>;

Example: select * from student where no=2 **or** marks>=200;

➢ **NOT** will negate the condition. It will select the records where condition is not satisfied.

➢ **Range Searching:** Oracle provides 3 types for searching data in a range.

1. BETWEEN
2. IN
3. IS

1) Between Operator

- BETWEEN operator is used to select data that is within a range of values.
- BETWEEN operator that contain specified lower and upper limit values.
- Two values in the range must be linked with keyword

AND.

- BETWEEN operators can be used with numeric and character data types.

Syntax: select * from <table_name> where <col> between <lower bound> and
<upper bound>;
Example: select * from student where marks between 200 **and** 400;
select empno, ename, deptno from emp where deptno>=15 **and** deptno<=25;

♦ **Not Between Operator**: used for to select data which is not within the range.

Syntax: select * from <table_name> where <col> not between <lower bound> and
<upper bound>;
Example: select * from student where marks not between 200 and 400;
select * from emp where empno **not between** 10 and 20;

2) In Operator

- If a value is required to be compared with list of values then IN predicate is used.
- IN operator reduces the need to use multiple OR conditions.
- Arithmetic operator (=) compares a single value to another single value.

Syntax: select * from <table_name> where <col> in (value1, value2, value3…valuen);
Example: select * from student where no **in** (1, 2, 3);
select empno, ename, deptno from emp where deptno 10 **or** deptno=20;
select empno, ename, deptno from empwhere deptno **in (20,30)**;

♦ **Not In Operator**: Not In operator is usd to select all the rows where value does not match in the list of values specified

Syntax: select * from <table_name> where <col> not in(value1, value2,value3…valuen);
Example: select * from student where no **not in** (1, 2, 3);
select empno, ename, deptno from emp where deptno **not in** (20,30);

➢ **Is Null Operator**: used to check the column value is null or not, if it is null display output

Syntax: select * from <table_name> where <col> is null;
Example: select * from student where marks **is null**;

1) If expression is a NULL value, the condition evaluates to TRUE.
2) If expression is not a NULL value, the condition evaluates to FALSE.

➢ **Is Not Null Operator**: used to test for NOT NULL value in the specified column.

Syntax: select * from <table_name> where <col> is not null;
Example: select * from student where marks **is not null;**

➢ **Like Operator**

Like operator is used to retrieve data based on specify pattern or characters. Like operator selects those rows only containing fields that match specified portions of character strings. LIKE is used with char, varchar, text, date time and small date time data. A wildcard allows the user to match fields that contain certain letters.

Syntax: select * from <table_name> where <col> like <pattern>;

%	It matches zero or more any characters.
_	Any single character search with the specified pattern
[ ]	Any single character within the specified range e.g.([a-f]) or set [abcdef])

Example:

```
Create table customer(
cno number,
cname varchar(10),
balance number(6)
city varchar(10));
```

Cno	Cname	Balance	City
1	Arjun	12550	Pandharpur
2	Ramesh	16550	Solapur
3	Jagan	18000	Sangola
4	Madhu	19000	Kolhapur
5	Varun	15250	Sangli
6	Ritu	14850	Satara

Find customer names whose salary is 15000.

```
select * from customer where salary like 15000;
```

Find customer names whose city name beginning start with 'Sa'.

```
select cname from customer where city like 'Sa%';
```

Find customer names ending with 'un'.

```
select * from customer where cname like '%un';
```

Find customer name whose name's second letter start with 'a'.

```
select * from customer where cname like '_a%';
```

Display customer details whose balance's third number start with '5'.

```
select * from customer where balance like '__5%';
```

Display customer details whose name's third letter start with 'h' from ending.

```
select * from customer where cname like '%_t%';
```

Find customer names whose city name start with "S" and end with "a"

```
Select cname from customer where city like 'S%a';
```

Find customer names whose name contains 2 a's.

```
select * from customer where cname like '%a%a%';
```

Find customer name whose city name have "ang" in any position

```
select cname from customer where city like '%ang%';
```

- **ORDER BY Clause**

The SQL ORDER BY clause is used for sorting data in ascending and descending order based on one or more columns. Some databases sort query results in ascending order by default.

Syntax:

```
select expressions from tables where conditions order by
expression [asc | desc];
Sort the result in ascending order by name and salary.
     select * from emp order by ename, salary;
To sort the result in descending order by name.
     select * from emp order by name desc;
```

- **GROUP BY Clause**
  - GROUP BY Statement is used to to organize similar data into groups with the help of some functions. i.e. if a particular column has same values in different rows then it will arrange these rows in a group.
  - GROUP BY returns only one result per group of data
  - A query containing a group by clause is processed in the following way:
    1. Select all rows that satisfy the condition specified in the where clause.
    2. Discard all groups that do not satisfy the condition in the having clause.
    3. Apply aggregate functions to each group.

Syntax: **select** column1, **aggregate function_name**(column2) **from** table_name **where** condition **group by** column1, column2;

1. **Aggregate function_name**: Name of the function. Example- SUM(),AVG().
2. **table_name**: Name of the table.
3. **condition**: Condition used.

Example: select dno, sum(sal) from emp group by dno;

Table Store_Information

Store_Name	**Product_ID**	**Sales**	**Txn_Date**
Sangola	1	1500	Jan-05-1999
Sangola	2	500	Jan-05-1999
Pune	1	250	Jan-07-1999
Sangola	1	300	Jan-08-1999
Sangli	1	700	Jan-08-1999

⊙ **GROUP BY a single column** : To find total sales for each store:

```
select store_name, sum(sales) from store_information
group by store_name;
```

Store Name	**SUM(Sales)**
Sangola	2300
Pune	250
Sangli	700

⊙ **GROUP BY multiple columns:** To have two or more columns associated with GROUP BY.

To find total sales for each product at each store:

```
select store_name, product_id, sum(sales)from
store_information group by store_name, product_id;
```

Store Name	**Product ID**	**SUM(Sales)**
Sangola	1	1800
Sangola	2	500
Pune	1	250
Sangli	1	700

⊙ **GROUP BY multiple columns and multiple functions**

To find total sales and the average sales for each product at each store:

```
select store_name, product_id, sum(sales), avg(sales)
from store_information group by store_name, product_id;
```

Store Name	**Product ID**	**SUM(Sales)**	**AVG(Sales)**
Sangola	1	1800	900
Sangola	2	500	500
Pune	1	250	250
Sangli	1	700	700

⊙ **Group by month / date / week:** To find total daily sales from Store_Information:

```
select txn_date, sum(sales)from store_information group
by txn_date order by txn_date;
```

Txn Date	SUM(Sales)
Jan-05-1999	2000

Total sales for both Sangola and Sangli stores.

```
select
```

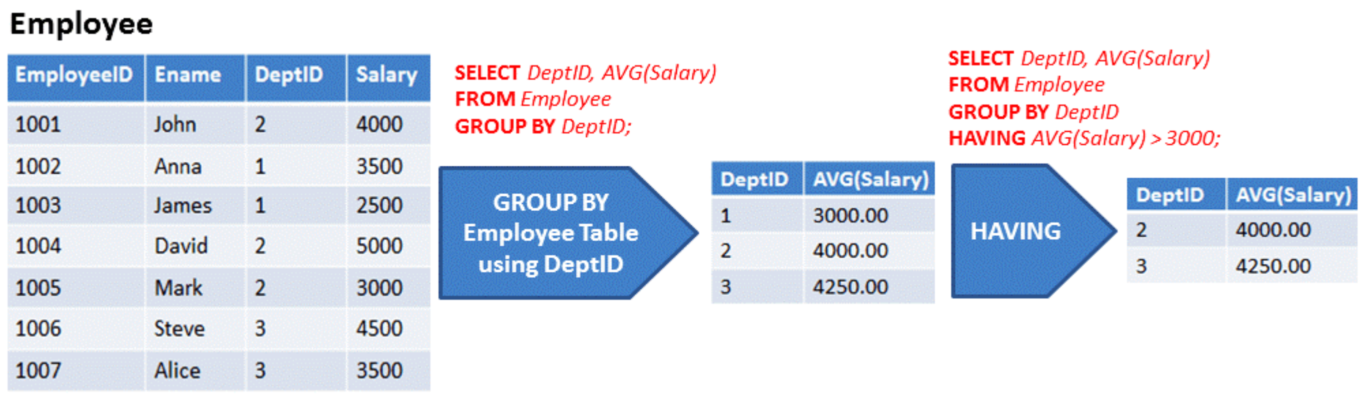
Jan-07-1999	250
Jan-08-1999	1000

```
store_name,sum(sales) from store_information where
store_name in('Sangola', 'Sangli') group by store_name;
```

- **HAVING Clause**
  - We know that WHERE clause is used to place conditions on columns but we want to place conditions on groups then, use HAVING clause.
  - The **HAVING** clause is used to filter group of rows based on a specified condition. It specifies the search condition for the group or aggregate. HAVING caluse is attached with **GROUP BY** clause. The HAVING clause works like the WHERE clause.
  - The HAVING clause was added to SQL because the WHERE keyword could not be used with aggregate functions.

Syntax: select column1, function_name(column2) **from** table_name **where** condition
group by column1, column2 **having** aggregate_function (<expression>) <operator> <value> **order by** column1, column2;

1. **function_name:** Name of the function used for example, SUM() , AVG().
2. **table_name:** Name of the table.
3. **Condition:** Condition used.



EmployeeID	Ename	DeptID	Salary
1001	John	2	4000
1002	Anna	1	3500
1003	James	1	2500
1004	David	2	5000
1005	Mark	2	3000
1006	Steve	3	4500
1007	Alice	3	3500

```
SELECT DeptID, AVG(Salary)
FROM Employee
GROUP BY DeptID;
```

DeptID	AVG(Salary)
1	3000.00
2	4000.00
3	4250.00

```
SELECT DeptID, AVG(Salary)
FROM Employee
GROUP BY DeptID
HAVING AVG(Salary) > 3000;
```

DeptID	AVG(Salary)
2	4000.00
3	4250.00

Employee table is grouped based on DeptID column and these grouped rows filtered using HAVING Clause with condition AVG(Salary) > 3000.

Example: select dno, count(eno) from emp group by dno having count(eno)>2;

<u>To see only the stores with sales over Rs.1500:</u>

```
select store_name, sum(sales) from store_information
group by store_name having sum(sales)>1500;
```

Store Name	**SUM(Sales)**
Sangola	**1800**

Total sales for both Sangola and Sangli stores.

```
select store_name, sum(sales) from store_information
group by store_name having store_name in('Sangola',
'Sangli');
```

WHERE	**HAVING**
Implemented in row operations.	Implemented in column operations.
Applied to Single row	Applied to Summarized row or groups.
It only fetches the particular data from particular rows according to the condition.	At first, complete data is fetched then separated according to the condition.
Aggregate Functions cannot appear in WHERE clause.	Aggregate Functions can appear in HAVING clause.
Used with SELECT and other statements such as UPDATE, DELETE etc.	Used with SELECT statement. Can't be used other statements.
Act as a Pre-filter	Act as a Post-filter
GROUP BY Comes after WHERE.	GROUP BY Comes before HAVING.

SQL Functions

Oracle functions can be used to manipulate data and returning the result. A function is performs a specific task. A function can accept user-supplied variables or constants and
perform operation on them. Such variables / constants are known as *arguments.*

```
Function_Name [ (argument-1, argument-2,.) ]
```

Functions are

1) Scalar Function (Single Row Function)
2) Aggregate (Group Function)
3) Miscellaneous Function

Single Row Functions

- Functions that work on only one value at a time are called <u>SCALAR Functions</u>.
- A single row function returns one result for every row of table / view.
  Example: LENGTH function calculates length of one string value.
- Single row functions can be grouped together depending on the data type of their arguments and return values.
  Ex: LENGTH function is related to string data type.

Functions can be classified according to different data types are:

1. Numeric Functions (For Number data type)

2. Character / String Functions (For String data type)
3. Date Function (For Date data type)
4. Conversion Functions (For Conversion of one data type to another)

- **Aggregate functions**
  - Functions that act on a set of values are called <u>Aggregate / Group Functions.</u>
    Example: SUM function calculates total of a column.
  - Group function returns a single result row for a group of rows.
  - Aggregate functions can appear in SELECT lists and in ORDER BY and HAVING
    clauses. They are commonly used with the GROUP BY clause in a SELECT statement.
  - Aggregate functions perform a calculation on a set of values and return a single, or summary, value.

Function	**Description**
AVG	Returns the average value of the given column or expression **Ex**: select avg (salary) as 'average salary' from emp;
COUNT	Number of rows where the value of the column is not NULL **Ex**: select count (dno) as 'number of deptwise ' from emp;
COUNT(*)	Number of rows returned including duplicates and NULLs **Ex**: select count (*) from employees where eno = 5;
MAX	Return Maximum value of the given column or expression. MAX can be used with numeric, character and date time columns. With character columns, MAX finds the highest value in the collating sequence. MAX ignores any null values. **Ex**: select max(salary) from emp; or select max(ename) from emp;
MIN	Return Minimum value of the given column or expression. MIN can be used with numeric, character and date time columns. With character

	columns, MIN finds the value that is lowest in the sort sequence. MIN ignores any null values. **Ex**: select min (salary) from emp; or select max(ename) from emp;
SUM	Returns the sum of all the values given column or expression. SUM can be used with numeric columns only. **Ex**: select sum (salary) as 'total salary' from emp;

➢ **Numeric Functions (For Number data type)**

- **ABS**: Absolute value is the measure of the magnitude of value. Absolute value is always a positive number.

Syntax: abs (*value*)
Ex: select abs(5), abs(-5), abs(0), abs(null) from dual;

```
ABS(5)        ABS(-5)    ABS(0)          ABS(NULL)
----------         ----------      ---------- -         --------
 ----
 5           -5           0
```

- **SIGN**: Sign gives the sign of a value.

Syntax: sign (*value*)
Ex: select sign(5), sign(-5), sign(0), sign(null) from dual;

```
SIGN(5)           SIGN(-5)         SIGN(0)         SIGN(NULL)
----------        ----------            ----------            --
 ------------
1           -1              0
```

- **SQRT**: This will give the square root of the given value.

Syntax: sqrt (value) -- here value must be positive.
Ex: select sqrt(4), sqrt(0), sqrt(null), sqrt(1) from dual;

```
SQRT(4)           SQRT(0)          SQRT(NULL)             SQRT(1)
----------        ----------            ---------------        --
 --------
2            0                                  1
```

- **MOD**: Display remainder value after value devides divisor.

Syntax: mod(value, divisor)
Ex: select mod(7,4), mod(1,5), mod(null,null), mod(0,0), mod(-7,4) from dual;

```
MOD(7,4)      MOD(1,5)   MOD(NULL,NULL)   MOD(0,0)    MOD(-7,4)
-------            --------          --------------      --------
          ----------
3            1                            0          -3
```

- **NVL**: This will substitute the specified value in the place of null values.

Syntax: nvl (*null_col, replacement_value*)
Ex: select * from student; -- here for 3rd row marks value is null

```
NO    NAME   MARKS
---  -------    ---------
```

```
1     a    100
2     b    200
3     c
select no, name, nvl(marks,300) from student;
NO     NAME  NVL(MARKS,300)
---    -------   ---------------------
1      a        100
2      b        200
3      c        300
select nvl(1,2), nvl(2,3), nvl(4,3), nvl(5,4) from dual;
NVL(1,2)   NVL(2,3)   NVL(4,3)   NVL(5,4)
----------      ----------      ----------      ---------
1          2          4          5
select nvl(0,0), nvl(1,1), nvl(null,null), nvl(4,4) from
dual;
NVL(0,0)   NVL(1,1)   NVL(null,null)       NVL(4,4)
---------- ----------      -----------------          ----------
0          1                            4
```

- **POWER:** Display m power nth value.

```
Syntax: power(m,n)
Ex: select power(2,5), power(0,0), power(null,null),
power(2,-5) from dual;
POWER(2,5)  POWER(0,0)      POWER(NULL,NULL)  POWER(2,-5)
-----------     -----------     -----------------                --
-------------
32                                            1 .03125
```

- **EXP:** This will raise e value to the give power.

```
Syntax: exp (value)
Ex:  select exp(1), exp(2), exp(0), exp(null), exp(-2)
from dual;
EXP(1)     EXP(2)           EXP(0)     EXP(NULL)        EXP(-2)
--------        ---------  -------         ----------       --------
--
2.71828183      7.3890561                        1 .135335283
```

- **CEIL:** Display lowest integer value which is greater than or equal to given value.

```
Syntax: ceil (value)
Ex:  select ceil(5), ceil(5.1), ceil(-5), ceil( -5.1),
ceil(0), ceil(null) from dual;
CEIL(5)    CEIL(5.1)  CEIL(-5)   CEIL(-5.1)  CEIL(0)   CEIL(NULL)
---------  --------         --------         ---------    -------
     ----------
5           6          -5         -5          0
```

- **FLOOR:** Display highest integer value which is lessthan or equal to given value.

```
Syntax: floor (value)
Ex: select floor(5), floor(5.1), floor(-5), floor(-5.1),
floor(0), floor(null) from dual;
FLOOR(5)    FLOOR(5.1)      FLOOR(-5)  FLOOR(-5.1)      FLOOR(0)
     FLOOR(NULL)
---------  ----------      ----------      ----------       --------
-    ------------
5            5          -5             -6          0
```

- **ROUND:** Display value *"m"* which is rounded to the *"n"* number of decimal places

```
Syntax: round (m,n)
```

```
Ex:  select round(123.2345), round(123.2345,2),
round(123.2354,2) from dual;
ROUND(123.2345) ROUND(123.2345,0)         ROUND(123.2345,2)
     ROUND(123.2345,-2)
--------------       ----------------              -------------
---             ----------------
123              123                 123.23
     100
```

- **TRUNC:** Display value **m** which is truncated to the **n** number of decimal places.

```
Syntax: trunc(m,n)
Ex:  select trunc(123.2345), trunc(123.2345,2),
trunc(123.2354,2) from dual;
TRUNC(123.2345) TRUNC(123.2345,2)         TRUNC(123.2345,-1)
     TRUNC(123.2345,-2)
--------------       ----------------              -------------
----            --------------
123              123.23              120                     100
```

- **GREATEST:** Display maximum value from the given values or expressions.

```
Syntax: greatest (value1, value2, value3 … valuen)
Ex: select greatest (1, 2, 3), greatest (-1, -2, -3) from
dual;
GREATEST (1,2,3)      GREATEST(-1,-2,-3)
--------------------   -----------------------
     3                -1
```

Ø If all the values are zeros then it will display zero.
Ø If any of one or all the parameters are nulls then it will display nothing.

- **LEAST:** Display minimum value from the given values or expression results.

```
Syntax: least (value1, value2, value3 … valuen)
Ex:  select least(1, 2, 3), least(-1, -2, -3) from dual;
LEAST(1,2,3)          LEAST(-1,-2,-3)
--------------------  -----------------------
     1               -3
```

Ø If all the values are zeros then it will display zero.
Ø If any of one or all the parameters are nulls then it will display nothing.

➢ String Functions

- **INITCAP:** Display the given string or column values with begining char as capital.

```
Syntax: initcap (string)
Ex: select initcap('computer') from dual;
        INITCAP
        -----------
        Computer
```

- **UPPER:** Display given string chars or column values in upper case.

```
Syntax: upper (string)
Ex: select upper('computer') from dual;
       UPPER
       -----------
      COMPUTER
```

- **LOWER**: Display the given string chars or column values in lower case.

Syntax: lower (*string*)
Ex: select lower('COMPUTER') from dual;

```
        LOWER
        -----------
       computer
```

- **LENGTH**: Display number of chars from the given string or column values

Syntax: length (*string*)
Ex: select length('computer') from dual; or select ename, length(ename) from emp;

```
    LENGTH
         ---------
         8
```

- **RPAD**: Display given string along with specific char in the right of string.

Syntax: rpad (*string, length [, padding_char]*)
Ex: select rpad('computer',15,'*'), rpad('computer',15,'*#') from dual;

```
          RPAD('COMPUTER'      RPAD('COMPUTER'
           ---------------           -----------------
           computer*******           computer*#*#*#*
```

-- Default padding character was blank space.

- **LPAD**: Display given string along with the specific char in the left of the string.

Syntax: lpad (*string, length [, padding_char]*)
Ex: select lpad('computer',15,'*'), lpad('computer',15,'*#') from dual;

```
     LPAD('COMPUTER'          LPAD('COMPUTER'
      -----------------       -----------------
      *******computer      *#*#*#*computer
```

-- Default padding character was blank space.

- **LTRIM**: Display given string by removing blank spaces from the left of string only.

Syntax: ltrim (*string [,unwanted_chars]*)
Ex: select ltrim('computer','co'), ltrim('computer','com') from dual;

```
     LTRIM(      LTRIM
      --------       ---------
       mputer     puter
```

select ltrim('computer','puter'), ltrim('computer','omputer') from dual;

```
      LTRIM('C  LTRIM('C
      ----------     ----------
       computer      computer
```

-- If you haven't specify any unwanted characters it will display entire string.

- **RTRIM**: Display given string by removing blank spaces from right of string only.

Syntax: rtrim (*string [, unwanted_chars]*)
Ex: select rtrim('computer','er'), rtrim('computer','ter') from dual;

```
     RTRIM(    RTRIM
      --------      ---------
     comput    compu
 select rtrim('computer','comput’),
rtrim('computer','compute') from dual;
     RTRIM('C  RTRIM('C
      ----------    ----------
     computer  computer
-- If you haven’t specify any unwanted characters it will
display entire string.
```

- **TRIM:** Display given string by eleminating blank spaces before and after the string.

Syntax: trim (*unwanted_chars* from *string*)
Ex: select trim('i' from 'indiani') from dual;

```
     TRIM(
     -----
     ndian
select trim('     welcome to     ') " trim" ||
initcap('oracle') from dual;
     welcome to Oracle
```

- **TRANSLATE:** display given string character by translating source character with corresponding target character.

Syntax: translate (*string, old_chars, new_chars*)
Ex: select translate('india','in','xy') from dual;

```
     TRANS
     --------
     xydxa
```

- **REPLACE:** Display given string by replacing source string with target string.

Syntax: replace (*string, old_chars [, new_chars]*)
Ex: select replace('india','in','xy'),
replace(‘india’,’in’) from dual;

```
     REPLACE   REPLACE
     -------       -------
     Xydia      dia
```

- **CONCAT:** This will be used to combine two strings only.

Syntax: concat (*string1, string2*)
Ex: select concat('computer',' operator') from dual;

```
     CONCAT('COMPUTER'
     -------------------------
     computer operator
 To combine more than two strings using concatenation
operator (||).
  select 'how' || ' are' || ' you' from dual;
     'HOW'||'ARE
     ---------------
     how are you
```

- **ASCII:** This will return the decimal representation in the database character set of the first character of the string.

Syntax: ascii (*string*)
Ex: select ascii('a'), ascii('apple') from dual;

```
     ASCII('A') ASCII('APPLE')
     --------       -----------
     97        97
```

- **CHR:** This will return the character having the binary equivalent to the string in either
  the database character set or the national character set.

Syntax: chr (*number*)
Ex: select chr(97) from dual;

```
     CHR
     -----
     a
```

- **SUBSTR:** This will be used to extract substrings.

Syntax: substr (string, start_chr_no [no_of_chars])
Ex: select substr('computer',2), substr('computer',2,5), substr('computer',3,7) from dual;

```
     SUBSTR(    SUBSTR     SUBSTR
     ----------      -------          --------
     omputer     omput      mputer
```

Ø If *no_of_chars* parameter is negative then it will display nothing.
Ø If both parameters except *string* are null or zeros then it will display nothing.
Ø If *no_of_chars* parameter is greater than the length of the string then it ignores and calculates based on the original string length.
Ø If *start_chr_count* is negative then it will extract the substring from right end.

```
     1     2   3    4    5     6   7    8
     C    O    M    P    U    T    E     R
     -8   -7   -6    -5  -4    -3   -2  -1
```

- **INSTR:** Display the position of char in the given string or column.

Syntax: instr (*string, search_str, [start_chr_count, [occurrence]]*)
Ex: select instr('information','o',4,1), instr('information','o',4,2) from dual;

```
     INSTR('INFORMATION','O',4,1)
INSTR('INFORMATION','O',4,2)
     ------------------------                -------------------
---------
               4                                 10
```

Ø If you are not specifying *start_chr_count* and *occurrence* then it will start search from the beginning and finds first occurrence only.
Ø If both parameters *start_chr_count* and *occurrence* are null, it will display nothing.

➢ DATE FUNCTIONS

Oracle default date format is DD-MON-YY. We can change the default format to our desired format by using following command.

```
alter session set nls_date_format = 'DD-MONTH-YYYY';
```

- **SYSDATE:** This will give the current date and time.

Ex: select sysdate from dual;

```
     SYSDATE
     -----------
     24-DEC-06
```

- **CURRENT_DATE**: This will returns the current date in the session's timezone.

```
Ex: select current_date from dual;
     CURRENT_DATE
     ------------------
     24-DEC-06
```

- **CURRENT_TIMESTAMP**: This will return the current timestamp with the active time zone information with system date, including fractional seconds.

```
Ex:  select current_timestamp from dual; or select
systimestamp from dual;
    select localtimestamp from dual; // returns local
timestamp
     CURRENT_TIMESTAMP
     ---------------------------------------------------------------------
     24-DEC-06   03.42.41.383369  AM   +05:30
```

- **DBTIMEZONE**: This will returns the current database time zone in UTC format. (Coordinated Universal Time)

```
Ex: select dbtimezone from dual;
     DBTIMEZONE
     ---------------
     -07:00
```

- **SESSIONTIMEZONE**: This will returns the value of the current session's time zone.

```
Ex:  select sessiontimezone from dual;
     SESSIONTIMEZONE
     ----------------------------------------------------
     +05:30
```

- **TO_CHAR**: This will be used to extract various date formats.

```
Syntax: to_char (date, format)
Ex: select to_char(sysdate,'dd month yyyy hh:mi:ss am
dy') from dual;
     TO_CHAR(SYSDATE,'DD MONTH YYYYHH:MI
     ----------------------------------------------------
     24 december 2006 02:03:23 pm sun
select to_char(sysdate,'dd month year') from dual;
     TO_CHAR(SYSDATE,'DDMONTHYEAR')
     -------------------------------------------------------
     24 december two thousand six
```

- **TO_DATE**: It will display any non-Oracle date format value in oracle's date format.

```
Ex: select to_date ('24/dec/2006', 'dd * month * day')
from dual;
     TO_DATE('24/DEC/20
     --------------------------
     24 * december * Sunday
     -- If you are not using to_char oracle will display
output in default date format.
```

- **ADD_MONTHS**: Display a date value after adding "n" number of months to the specified date.

```
Syntax: add_months (date, no_of_months)
```

Ex:select add_months('11-jan-90',5)from dual; **or** select add_months(sysdate,5)from emp;

```
     ADD_MONTHS
     ----------------
     11-JUN-90
```

select add_months ('11-jan-90', -5) from dual;

```
     ADD_MONTH
     --------------------- Ø If no_of_months is zero then it will
display the same date.
     11-AUG-89       Ø If no_of_months is null then it will display
nothing.
```

- **MONTHS_BETWEEN**: it shows number of months between dates.

Syntax: months_between (*date1, date2*)
Ex: select months_between ('11-aug-1990','11-jan-1990') from dual;

```
           MONTHS_BETWEEN ('11-AUG-1990','11-JAN-1990')
           ------------------------------------------------------
------
           7
```

select months_between(sysdate,'21-may-13') from dual;;

- **NEXT_DAY**: display the date value of given weekdayname after the specified date.

Syntax: next_day (*date, day*)
Ex: select next_day('24-dec-2006','sun') from dual;

```
     NEXT_DAY(
     -------------
     31-DEC-06       -- If the day parameter is null then it
will display nothing.
```

- **LAST_DAY**: Display the date value of last day in the month.

Syntax: last_day(*date*)
Ex: select last_day('24-dec-2006') from dual;

```
     LAST_DAY(
     -------------
     31-DEC-06
```

- **EXTRACT**: This is used to extract a portion of the date value.

Syntax: extract ((year | month | day | hour | minute | second), *date*)
Ex: select extract (year from sysdate) from dual;

```
     EXTRACT(YEARFROMSYSDATE)
     ------------------------------------
     2006                       -- You can extract only one
value at a time.
```

➢ **CONVERSION FUNCTIONS**

- **TO_CHAR (Date conversion)**: It accepts Oracle's date type data and convert it into required char format.

Syntax to_char(date_value, ['format'])
Exampleselect to_char('24-jun-14', 'month dd, yyyy') "new date format" from dual;

```
     New date format
     --------------------
     June  24, 2014
```

- **TO_CHAR (Number conversion):** it accepts number type data and convert it into character type data.

Syntax to_char(n,['format'])
Example select to_char (12345, '$099,999') "to char" from dual;

```
    To char
    --------------
    $012,345
```

- **TO_DATE:** It converts character data type into date data type. The format is a date format specifying the format of char_value.

Syntax to_date(char_value,['format'])
Example select to_date ('05/sep/14','dd/mon/yy') "to date" from dual;

```
    To date
    -------------
    05-SEP-14
```

- **TO_NUMBER:** It accepts the chardata which contains a sequence of digits and convert it into number type data.

Syntax to_number(char_value, ['format'])
Example select to_number('1234.56', '9999.9') "to number" from dual;

```
    To number
    -------------
    1234.5
```

Relational Algebra operations Relational algebra mainly provides a theoretical foundation for relational databases and SQL. The main purpose of using Relational Algebra is to define operators that transform one or more input relations into an output relation. Given that these operators accept relations as input and produce relations as output, they can be combined and used to express potentially complex queries that transform potentially many input relations (whose data are stored in the database) into a single output relation (the query results). As it is pure mathematics, there is no use of English Keywords in Relational Algebra and operators are represented using symbols.

Fundamental Operators

These are the basic/fundamental operators used in Relational Algebra.

1. Selection(σ)
2. Projection(π)
3. Union(U)
4. Set Difference(-)
5. Set Intersection(∩)
6. Rename(ρ)
7. Cartesian Product(X)

1. Selection(σ): It is used to select required tuples of the relations.

Example:

A	**B**	**C**
1	2	4
2	2	3
3	2	3
4	3	4

For the above relation, **σ(c>3)R** will select the tuples which have c more than 3.

A	**B**	**C**
1	2	4
4	3	4

Note: The selection operator only selects the required tuples but does not display them. For display, the data projection operator is used.

2. Projection(π): It is used to project required column data from a relation.

Example: Consider Table 1. Suppose we want columns B and C from Relation R.

π(B,C)R will show following columns.

B	**C**
2	4
2	3
3	4

Note: By Default, projection removes duplicate data.

3. Union(U): Union operation in relational algebra is the same as union operation in set theory.

Example:

FRENCH

Student_Name	**Roll_Number**
Ram	01
Mohan	02
Vivek	13
Geeta	17

GERMAN

Student_Name	**Roll_Number**
Vivek	13

Student_Name	**Roll_Number**
Geeta	17
Shyam	21
Rohan	25

Consider the following table of Students having different optional subjects in their course.

π(Student_Name)FRENCH U π(Student_Name)GERMAN

Student_Name
Ram
Mohan
Vivek
Geeta
Shyam
Rohan

Note: The only constraint in the union of two relations is that both relations must have the same set of Attributes.

4. Set Difference(-): Set Difference in relational algebra is the same set difference operation as in set theory.

Example: From the above table of FRENCH and GERMAN, Set Difference is used as follows

π(Student_Name)FRENCH - π(Student_Name)GERMAN

Student_Name
Ram
Mohan

Note: The only constraint in the Set Difference between two relations is that both relations must have the same set of Attributes.

5. Set Intersection(∩): Set Intersection in relational algebra is the same set intersection operation in set theory.

Example: From the above table of FRENCH and GERMAN, the Set Intersection is used as follows

π(Student_Name)FRENCH ∩ π(Student_Name)GERMAN

Student_Name
Vivek
Geeta

Note: The only constraint in the Set Difference between two relations is that both relations must have the same set of Attributes.

6. Rename(ρ): Rename is a unary operation used for renaming attributes of a relation.

ρ(a/b)R will rename the attribute 'b' of the relation by 'a'.

7. Cross Product(X): Cross-product between two relations. Let's say A and B, so the cross product between A X B will result in all the attributes of A followed by each attribute of B. Each record of A will pair with every record of B.

Example:

A

Name	**Age**	**Sex**
Ram	14	M
Sona	15	F
Kim	20	M

ID	**Course**
1	DS
2	DBMS

SQL JOINS

A JOIN clause is used to combine rows from two or more tables, based on a related column between them.

Let's look at a selection from the "Orders" table:

OrderID	**CustomerID**	**OrderDate**
10308	2	1996-09-18
10309	37	1996-09-19
10310	77	1996-09-20

Then, look at a selection from the "Customers" table:

CustomerID	**CustomerName**	**ContactName**	**Country**

1	Alfreds Futterkiste	Maria Anders	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mexico

Notice that the "CustomerID" column in the "Orders" table refers to the "CustomerID" in the "Customers" table. The relationship between the two tables above is the "CustomerID" column.

Then, we can create the following SQL statement (that contains an INNER JOIN), that selects records that have matching values in both tables:

Example:-

```
SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderDate
FROM Orders
INNER JOIN Customers ON Orders.CustomerID=Customers.CustomerID;
```

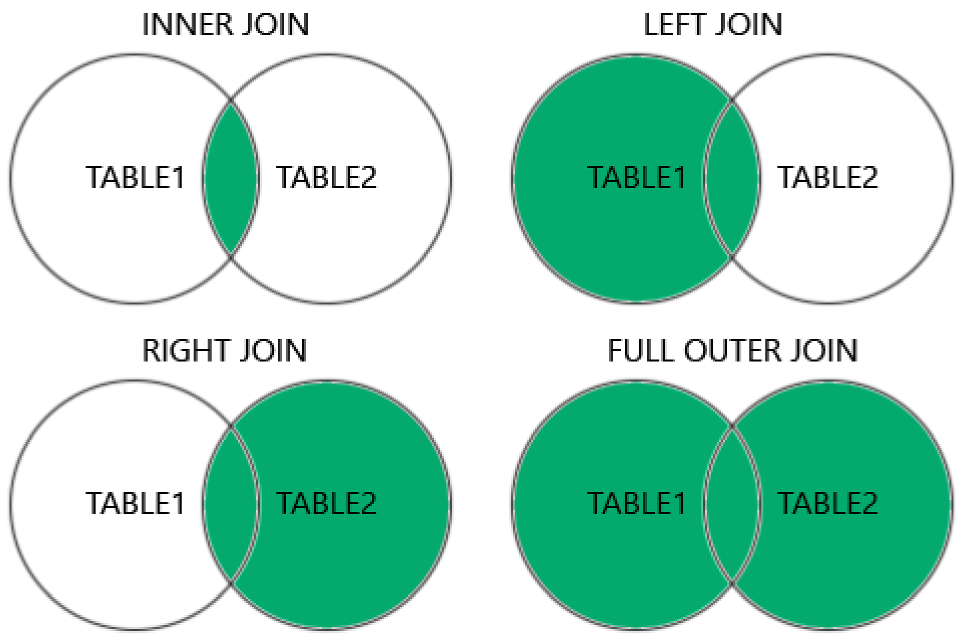
and it will produce something like this:

OrderID	**CustomerName**	**OrderDate**
10308	Ana Trujillo Emparedados y helados	9/18/1996
10365	Antonio Moreno Taquería	11/27/1996
10383	Around the Horn	12/16/1996
10355	Around the Horn	11/15/1996
10278	Berglunds snabbköp	8/12/1996

Different Types of SQL JOINs

Here are the different types of the JOINs in SQL:

- (INNER) JOIN: Returns records that have matching values in both tables
- LEFT (OUTER) JOIN: Returns all records from the left table, and the matched records from the right table
- RIGHT (OUTER) JOIN: Returns all records from the right table, and the matched records from the left table
- FULL (OUTER) JOIN: Returns all records when there is a match in either left or right table



INNER JOIN

The INNER JOIN keyword selects records that have matching values in both tables.

Let's look at a selection of the **Products** table:

ProductID	ProductName	CategoryID	Price
1	Chais	1	18

2	Chang	1	19
3	Aniseed Syrup	2	10

And a selection of the **Categories** table:

CategoryID	CategoryName	Description
1	Beverages	Soft drinks, coffees, teas, beers, and ales
2	Condiments	Sweet and savory sauces, relishes, spreads, and seasonings
3	Confections	Desserts, candies, and sweet breads

We will join the Products table with the Categories table, by using the CategoryID field from both tables:

```
SELECT ProductID, ProductName, CategoryName
FROM Products
INNER JOIN Categories ON Products.CategoryID =
Categories.CategoryID;
```

The INNER JOIN keyword returns only rows with a match in both tables. Which means that if you have a product with no CategoryID, or with a CategoryID that is not present in the Categories table, that record would not be returned in the result.

SQL LEFT JOIN Keyword

The LEFT JOIN keyword returns all records from the left table (table1), and the matching records from the right table (table2). The result is 0 records from the right side, if there is no match.

LEFT JOIN Syntax

```
SELECT column_name(s)
FROM table1
LEFT JOIN table2
ON table1.column_name = table2.column_name;
```

In some databases LEFT JOIN is called LEFT OUTER JOIN.

Below is a selection from the "Customers" table:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico

And a selection from the "Orders" table:

OrderID	CustomerID	EmployeeID	OrderDate	ShipperID
10308	2	7	1996-09-18	3
10309	37	3	1996-09-19	1
10310	77	8	1996-09-20	2

SQL LEFT JOIN Example

The following SQL statement will select all customers, and any orders they might have:

```
SELECT Customers.CustomerName, Orders.OrderID
FROM Customers
LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID
ORDER BY Customers.CustomerName;
```

The LEFT JOIN keyword returns all records from the left table (Customers), even if there are no matches in the right table (Orders).

SQL RIGHT JOIN Keyword

The RIGHT JOIN keyword returns all records from the right table (table2), and the matching records from the left table (table1). The result is 0 records from the left side, if there is no match.

RIGHT JOIN Syntax

```
SELECT column_name(s)
FROM table1
RIGHT JOIN table2
ON table1.column_name = table2.column_name;
```

Below is a selection from the "Orders" table:

OrderID	CustomerID	EmployeeID	OrderDate	ShipperID
10308	2	7	1996-09-18	3
10309	37	3	1996-09-19	1
10310	77	8	1996-09-20	2

And a selection from the "Employees" table:

EmployeeID	LastName	FirstName	BirthDate	Photo
1	Davolio	Nancy	12/8/1968	EmpID1.pic

2	Fuller	Andrew	2/19/1952	EmpID2.pic
3	Leverling	Janet	8/30/1963	EmpID3.pic

```
SELECT Orders.OrderID, Employees.LastName, Employees.FirstName
FROM Orders
RIGHT JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID
ORDER BY Orders.OrderID;
```

The RIGHT JOIN keyword returns all records from the right table (Employees), even if there are no matches in the left table (Orders).

SQL FULL OUTER JOIN Keyword

The FULL OUTER JOIN keyword returns all records when there is a match in left (table1) or right (table2) table records.

Tip: FULL OUTER JOIN and FULL JOIN are the same.

FULL OUTER JOIN Syntax

```
SELECT column_name(s)
FROM table1
FULL OUTER JOIN table2
ON table1.column_name = table2.column_name
WHERE condition;
```

FULL OUTER JOIN can potentially return very large result-sets!

Below is a selection from the "Customers" table:

CustomerID	**CustomerName**	**ContactName**	**Address**	**City**	**PostalCode**	**Country**
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Empareda	Ana Trujillo	Avda. de la Constitu	México D.F.	05021	Mexico

	dos y helados		ción 2222			
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico

And a selection from the "Orders" table:

OrderID	CustomerID	EmployeeID	OrderDate	ShipperID
10308	2	7	1996-09-18	3
10309	37	3	1996-09-19	1
10310	77	8	1996-09-20	2

SQL FULL OUTER JOIN Example

The following SQL statement selects all customers, and all orders:

```
SELECT Customers.CustomerName, Orders.OrderID
FROM Customers
FULL OUTER JOIN Orders ON Customers.CustomerID=Orders.CustomerID
ORDER BY Customers.CustomerName;
```

The FULL OUTER JOIN keyword returns all matching records from both tables whether the other table matches or not. So, if there are rows in "Customers" that do not have matches in "Orders", or if there are rows in "Orders" that do not have matches in "Customers", those rows will be listed as well.

SQL Self Join

A self join is a regular join, but the table is joined with itself.

Self Join Syntax

```
SELECT column_name(s)
FROM table1 T1, table1 T2
WHERE condition;
```

Below is a selection from the "Customers" table:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico

SQL Self Join Example

The following SQL statement matches customers that are from the same city:

```
SELECT A.CustomerName AS CustomerName1,
B.CustomerName AS CustomerName2, A.City
FROM Customers A, Customers B
WHERE A.CustomerID <> B.CustomerID
AND A.City = B.City
ORDER BY A.City;
```