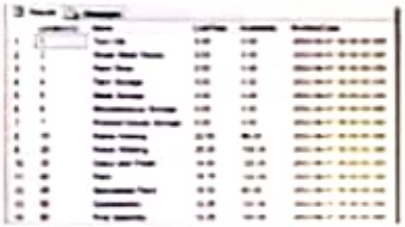
SQL (Structured Query Language)

- **SQL** is a **language** to specify **queries** in **structured** manner.
  - **Structured** means **relational data.**
  - **SQL** is a language to specify **queries** in a **relational database.**
- **SQL (Structured Query Language)** is a computer language for **storing, manipulating,** and **retrieving** data stored in **relational databases.**
- **SQL** allows **users** to communicate with **Relational Databases** and retrieve data from their **tables**
- **SQL** is the standard language for **RDBMS.**
  - All **Relational Database Management Systems (RDBMS)** like "*MySQL, MS Access, Oracle, Sybase, DB2, Informix, postgres and SQL Server*" **use SQL as standard database language.**
  - The data in RDBMS is stored in database objects called **tables.**
  - A **table** is a collection of related data entries and it consists of **columns** and **rows.**



History of SQL

- Dr. E.F. Codd published a paper on **Relational Model** ("A Relational Model Data from Large Shared Data Banks") in 1970 in ACM journal. Using the Mathematical concepts of Relational Algebra and Tuple Relational Calculus. This model was accepted as definitive model for **RDBMS.**
- **IBM** implemented the **SQL language**, originally called **SEQUEL** (**Structured English Query Languag** as part of the ***System R*** project in the early **1970s.**
  - **SEQUEL** is renamed/shortened to **SQL (Structured Query Language)**
- **SQL** became a standard of the ***American National Standards Institute (ANSI)*** in **1986**, and of the ***International Organization for Standardization (ISO)*** in **1987.**
- **ANSI** and **ISO** standard SQL has different **versions :**
  - SQL-86, SQL-89, SQL-92 , SQL:1999 , SQL:2003, SQL:2006, SQL:2008, SQL:2011, SQL:2016
- Vendors of Commercial systems (like Oracle, SQL Server etc.) offers most of the **major features**, plus **varying feature sets** from later standards and special proprietary **extensions.**

SQL (Structured Query Language)

- **SQL** is a **Domain-Specific Language.**
- **SQL** is **Declarative language i.e.** a **non-procedural language.**
  - **SQL** allows to declare **what we want to do**, but not how to do.
    - In **SQL**, we can specify through queries that **what data we want** but does not specify how to get those data.
    - Whereas, **C** is procedural language as it uses functions, procedures, loops, etc. **to specify what to do and how to do it.**

Question: How the query is actually performed in SQL?
Answer: The database/SQL engine is very powerful. When we write an SQL query, it first parses it; then it figures out its internal algorithms and it tries to select an algorithm which will be the best for that particular query. So, then it applies that algorithm, finds the answer and returns it.

- **SQL** is based on ***Relational Algebra*** and ***Tuple Relational Calculus.***

What Can SQL do?

- SQL can **execute queries** against a database
- SQL can **retrieve data** from a database
- SQL can **insert records** in a database
- SQL can **update records** in a database
- SQL can **delete records** from a database
- SQL can **create** new **databases**
- SQL can **create** new **tables** in a **database**
- SQL can **create** stored **procedures** in a **database**
- SQL can **create views** in a **database**
- SQL can **set permissions** on tables, procedures, and views

Types of SQL Commands

- **Data Definition Language (DDL)** commands are used to define the structure and schema of the database.
  - **CREATE** - to create new table or database
  - **ALTER** - to alter the structure of table
  - **DROP** - to delete a table from database
  - **TRUNCATE** - to delete all records from table
  - **RENAME** - to rename a table
- **Data Manipulation Language (DML)** commands are used for accessing and manipulating the data stored in the database.
  - **SELECT** - to retrieve data from the database
  - **INSERT** - to insert a new row into a table
  - **UPDATE** - to update existing row in a table
  - **DELETE** - to delete a row from a table

Programmers call these DML operations "CRUD".
CRUD** stands for:* ***Create, Read, Update, Delete
Performed by: ***INSERT, SELECT, UPDATE, DELETE***

Types of SQL Commands

- **Data Definition Language (DDL)** commands are used to define the structure and schema of the database.
  - **CREATE** - to create new table or database
  - **ALTER** - to alter the structure of table
  - **DROP** - to delete a table from database
  - **TRUNCATE** - to delete all records from table
  - **RENAME** - to rename a table

- **Data Manipulation Language (DML)** commands are used for accessing and manipulating the data stored in the database.
  - **SELECT** - to retrieve data from the database
  - **INSERT** - to insert a new row into a table
  - **UPDATE** - to update existing row in a table
  - **DELETE** - to delete a row from a table

Programmers call these DML operations "CRUD".
CRUD** stands for:* ***Create, Read, Update, Delete
Performed by: ***INSERT, SELECT, UPDATE, DELETE***

Types of SQL Commands...

- **Data Control Language (DCL)** are the commands to control the access of the data stored in the database
  - **GRANT** - grant permission to user for database access
  - **REVOKE** - take back granted permission from user
- **Transaction Control Language (TCL)** commands are used to manage the changes made by the DML statement.
  - **COMMIT** - to permanently save the transaction
  - **ROLLBACK** - to undo transaction
  - **SAVEPOINT** - to temporarily save a transaction so that you can rollback to that point whenever necessary

- **Note:** TCL Commands are used for only DML commands while DDL and DCL commands are auto-commited

SQL Rules

- **SQL** is **NOT case sensitive.**
  - For example: select is the same as SELECT
- But ***names*** of **databases, tables and columns** are **case sensitive**.
  - For example: In given SQL query, ACCOUNT is different from account

```
SELECT * FROM ACCOUNT;
SELECT * FROM account;
```

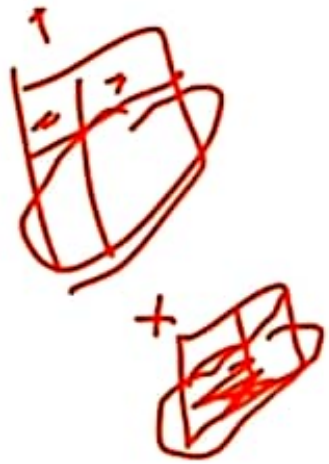
- **SQL statements** can use **multiple lines**
  - End each SQL statement with a **semi-colon** ;

Contents

- DDL Commands
- CREATE
  - SQL Data types
  - SQL Constraints
- ALTER
- DROP
- TRUNCATE
- RENAME

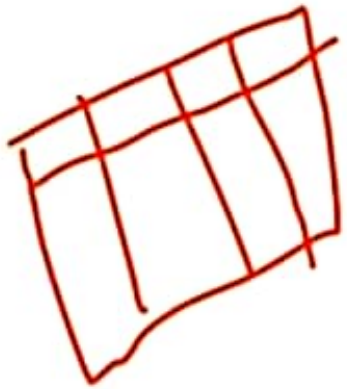
SQL: Data Definition Language (DDL)

- **Data Definition Language (DDL)** is used for creating and modifying the database objects such as tables, indices, views and users.
- **DDL Commands** are used to define the structure and schema of the database
- All the command of DDL are *auto-committed* that means it permanently save all the changes in the database.
- **Data Definition Language (DDL) Commands:**
  - **CREATE** - to create new table or database
  - **ALTER** - to alter (or modify) the structure of table
  - **DROP** - to delete a table from database
  - **TRUNCATE** - to delete all records from table
  - **RENAME** - to rename a table



1. CREATE

- **CREATE** statement is used to create database schema and to define the type and structure of the data to be stored in the database.
- **CREATE** statement can be used for
  - **Creating a Database,**
  - **Creating a Table**, etc.



I. Creating a DATABASE:

- **CREATE DATABASE** statement is used **to create a database in RDBMS.**

Syntax:

```
CREATE DATABASE database_name;
```

Example:

```
CREATE DATABASE my_db;
```

Note:

- **SHOW statement**: To see existing **databases** and **tables**

```
SHOW databases;
SHOW tables;
```

- **USE statement**: To use or select any existing database

Syntax **USE *database_name*;**

Example: **USE *my_db*;**

II. Creating a TABLE: Imp

- **CREATE TABLE** statement is used **to create a new table in a database.**
  - It specifies ***column names*** of the table, its ***data types*** (e.g. varchar, integer, date, etc.) and can also specify ***integrity constraints*** (e.g. Primary key, foreign key, not null, unique).

Syntax:

```
CREATE TABLE  table_name
(
        column_name1    datatype,
        column_name2    datatype,
             …                  …
        column_name n   datatype,
        (Integrity Constraints)
);
```

Example: Create table

Example 1:

```
CREATE TABLE Emp(
    Emp_ID int,
    Name varchar(20),
    Age int,
    Address varchar(100),
    Salary numeric(10,2)
);
```

Emp_ID	Name	Age	Address	Salary

Example 2:

```
CREATE TABLE branch(
    branch-name varchar(15),
    branch-city varchar(30),
    assets integer,
    primary key (branch-name),
    check(assets>0)
);
```

branch-name	branch-city	assets

Note:

DESC (or DESCRIBE) statement: To describe the details of the table structure

Syntax: **DESC *table_name*;**

Example: **DESC *Emp*;**

SQL DATA TYPES

- **Each column** in a database **table** is required to have a ***name*** and a ***data type.***
- The **data type** is a guideline for SQL to understand *what type of data is expected inside of each column,* and it also *identifies how SQL will interact with the stored data.*

- **Note:** Data types might have different names in different database. And even if the name is the same, the size and other details may be different! Always check the documentation!

Data type	Description
CHAR(size)	A FIXED length string (can contain letters, numbers, and special characters), with user-specified length *size*.
VARCHAR(size)	A VARIABLE length string (can contain letters, numbers, and special characters), with user-specified length *size*. Full From Character Varying
INT / INTEGER	A medium Integer (Signed range is from -2,147,483,648 to 2,147,483,647. Unsigned range is from 0 to 4294967295). 4 bytes
SMALLINT	A small integer (Signed range is from -32,768 to 32,767. Unsigned range is from 0 to 65,535.) 2 bytes
BIGINT	A large integer (Signed range is from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,808. Unsigned range is from 0 to 18446744073709551615). 8 bytes
NUMERIC(size,d)/ DECIMAL(size,d)/ FLOAT(size,d)	A small floating point number. Its *size* parameter specifies the total number of digits. The number of digits after the decimal point is specified by *d* parameter. For NUMRIC(3,1), 46.2 can be stored but not 466.0 or 0.32
REAL, DOUBLE PRECISION	Floating point and double-precision floating point numbers, with machine-dependent precision.
FLOAT(n)	A floating point number, with user-specified precision of at least *n* digits.
DOUBLE(size, d)	A large floating point number. The total number of digits is specified in *size*. The number of digits after the decimal point is specified in the *d* parameter

Data type	Description
DATE	A date. Format: YYYY-MM-DD. The supported range is from '1000-01-01' to '9999-12-31'
TIME	A time. Format: hh:mm:ss. The supported range is from '-838:59:59' to '838:59:59'
DATETIME / TIMESTAMP	A date and time combination. Format: YYYY-MM-DD hh:mm:ss.
YEAR	A year in four-digit format. Values allowed in four-digit format: 1901 to 2155, and 0000.

Note: There are many other data types.....

Data type	Description
DATE	A date. Format: YYYY-MM-DD. The supported range is from '1000-01-01' to '9999-12-31'
TIME	A time. Format: hh:mm:ss. The supported range is from '-838:59:59' to '838:59:59'
DATETIME / TIMESTAMP	A date and time combination. Format: YYYY-MM-DD hh:mm:ss.
YEAR	A year in four-digit format. Values allowed in four-digit format: 1901 to 2155, and 0000.

Note: There are many other data types.....

SQL Constraints

- **SQL constraints** are used to specify rules for the data in a table.
- **Constraints** are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the table. If there is any violation between the constraint and the data action, the action is aborted.
- **Constraints** can be **column level** or **table level**.
  - ***Column level*** constraints apply to a column
  - ***Table level*** constraints apply to the whole table.
- **Constraints** can be specified:
  - when a table is created with the **CREATE TABLE** statement, or
  - we can use the **ALTER TABLE** statement to create constraints even after the table is created.

SQL Constraints:

- ***NOT NULL***: Ensures that a column cannot have NULL value.
- ***DEFAULT***: Provides a default value for a column when none is specified.
- ***UNIQUE***: Ensures that all values in a column are different.
- ***PRIMARY KEY***: A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table
- ***FOREIGN KEY***: Uniquely identifies a row/record in another table.
- ***CHECK***: Ensures that all the values in a column satisfies certain conditions
- ***INDEX***: Used to create and retrieve data from the database very quickly

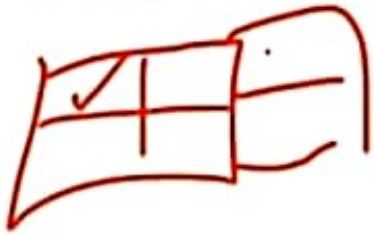
Example:

```
CREATE TABLE  Emp
(
        Emp_ID int NOT NULL,
        Name varchar(20) NOT NULL,
        Age int NOT NULL,
        Address varchar(100) DEFAULT 'India',
        Salary numeric(10,2) NOT NULL,
        DeptID int,

        PRIMARY KEY(Emp_ID),
        FORIEGN KEY (Dept_ID) REFERENCES Dept(Dept_ID),
        CHECK(Age>=18)
);
```

2. ALTER

- ***ALTER TABLE*** statement is **used to modify the structure of the existing table**
- It is used to add, delete, modify or rename columns in an existing table.
- It is also used to add and drop various constraints on an existing table.



1. **ALTER TABLE – ADD COLUMN**
   - For adding new columns in a table
2. **ALTER TABLE - DROP COLUMN**
   - For removing existing columns in table
3. **ALTER TABLE - MODIFY COLUMN**
   - To modify existing columns in a table
4. **ALTER TABLE – RENAME COLUMN**
   - To rename an existing column in a table

1. ALTER TABLE – ADD COLUMN

- For adding new columns in a table
- ***Syntax:***

```
ALTER TABLE table_name
ADD column_name(s)  datatype(s);
```

Example:

```
ALTER TABLE Emp
ADD Email varchar(100);
```

Emp_ID	Name	Age	Address	Salary	Email

2. ALTER TABLE - DROP COLUMN

- For removing existing columns in a table
- *Syntax*:

```
ALTER TABLE table_name
DROP column_name(s);
```

Example:

```
ALTER TABLE Emp
DROP Email;
```

Emp_ID	Name	Age	Address	Salary

3. ALTER TABLE - MODIFY COLUMN

- **To modify existing columns in a table**
  - To change **data type** of any column or to modify its **size.**
- ***Syntax*:**

```
ALTER TABLE table_name .
MODIFY column_name  datatype;
```

Example:

```
ALTER TABLE Emp
MODIFY Name varchar(100);
```

Check it with **DESC** Command

3. DROP

- **DROP TABLE** statement **completely removes a table from the database.**
  - This command will destroy the table structure and the data stored in it.

Syntax:

```
DROP TABLE table_name;
```

Example:

```
DROP TABLE Emp;
```

4. ALTER TABLE – RENAME COLUMN

- To rename an existing column in a table
- ***Syntax***:

```
ALTER TABLE table_name
RENAME old_column_name To new_column_name;
```

Example:

```
ALTER TABLE Emp
MODIFY Address To Location;
```

Emp_ID	Name	Age	Location	Salary

3. DROP

- **DROP TABLE** statement **completely removes a table from the database.**
  - This command will destroy the table structure and the data stored in it.

Syntax:

```
DROP TABLE table_name;
```

Example:

```
DROP TABLE Emp;
```

DROP command is also be used on Databases.

DROP DATABASE statement is used to delete the complete database.

Syntax: `DROP DATABASE database_name;`

Example: `DROP DATABASE my_db;`

4. TRUNCATE

- **TRUNCATE TABLE** statement is used **to remove all rows (complete data) from a table.** It is similar to the DELETE statement with no WHERE clause.
- ***TRUNCATE TABLE Vs DROP TABLE***
  - DROP TABLE command can also be used to delete complete table but it deletes table structure too. TRUNCATE TABLE doesn't delete the structure of the table.
- ***Syntax:***

  TRUNCATE TABLE *table_name;*

Emp

Emp_ID	Name	Age	Location	Salary
1	Rahul	32	Delhi	2000
2	Kamal	25	Pune	1500

Example:

TRUNCATE TABLE *Emp;*

Emp_ID	Name	Age	Location	Salary

5. RENAME

- ***RENAME TABLE*** statement is used **to change the name of a table.**
- ***Syntax*:**

RENAME TABLE *old_table* _name To *new_table_name*;

Example:

RENAME TABLE *Emp* To *Employee*;

Check it with
Show Tables;
Command

Contents:

- **DML COMMANDS**
  - **CRUD**
- **SELECT**
  - WHERE Clause
  - Basic Query Structure
  - Optional Clauses
  - SQL Operators
- **INSERT**
- **UPDATE**
- **DELETE**

SQL: Data Manipulation Language (DML)

- **Data Manipulation Language (DML)** commands are used **for accessing and manipulating the data stored in the database.**
- The DML commands are *not auto-committed* that means it can't permanently save all the changes in the database. They can be rollback.
- **Data Manipulation Language (DML)**
  - **SELECT** - to retrieve data from the database ⟸ **Data Query language (DQL)**
  - **INSERT** - to insert a new row into a table
  - **UPDATE** - to update existing row in a table
  - **DELETE** - to delete a row from a table

  → **Modification of Database**

R C U D

Programmers call these DML operations "CRUD".
***CRUD** stands for:* *Create, Read, Update, Delete*
***Performed** by:* *INSERT, SELECT, UPDATE, DELETE*

1. SELECT (mostly used SQL command/query)

- **SELECT** statement is **used to select a set of data from a database table.** Or Simply **SELECT** statement is **used to retrieve data from a database table.**
  - It returns data in the form of a result table. These result tables are called result-sets.
- **SELECT** is also called **DQL** because it is used to **query** information from a database table
- **SELECT** statement specifies **column names**, and **FROM** specifies **table name**
- **SELECT** command is used with different **Conditions** and **CLAUSES**.

Basic Syntax:

To retrieve selected fields from the table:

```
SELECT column1, column2, ... columnN
FROM table_name(s);
```

To retrieve all the fields from the table:

```
SELECT *
FROM table_name(s);
```

Example Table: Emp

Columns/Fields/Attributes

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

Records/ Rows/ Tuples

Example: SELECT

Example 1:

To retrieve columns *Emp_ID, Name* from *Emp* table

```
SELECT Emp_ID, Name
FROM Emp;
```

Emp_ID	Name
1	Rahul
2	Kamal
3	Karan
4	Chirag
5	Harsh
6	Kajal
7	Mahi

Example 2:

To retrieve *all the fields* from *the Emp* table

```
SELECT *
FROM Emp;
```

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

WHERE CLAUSE

- **WHERE Clause** is used **to select a particular record based on a condition**. It is used to filter records.
- **WHERE Clause is used to specify a condition while fetching the data from a single table or by joining with multiple conditions**
- The **WHERE** clause is not only used in the **SELECT** statement, but it is also used in the **UPDATE**, **DELETE** statement.
- ***Syntax: "SELECT with WHERE Clause"***

```
SELECT column1, column2, ... columnN
FROM table_name(s)
WHERE condition;
```

Example:

To retrieve *Emp_ID, Name* from *Emp* table whose *Salary* is greater than 2000

```
SELECT Emp_ID, Name
FROM Emp
WHERE Salary>2000;
```

Emp_ID	Name
4	Chirag
5	Harsh
6	Kajal
7	Mahi

Basic Query Structure

- **SQL** is based on **set and relational operations** with certain modifications and enhancements
- A **typical SQL query** has the form:

```
select A1, A2, ..., An
from r1, r2, ..., rm
where P
```

  - **Select command** is equivalent to **Projection** (Π)
  - **Where Clause** is equivalent to **Selection** (σ)

  - A_is represent *attributes*
  - r_is represent *relations*
  - *P* is a *predicate (or condition).*
- This **SQL query** is equivalent to the **relational algebra expression.**

$$\Pi_{A1.A2.\ldots.An}(\sigma_P(r_1 \times r_2 \times \ldots \times r_m))$$

- The **result** of an **SQL query** is a **relation (or table).**

Optional Clauses in SELECT statemer

- [**WHERE Clause**] : It specifies which rows to retrieve by specifying **conditions.**
- [**GROUP BY Clause**] : **Groups rows** that share a **property** so that the **aggregate function** can be applied to each group.
- [**HAVING Clause**] : It **selects** among the **groups** defined by the **GROUP BY** clause by specifying **conditions. .**
- [**ORDER BY**] : It specifies an order in which to return the rows.

- [**DISTINCT Clause**]: It is used to remove duplicates from results set of a **SELECT** statement. (**SELECT DISTINCT**)

SQL Operators

1. SQL Arithmetic Operator
2. SQL Comparison Operators
3. SQL Logical Operators
4. SQL Special Operators

Optional Clauses in SELECT statemer

- **[WHERE Clause]** : It specifies which rows to retrieve by specifying **conditions.**
- **[GROUP BY Clause]** : **Groups rows** that share a **property** so that the **aggregate function** can be applied to each group.
- **[HAVING Clause]** : It **selects** among the **groups** defined by the **GROUP BY** clause by specifying **conditions. .**
- **[ORDER BY]** : It specifies an order in which to return the rows.

- **[DISTINCT Clause]:** It is used to remove duplicates from results set of a **SELECT** statement. (**SELECT DISTINCT**)

1. SQL Arithmetic Operators

Operator	Description
+	Add
–	Subtract
*	Multiply
/	Divide
%	Modulo

Example:

To retrieve *Emp_ID, Name, Salary plus 3000 from Emp* table

```
SELECT Emp_ID, Name, Salary + 3000
FROM Emp;
```

Emp_ID	Name	Salary
1	Rahul	5000.00
2	Kamal	4500.000
3	Karan	5000.00
4	Chirag	9500.00
5	Harsh	11500.00
6	Kajal	7500.00
7	Mahi	13000.00

2. SQL Comparison Operators

Operator	Description
=	Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
<> or !=	Not equal to

Example:

To retrieve *Emp_ID, Name* from *Emp* table whose *Salary* is greater than 2000

```
SELECT Emp_ID, Name, Salary
FROM Emp
WHERE Salary >2000;
```

Emp_ID	Name	Salary
4	Chirag	6500.00
5	Harsh	8500.00
6	Kajal	4500.00
7	Mahi	10000.00

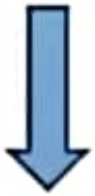
3. SQL Logical Operators

Operator	Description
AND	It displays a record if all the conditions separated by AND are TRUE.
OR	It displays a record if any of the conditions separated by OR is TRUE
NOT	• It displays a record if the condition(s) is NOT TRUE. • It reverses the meaning of any operator with which it is used. This is a negate operator. Eg: NOT EXISTS, NOT BETWEEN, NOT IN, IS NOT NULL, etc.

Example:

To retrieve all records from *Emp* table whose *salary* is greater than 2000 and age is less than 25

```
SELECT *
FROM Emp
WHERE Salary>2000 AND Age<25;
```



Emp_ID	Name	Age	Address	Salary
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

4. SQL Special Operators

Operator	Description
ALL	It compares a value to all values in another value set (in subqueries)
ANY	It compares the values in the list according to the condition (in subqueries).
BETWEEN	It is used to search for values that are within a set of values (given minimum and maximum values).
EXISTS	It is used to search for the presence of a row in a specified table (in subqueries).
IN	It compares a value to that specified list value (and in subquries).
LIKE	It compares a value to similar values using wildcard operator (% , _).
IS NULL	It checks for missing data or NULL values

Example:

To retrieve all records from *Emp* table with salary between 2000 and 5000

```
SELECT *
FROM Emp
WHERE Salary BETWEEN 2000 AND 5000;
```

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
3	Karan	23	Pune	2000.00
6	Kajal	22	Bilaspur	4500.00

2. INSERT

- The **INSERT INTO** statement is **used to insert new records in a table.**
- It is possible to write the **INSERT INTO** statement in **two ways.**

Syntax 1: **Specify both** the **column names** and the **values** to be inserted

```
INSERT INTO table_name(column1, column2, column3,...columnN)
VALUES (value1, value2, value3,...valueN);
```

Syntax 2: **Specify only values** to be inserted. But needed to **remember** the **column order**

```
INSERT INTO table_name
VALUES (value1, value2, value3,...valueN);
```

Example: INSERT INTO

- INSERT INTO Emp (Emp_ID, Name, Age, Address, Salary) VALUES (1, 'Rahul', 32, 'Delhi',2000);
- INSERT INTO Emp (Emp_ID, Name, Age, Address, Salary) VALUES (2, 'Kamal', 25, 'Pune',1500);
- INSERT INTO Emp (Emp_ID, Name, Age, Address, Salary) VALUES (3, 'Karan', 23, 'Pune', 2000);

1st way

- INSERT INTO Emp VALUES (4, 'Chirag', 25, 'Mumbai', 6500);
- INSERT INTO Emp VALUES (5, 'Harsh', 32, 'Mumbai', 8500);
- INSERT INTO Emp VALUES (6, 'Kajal', 22, 'Bilaspur', 4500);
- INSERT INTO Emp VALUES (7, 'Mahi', 24, 'Patna', 10000);

2nd way

Example: INSERT INTO

```
INSERT INTO Emp (Emp_ID, Name, Age, Address, Salary) VALUES (1, 'Rahul', 32, 'Delhi',2000);
INSERT INTO Emp (Emp_ID, Name, Age, Address, Salary) VALUES (2, 'Kamal', 25, 'Pune',1500);
INSERT INTO Emp (Emp_ID, Name, Age, Address, Salary) VALUES (3, 'Karan', 23, 'Pune', 2000);
```

1st way

```
INSERT INTO Emp VALUES (4, 'Chirag', 25, 'Mumbai', 6500);
INSERT INTO Emp VALUES (5, 'Harsh', 32, 'Mumbai', 8500);
INSERT INTO Emp VALUES (6, 'Kajal', 22, 'Bilaspur', 4500);
INSERT INTO Emp VALUES (7, 'Mahi', 24, 'Patna', 10000);
```

2nd way

To Check:

```
SELECT *
FROM Emp;
```

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

3. UPDATE

- The **UPDATE** statement is **used to modify the existing records in a table.**
- **Note**: ***Always use the WHERE clause with the UPDATE statement to update the selected rows, otherwise all the rows would be affected.***

Syntax:

```
UPDATE table_name
SET column1 = value1, column2 = value2, ..... columnN = valueN
WHERE condition;
```

Example: UPDATE

Example:

To update the address of Emp_ID: 3 to *Chennai*

```
UPDATE Emp
SET Address='Chennai'
WHERE Emp_ID=3;
```

To Check:

```
SELECT *
FROM Emp;
```

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
2	Kamal	25	Pune	1500.00
3	Karan	23	Chennai	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

4. DELETE

- The **DELETE** statement is **used to delete existing records in a table.**
- **Note**: ***Always use the WHERE clause with a DELETE statement to delete the selected rows, otherwise all the records would be deleted.***

Syntax:

```
DELETE FROM table_name
WHERE condition;
```

Example: DELETE

Example:

To delete the employee records of *Pune* location

```
DELETE FROM Emp
WHERE Address='Pune';
```

To Check:

```
SELECT *
FROM Emp;
```

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

SELECT DISTINCT: "DISTINCT Clause"

- The **SELECT DISTINCT** statement is **used to return only distinct (different) values.**

  Or **simply it is used to return only unique values.**

- **DISTINCT Clause** is used **to remove duplicates** from results set of a **SELECT** statement.
  - Inside a table, a **column** often **contains many duplicate values;** and sometimes you only want to list the different (distinct) values.

Syntax:

```
SELECT DISTINCT column1, column2, ... columnN
FROM table_name;
```

Example Table: Emp

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

SELECT without DISTINCT

```
SELECT Address
FROM Emp;
```

Address
Delhi
Pune
Pune
Mumbai
Mumbai
Bilaspur
Patna

SELECT without DISTINCT

```
SELECT Address
FROM Emp;
```

Address
Delhi
Pune
Pune
Mumbai
Mumbai
Bilaspur
Patna

SELECT with DISTINCT

Example:

To show distinct (different) addresses in *Emp* table

```
SELECT DISTINCT Address
FROM Emp;
```

Address
Delhi
Pune
Mumbai
Bilaspur
Patna

To count Distinct values

Example:

To show number of distinct (different) addresses in *Emp* table

```
SELECT  COUNT (DISTINCT Address)
FROM Emp;
```

5

COUNT
is an aggregate function

SELECT DISTINCT....

Note:

- When only **one column** (expression) is **provided** in the **DISTINCT clause**, the query will return the **unique values** for that **column.**
- When **more than one column** (expression) is **provided** in the **DISTINCT clause**, the query will retrieve **unique combinations** for the **columns** listed.
- In SQL, the **DISTINCT clause** doesn't **ignore NULL values.** So when using the **DISTINCT** clause in your SQL statement, your result set will include **NULL** as a distinct value.

For Eg: Given Emp Address field

Address
Pune
Pune
Mumbai
NULL
NULL

```
SELECT DISTINCT Address
FROM Emp;
```

Address
Pune
Mumbai
NULL

SQL: AND, OR, NOT OPERATOR

- The **WHERE** clause can be combined with **AND**, **OR**, and **NOT** operators.
  - To make more precise conditions for fetching data from database by combining more than one condition together.
- **The AND and OR operators are used to filter records based on more than one condition**
  - The **AND** operator displays a **record** if **all the conditions** separated by **AND** are **TRUE.**
  - The **OR** operator displays a **record** if **any of the conditions** separated by **OR** is **TRUE.**
- **The NOT operator displays a record if the condition(s) is NOT TRUE**
  - **NOT** reverses the meaning of any operator with which it is used. This is a **negate operator.**
  - **Eg: NOT IN, NOT BETWEEN, NOT EXISTS, IS NOT NULL,** etc.

Example Table: Emp

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

SQL: AND OPERATOR

- **AND operator (or AND condition)** is used to combine multiple conditions with **WHERE** clause, in **SELECT**, **UPDATE** or **DELETE** statements.
- All conditions must be satisfied for a record to be selected.
- ***Syntax*:**

```
SELECT column1, column2, ... columnN
FROM table_name
WHERE condition1 AND condition2.....AND conditionN;
```

Example:

Selects all records from *Emp* table whose *salary* is greater than 2000 and age is less than 25

```
SELECT *
FROM Emp
WHERE Salary>2000 AND Age<25;
```

Emp_ID	Name	Age	Address	Salary
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

SQL: OR OPERATOR

- **OR operator (or OR condition)** is used to combine multiple conditions with **WHERE** clause, in **SELECT, UPDATE** or **DELETE** statements.
- Any one of the conditions must be satisfied for a record to be selected.
- ***Syntax*:**

```
SELECT column1, column2, ... columnN
FROM table_name
WHERE condition1 OR condition2.....OR conditionN;
```

Example:

Selects all records from *Emp* table whose *salary* is greater than 2000 or whose age is less than 25

```
SELECT *
FROM Emp
WHERE Salary>2000 OR Age<25;
```

Emp_ID	Name	Age	Address	Salary
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

SQL: NOT OPERATOR

- ***NOT operator (or NOT condition)*** is used to negate the conditions with the **WHERE** clause, in **SELECT, UPDATE** or **DELETE** statements.
- It displays a record if the condition(s) is NOT TRUE
- **NOT** reverses the meaning of any operator with which it is used. This is a **negate operator.**
  - **Eg: NOT IN, NOT BETWEEN, NOT EXISTS, IS NOT NULL, etc.**
- ***Syntax*:**

```
SELECT column1, column2, ... columnN
FROM table_name
WHERE NOT condition;
```

SQL: NOT OPERATOR

- ***NOT operator (or NOT condition)*** is used to negate the conditions with the **WHERE** clause, in **SELECT, UPDATE** or **DELETE** statements.
- It displays a record if the condition(s) is NOT TRUE
- **NOT** reverses the meaning of any operator with which it is used. This is a **negate operator.**
  - **Eg: NOT IN, NOT BETWEEN, NOT EXISTS, IS NOT NULL, etc.**
- ***Syntax*:**

```
SELECT column1, column2, ... columnN
FROM table_name
WHERE NOT condition;
```

Example:

Selects all records from where address is *not Mumbai*

```
SELECT *
FROM Emp _
WHERE NOT Address='Mumbai';
```

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

Example: Combining AND and OR Conditions

- The SQL **AND condition** and **OR condition** can be combined to test for multiple conditions with the **WHERE** clause, in **SELECT, UPDATE** or **DELETE** statements.
- When combining these conditions, it is **important** to **use parentheses ()**.
  - The **parentheses determine the order** that the AND and OR conditions are evaluated.
- **Precedence order: NOT, AND, OR**

Example:

Selects all records from *Emp* table whose *salary* is greater than 2000 and age is less than 25 or whose Emp ID is 2.

```
SELECT *
FROM Emp
WHERE (Salary>2000 AND Age<25) OR (Emp_ID=2);
```

Emp_ID	Name	Age	Address	Salary
2	Kamal	25	Pune	1500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

SQL: IN OPERATOR

- The IN operator allows us **to specify multiple values in a WHERE clause.**
  - It allows us to easily test if an expression matches any value in a list of values
- The IN operator is **a shorthand for multiple OR conditions.** So it **used to replace many OR conditions.**
- **Syntax:**

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN(value1, value2, ... valueN);
```

OR (In Subqueries)

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN(SELECT STATEMENT);
```

Subquery
A subquery is a *select-from-where* expression that is nested within another query.

Example Table: Emp

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

Example: IN OPERATOR

Example 1:

It selects all Employees located in Delhi, Bilaspur and Patna

```
SELECT *
FROM Emp
WHERE Address IN ('Delhi', 'Bilaspur','Patna);
```

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

Example 2:

It selects all Employees whose salary is either 2000, 4500 or 10000

```
SELECT *
FROM Emp
WHERE Salary IN (2000, 4500,10000);
```

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
3	Karan	23	Pune	2000.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

Example: IN OPERATOR using Subquery

Example 3:

It selects all employees that are from the same address as the customers

```
SELECT *
FROM Emp
WHERE Address IN(SELECT Address FROM Customers);
```

Address
Delhi
Pune
Pune
Mumbai
Mumbai
Bilaspur
Patna

Address
Delhi
Bhopal
Bangaluru
Chennai
Bilaspur

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
6	Kajal	22	Bilaspur	4500.00

SQL: NOT IN OPERATOR

- **NOT IN operator** is just opposite of **IN operator**
- **Syntax**:

```
SELECT column_name(s)
FROM table_name
WHERE column_name NOT IN (value1, value2, ... valueN);
```

Example:

It selects all Employees *not* located in Delhi, Bilaspur and Patna

```
SELECT *
FROM Emp
WHERE Address NOT IN ('Delhi','Bilaspur','Patna');
```

Emp_ID	Name	Age	Address	Salary
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00

SQL: BETWEEN OPERATOR

- **BETWEEN** operator (**BETWEEN...AND**) is used **to select values within given range** (given minimum and maximum values).
- The values can be numbers, text, or dates.
- The **BETWEEN** operator is **inclusive**: **begin** and **end** values are **included**.
- **Syntax**:

```
SELECT column_name(s)
FROM table_name
WHERE column_name BETWEEN value1 AND value2;
```

Example:

It selects all Employees with the Salary between 2000 to 7000

```
SELECT *
FROM Emp
WHERE Salary BETWEEN 2000 AND 7000;
```

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.00
6	Kajal	22	Bilaspur	4500.00

SQL: NOT BETWEEN OPERATOR

- **NOT BETWEEN** operator is opposite of **BETWEEN** operator
- **Syntax**:

```
SELECT column_name(s)
FROM table_name
WHERE column_name NOT BETWEEN value1 AND value2;
```

Example:

It selects all Employees with the salary not between 2000 to 7000

```
SELECT *
FROM Emp
WHERE Salary NOT BETWEEN 2000 AND 7000;
```

Emp_ID	Name	Age	Address	Salary
2	Kamal	25	Pune	1500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.0

SQL: LIKE OPERATOR

- The **LIKE operator** is used in a **WHERE clause** to search for a specified **pattern** in a column.
- The **LIKE operator** is **used to match string pattern values using two wildcard characters.**
- ***Wildcard Character***: *It is used to substitute one or more characters in a string*
- There are **two wildcards used** in conjunction with the **LIKE operator.**
  - **The percent sign (%)**: It represents **zero, one or multiple characters.**
  - **The underscore (_)**: It represents only **a single character.**
- The **percent sign (%)** and the **underscore (_)** can also be used in **combinations!**

Syntax:

```
SELECT column1, column2, ...
FROM table_name
WHERE columnN LIKE pattern;
```

Example 1:

It selects all Employees with a *Name* starting with "K"

```
SELECT * FROM Emp
WHERE Name LIKE "K%";
```

Emp_ID	Name	Age	Address	Salary
2	Kamal	25	Pune	1500
3	Karan	23	Pune	2000.00
6	Kajal	22	Bilaspur	4500.00

Example 2:

It selects all Employees with a *Name* that have "a" in second position

```
SELECT * FROM Emp
WHERE Name LIKE "_a%";
```

Emp_ID	Name	Age	Address	Salary
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

Example 3:

It selects all Employees with a *Name* starting with "M" and are at least 4 characters in length

```
SELECT * FROM Emp
WHERE Name LIKE "M_ _ _%";
```

Emp_ID	Name	Age	Address	Salary
7	Mahi	24	Patna	10000.00

Example 4:

It selects all records from Emp where salary starts with 200

```
SELECT * FROM Emp
WHERE Salary LIKE "200%";
```

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
3	Karan	23	Pune	2000.00

Examples : LIKE operator with wildcard operator (% , _):

LIKE Operator	Description
WHERE Name LIKE 'A%'	Finds any values that start with "A", Eg: Aman, Aakriti
WHERE Name LIKE '%a'	Finds any values that end with "a", Eg: Neha, Soma
WHERE Name LIKE '%oh%'	Finds any values that have "oh" in any position, Eg: Sohan, Mohan
WHERE Name LIKE '_r%'	Finds any values that have "r" in the second position, Eg: Priya, Priyesh
WHERE Name LIKE 'K_ _ _ _%'	Finds any values that start with "K" and are at least 5 characters in length, Eg: Kamal, Karan, Kajal
WHERE Name LIKE 'A_ _%'	Find any values that start with "A" and are at least 3 characters in length, Eg: Aki, Anu
WHERE Name LIKE 'A%N'	Finds any values that start with "A" and ends with "N", Eg: Aman, Raman

SQL NULL Values

- Attributes can have **Null values**, if permitted by the schema definition for a table (i.e., no ***NOT NULL Constraint***),
- **NULL** represents a ***missing value, unknown value, non-existent or non-applicable value.***
- A **NULL value** in a table is a value in a field that appears to be **blank (empty)** i.e. with **no value.**

➢ **Note**: A **NULL value** is **different** than a **zero** value or a field that contains **spaces.**

➢ **A field with a NULL value is one that has been left blank during record creation!**

Emp_ID	Name	Age	Address	Salary	Email
1	Rahul	32	Delhi	2000.00	rr@gmail.com
2	Kamal	25	Pune	1500.00	kk@gmail.com
3	Karan	23	Pune	2000.00	NULL
4	Chirag	25	Mumbai	6500.00	cc@gmail.com
5	Harsh	32	Mumbai	8500.00	hh@gmail.com
6	Kajal	22	Bilaspur	4500.00	NULL
7	Mahi	24	Patna	10000.00	mm@gmail.com

SQL NULL Values...

- Result of any **arithmetic expression** involving ***null*** is ***null***
- Result of **where** clause condition is ***false*** if it evaluates to ***null***.
- **and, or, not operators** handles ***null*** as follows:

and	true	false	null
true	true	false	null
null	null	false	null
false	false	false	false

or	true	false	null
true	true	true	true
null	true	null	null
false	true	false	null

not	
true	false
null	null
false	true

How to test NULL Values?

- It is **not possible to test for NULL values** with ***comparison operators***, such as =, <, or <>.
- For **comparison**, **SQL** uses the **IS NULL** and **IS NOT NULL operators** instead.

SQL: IS NULL OPERATOR

- The **IS NULL** operator is used to check for **NULL** values (or empty values).
- It allows us to find out the set of records in which value for a particular column is **NULL**
- It returns **TRUE** if a NULL value is found, otherwise it returns **FALSE**.
- It can be used in a **SELECT**, **UPDATE**, or **DELETE** statement with **Where** Clause.

➢ **IS NULL Syntax:**

```
SELECT column_names
FROM table_name
WHERE column_name IS NULL;
```

Example: IS NULL OPERATOR

List employees who have no associated email Ids

```
SELECT *
FROM Emp
WHERE Email IS NULL;
```

Emp Table

Emp_ID	Name	Age	Address	Salary	Email
1	Rahul	32	Delhi	2000.00	rr@gmail.com
2	Kamal	25	Pune	1500.00	kk@gmail.com
3	Karan	23	Pune	2000.00	NULL
4	Chirag	25	Mumbai	6500.00	cc@gmail.com
5	Harsh	32	Mumbai	8500.00	hh@gmail.com
6	Kajal	22	Bilaspur	4500.00	NULL
7	Mahi	24	Patna	10000.00	mm@gmail.com

Emp_ID	Name	Age	Address	Salary	Email
3	Karan	23	Pune	2000.00	NULL
6	Kajal	22	Bilaspur	4500.00	NULL

SQL: IS NOT NULL OPERATOR

- **IS NOT NULL** operator is **opposite** of **IS NULL** operator
- The **IS NOT NULL operator** is **used to check for NOT NULL values (or non-empty values**
  - It returns **TRUE** if a NULL value is not found, otherwise it returns **FALSE.**
  - It can be used in a **SELECT, UPDATE,** or **DELETE** statement with **Where** Clause.
  - **IS NOT NULL Syntax:**

```
SELECT column_names
FROM table_name
WHERE column_name IS NOT NULL;
```

SQL: IS NOT NULL OPERATOR

List employees who have associated email Ids

```
SELECT *
FROM Emp
WHERE Email IS NOT NULL;
```

Emp_ID	Name	Age	Address	Salary	Email
1	Rahul	32	Delhi	2000.00	rr@gmail.com
2	Kamal	25	Pune	1500.00	kk@gmail.com
4	Chirag	25	Mumbai	6500.00	cc@gmail.com
5	Harsh	32	Mumbai	8500.00	hh@gmail.com
7	Mahi	24	Patna	10000.00	mm@gmail.com

Emp Table

Emp_ID	Name	Age	Address	Salary	Email
1	Rahul	32	Delhi	2000.00	rr@gmail.com
2	Kamal	25	Pune	1500.00	kk@gmail.com
3	Karan	23	Pune	2000.00	NULL
4	Chirag	25	Mumbai	6500.00	cc@gmail.com
5	Harsh	32	Mumbai	8500.00	hh@gmail.com
6	Kajal	22	Bilaspur	4500.00	NULL
7	Mahi	24	Patna	10000.00	mm@gmail.com

SQL Aliases

- **SQL ALIASES** can be used to create a temporary name for **columns** or **tables.**
- **SQL Aliases** is used to give an alias name to a table or a column
  - **COLUMN ALIASES** are used to make column headings in result set easier to read.
  - **TABLE ALIASES** are used to shorten SQL query to make it easier to read or when there are more than one table is involved.
- SQL Aliases:
  - Aliases are created to make table or column names easier to read
  - An **alias** only exists for the duration of the query. i.e. The renaming is just a temporary change and the actual table name does not change in the database.
  - Aliases are useful when table or column names are big or not very readable.
  - These are preferred when there are **more than one** table involved in a query.

Syntax: SQL Aliases

- **Column Alias Syntax:**

```
SELECT column_name AS alias_name
FROM table_name;
```

- **Table Alias Syntax:**

```
SELECT column_name(s)
FROM table_name AS alias_name;
```

Example: Column Alias

Emp

Emp_ID	Name	Age	Address	Salary	Dept_Id
1	Rahul	32	Delhi	2000.00	3
2	Kamal	25	Pune	1500.00	3
3	Karan	23	Pune	2000.00	3
4	Chirag	25	Mumbai	6500.00	1
5	Harsh	32	Mumbai	8500.00	1
6	Kajal	22	Bilaspur	4500.00	1
7	Mahi	24	Patna	10000.00	1

```
SELECT Emp_ID AS ID, Name
FROM Emp;
```

ID	Name
1	Rahul
2	Kamal
3	Karan
4	Chirag
5	Harsh
6	Kajal
7	Mahi

Example: Table Alias

Emp

Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	3
2	Kamal	25	Pune	1500.00	NULL
3	Karan	23	Pune	2000.00	3
4	Chirag	25	Mumbai	6500.00	1
5	Harsh	32	Mumbai	8500.00	1
6	Kajal	22	Bilaspur	4500.00	NULL
7	Mahi	24	Patna	10000.00	NULL

Dept

Dept_ID	D_Name
1	Sales
2	HR
3	Finance
4	Marketing

```
SELECT E. Emp_ID, E.Name,D.Dept_ID, D.D_Name
FROM Emp AS E, Dept AS D
WHERE E. Dept_ID=D.Dept_ID;
```

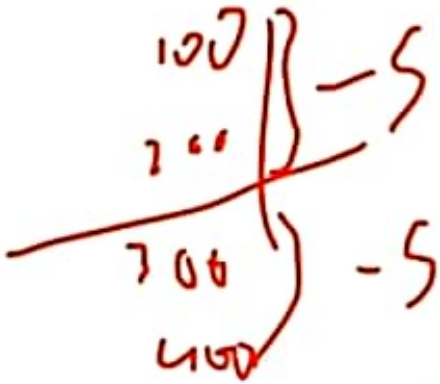
Emp_ID	Name	Dept_Id	D_Name
1	Rahul	3	Finance
3	Karan	3	Finance
4	Chirag	1	Sales
5	Harsh	1	Sales

SQL Functions

- SQL provides many built-in functions to perform operations on data.
- These functions are useful while performing mathematical calculations, string concatenations, sub-strings etc.
- SQL functions are divided into two categories:
  1. **Aggregate Functions**
  2. **Scalar Functions**
- **Aggregate functions and Scalar functions both** return a *single value.*
- **Aggregate functions** operate on *many records* while **Scalar functions** operate on *each record independently*

Aggregate Functions in SQL

- **Aggregate functions** performs calculations on set of values and **returns a single value.**
  1. **COUNT()**
  2. **SUM()**
  3. **AVG()**
  4. **MAX()**
  5. **MIN()**



- **Aggregate functions** are often used by **GROUP BY** clause of the **SELECT** statement ***to group multiple rows together as input to form a single value output***
- **Aggregate functions** ignore NULL values, except count(*)

1. COUNT() Function

- **COUNT()**: **Count function** returns the number of rows that matches the specified criterion
- **Syntax**:

```
SELECT COUNT(column_name)
FROM table_name
WHERE condition;
```

Example: COUNT()

Student Table

ID	Name	Marks
1	A	90
2	B	40
3	C	70
4	D	60
5	E	70
6	F	NULL

- ***Find the total number of student records***

  COUNT(*) returns total no. of records

```
SELECT COUNT(*)
FROM Student;
```

COUNT(*)
6

- ***Find the count of entered marks***

  COUNT(Marks) returns no. of non null values over column Marks

```
SELECT COUNT(Marks)
FROM Student;
```

COUNT(Marks)
5

- ***Find the count of distinct entered marks***

  COUNT(DISTINCT Marks) returns no. of distinct non null values over column Marks

```
SELECT COUNT(DISTINCT Marks)
FROM Student;
```

COUNT(DISTINCT Marks)
4

2. SUM() Function

- **SUM()**: **Sum function** returns total sum of a selected **numeric** columns.
- **Syntax**:

```
SELECT SUM(column_name)
FROM table_name
WHERE condition;
```

Example: SUM()

Student Table

ID	Name	Marks
1	A	90
2	B	40
3	C	70
4	D	60
5	E	70
6	F	NULL

- ***Find the sum of student marks***

SUM(Marks) *sum all non null values of column marks*

```
SELECT SUM(Marks)
FROM Student;
```

SUM(Marks)
330

- ***Find the distinct sum of student marks***

SUM(DISTINCT Marks) *sum all distinct non null values of column marks*

```
SELECT SUM(DISTINCT Marks)
FROM Student;
```

SUM(DISTINCT Marks)
260

Using 'AS' for Renaming the column

```
SELECT SUM(Marks) AS 'Sum Marks'
FROM Student;
```

Sum Marks
330

3. AVG() Function

- **AVG(): Average function** returns the average value of selected **numeric** column.
- **Syntax:**

```
SELECT AVG(column_name)
FROM table_name
WHERE condition;
```

Example: AVG()

Student Table

ID	Name	Marks
1	A	90
2	B	40
3	C	70
4	D	60
5	E	70
6	F	NULL

- *Find the average marks of students*

```
SELECT AVG(Marks)
FROM Student;
```

AVG(Marks)
66

AVG(Marks) = SUM(Marks)/COUNT(Marks)= 330/5 =66

- *Find the distinct average marks of students*

```
SELECT AVG(DISTINCT Marks)
FROM Student;
```

AVG(DISTINCT Marks)
65

AVG(DISTINCT Marks) = SUM(DISTINCT Marks)/COUNT(DISTINCT Marks)= 260/4 =65

Using 'AS' for Renaming the column

```
SELECT AVG(Marks) AS 'Average Marks'
FROM Student;
```

Average Marks
66

4. MAX() Function

- MAX(): Maximum function returns maximum value of selected column
- Syntax:

```
SELECT MAX(column_name)
FROM table_name
WHERE condition;
```

Example: MAX()

- *Find the maximum student marks:* **MAX(Marks)**

```
SELECT MAX(Marks)
FROM Student;
```

MAX(Marks)
90

Student Table

ID	Name	Marks
1	A	90
2	B	40
3	C	70
4	D	60
5	E	70
6	F	NULL

5. MIN() Function

- **MIN(): Minimum function** returns minimum value of selected column.
- **Syntax:**

```
SELECT MIN(column_name)
FROM table_name
WHERE condition;
```

Example: MIN()

- *Find the minimum students marks:* MIN(Marks)

```
SELECT MIN(Marks)
FROM Student;
```

MIN(Marks)
40

Student Table

ID	Name	Marks
1	A	90
2	B	40
3	C	70
4	D	60
5	E	70
6	F	NULL

SQL Functions

- SQL provides many built-in functions to perform operations on data.
- These functions are useful while performing mathematical calculations, string concatenations, sub-strings etc.
- SQL functions are divided into two categories:
  1. **Aggregate Functions**
  2. **Scalar Functions**

- Aggregate functions and Scalar functions **both** return a single value.
- **Aggregate functions** operate on many records while **Scalar functions** operate on each record independently

Scalar Functions

- Scalar functions return a single value from an input value.
  1. **UCASE()**
  2. **LCASE()**
  3. **MID()**
  4. **LENGTH()**
  5. **ROUND()**
  6. **NOW()**
  7. **FORMAT()**

1. UCASE() Function

- **UCASE(): Upper case function** is used to convert value of string column to uppercase characters.
- **Syntax:**

```
SELECT UCASE(column_name)
FROM table_name;
```

2. LCASE() Function

- **LCASE(): Lower case function** is used to convert value of string column to lower case characters.
- **Syntax:**

```
SELECT LCASE(column_name)
FROM table_name;
```

Example: UCASE()

- *Converting string values of name column to upper case*

```
SELECT UCASE(Name)
FROM Emp;
```

UCASE(Name)
RAHUL
KAMAL
KARAN
CHIRAG
HARSH
KAJAL
MAHI

Emp Table

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

2. LCASE() Function

- **LCASE(): Lower case function** is used to convert value of string column to lower case characters.
- **Syntax:**

```
SELECT LCASE(column_name)
FROM table_name;
```

Example: LCASE()

- *Converting string values of name column to lower case*

```
SELECT LCASE(Name)
FROM Emp;
```

LCASE(Name)
rahul
kamal
karan
chirag
harsh
kajal
mahi

Emp Table

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

3. MID() Function

- **MID(): MID function** is used to extract substrings from column values of string type in a table.
- **Syntax:**

```
SELECT MID(column_name, start, length)
FROM table_name;
```

Note:

Length is optional

Start signifies the starting position of string

Example: MID()

- *Extract sub-string from name column from second position having length of 3*

```
SELECT MID(Name,2,3)
FROM Emp;
```

MID(Name,2,3)
ahu
ama
ara
hir
ars
aja
ahi

Emp Table

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

Example: MID()

- *Extract sub-string from name column from second position having length of 3*

```
SELECT MID(Name,2,3)
FROM Emp;
```

MID(Name,2,3)
ahu
ama
ara
hir
ars
aja
ahi

Substr

Emp Table

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

4. LENGTH() Function

- **LENGTH()**: **Length function** returns the length of a string in the column.
- **Syntax**:

```
SELECT LENGTH(column_name)
FROM table_name;
```

Example: LENGTH()

Find the length of string of name column

```
SELECT LENGTH(Name)
FROM Emp;
```

LEN(Name)
5
5
5
6
5
5
4

Emp Table

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

5. ROUND() Function

- **ROUND()**:The **ROUND() function** is used to round a **numeric** column to the number of decimals specified.
- **Syntax:**

```
SELECT ROUND(column_name, decimals)
FROM table_name;
```

Note:
Decimals represents **number of decimals to be fetched.**

Example: ROUND()

- *Find the round-off salary of employees*

```
SELECT ROUND(Salary)
FROM Emp;
```

or

```
SELECT ROUND(Salary,0)
FROM Emp;
```

ROUND(Salary)
2001
1500
2000
6501
8500
4500
10000

Emp Table

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.50
2	Kamal	25	Pune	1500.20
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.80
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

Example: ROUND()

- *Find round-off salary of employees (upto 1 decimal place)*

```
SELECT ROUND(Salary,1)
FROM Emp;
```

ROUND(Salary,1)
2001.5
1500.2
2000.0
6501.8
8500.0
4500.0
10000.0

Emp Table

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.50
2	Kamal	25	Pune	1500.20
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.80
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

6. NOW() Function

- **NOW()**: The **NOW() function** returns the **current system date and time.**
- **Syntax:**

```
SELECT NOW()
FROM table_name;
```

Example: NOW()

- *Fetching current system date & time*

```
SELECT NOW() ;
```

NOW()
2021-01-25 13:40:25

- *Fetching current system date & time in a column*

```
SELECT Name, NOW() AS 'Date & Time'
FROM Emp;
```

Name	Date & Time
Rahul	2021-01-25 13:40:25
Kamal	2021-01-25 13:40:25
Karan	2021-01-25 13:40:25
Chirag	2021-01-25 13:40:25
Harsh	2021-01-25 13:40:25
Kajal	2021-01-25 13:40:25
Mahi	2021-01-25 13:40:25

7. FORMAT() Function

- **FORMAT()**: The **FORMAT()** function is used to format how a column is to be displayed.
- **Syntax:**

```
SELECT FORMAT(column_name, format)
FROM table_name;
```

Example: FORMAT()

- *Formatting current date as 'YYYY-MM-DD'*

```
SELECT Name, FORMAT(NOW(),'YYYY-MM-DD') AS 'Date'
FROM Emp;
```

Name	Date
Rahul	2021-01-25
Kamal	2021-01-25
Karan	2021-01-25
Chirag	2021-01-25
Harsh	2021-01-25
Kajal	2021-01-25
Mahi	2021-01-25

ORDER BY Clause

- The output of the **SELECT** queries do not have any specific order. *The* ***ORDER BY Clause allows us to specify order for the query output.***
- **ORDER BY Clause** is used with **SELECT statement** for arranging retrieved data in sorted order.
- The **ORDER BY Clause** by default sorts the retrieved data in **ascending order** [**ASC**].
- To sort the data in **descending order**, **DESC** keyword is used with ORDER BY clause.
- **Syntax**:

```
SELECT column-list
FROM table_name
[WHERE conditions]
ORDER BY column1, column2, .. columnN [ASC | DESC];
```

Example Table: Emp

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

Query 1 : (Eg: Ascending Order - Default)

- To display details of Emp table in ascending order by *salary*

```
SELECT *
FROM Emp
ORDER BY Salary;
```

Emp_ID	Name	Age	Address	Salary
2	Kamal	25	Pune	1500.00
1	Rahul	32	Delhi	2000.00
3	Karan	23	Pune	2000.00
6	Kajal	22	Bilaspur	4500.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
7	Mahi	24	Patna	10000.00

Query 2: (Eg: Ascending Order with multiple columns)

- **To sort details of Emp table in ascending order by *name* and *salary***

```
SELECT *
FROM Emp
ORDER BY Name, Salary;
```

Emp_ID	Name	Age	Address	Salary
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
7	Mahi	24	Patna	10000.00
1	Rahul	32	Delhi	2000.00

In ORDER BY Clause multiple columns can be used.

In this case

1st column is used as **primary ordering** field,

2nd column is used as **secondary ordering** field,

and so on

Query 3: (Eg: Descending Order)

- To sort details of Emp table in descending order by name

```
SELECT *
FROM Emp
ORDER BY Name DESC;
```

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
7	Mahi	24	Patna	10000.00
3	Karan	23	Pune	2000.00
2	Kamal	25	Pune	1500.00
6	Kajal	22	Bilaspur	4500.00
5	Harsh	32	Mumbai	8500.00
4	Chirag	25	Mumbai	6500.00

GROUP BY Clause

- In SQL, the **GROUP BY** statement is used for organizing similar data into groups.
  - For Eg: "find the number of customers in each country".
- The **GROUP BY** statement is often used with **aggregate functions** (COUNT, MAX, MIN, SUM, AVG) to group the result-set by one or more columns.
- **Imp Points:**
  - **GROUP BY** clause is **used with** the **SELECT** statement in the **SQL** query.
  - **GROUP BY** clause is **placed after** the **WHERE** clause in **SQL.**
  - **GROUP BY** clause is **placed before** the **ORDER BY** clause in **SQL.**

Syntax:

```
SELECT column_name(s), aggregate_function(column_name)
FROM table_name
[WHERE condition]
GROUP BY column_name(s)
[ORDER BY column_name(s)];
```

Example Table: Emp

Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	500
2	Kamal	25	Pune	1500.00	500
3	Karan	23	Pune	2000.00	500
4	Chirag	25	Mumbai	6500.00	501
5	Harsh	32	Mumbai	8500.00	501
6	Kajal	22	Bilaspur	4500.00	502
7	Mahi	24	Patna	10000.00	502

Query 1:

(GROUP BY with COUNT aggregate function)

Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	500
2	Kamal	25	Pune	1500.00	500
3	Karan	23	Pune	2000.00	500
4	Chirag	25	Mumbai	6500.00	501
5	Harsh	32	Mumbai	8500.00	501
6	Kajal	22	Bilaspur	4500.00	502
7	Mahi	24	Patna	10000.00	502

Write a query to find the number of employee in each department

```
SELECT Dept_id, COUNT(*)
FROM Emp
GROUP BY Dept_id
```

Dept_ID	COUNT(*)
500	3
501	2
502	2

Query 2:

(GROUP BY with COUNT and ORDER BY)

Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	500
2	Kamal	25	Pune	1500.00	500
3	Karan	23	Pune	2000.00	500
4	Chirag	25	Mumbai	6500.00	501
5	Harsh	32	Mumbai	8500.00	501
6	Kajal	22	Bilaspur	4500.00	502
7	Mahi	24	Patna	10000.00	502

Write a query to find the number of employee in each department, sorted low to high

```
SELECT Dept_id, COUNT(*)
FROM Emp
GROUP BY Dept_id
ORDER BY COUNT(*);
```

Dept_ID	COUNT(*)
501	2
502	2
500	3

Query 3:

(GROUP BY with SUM aggregate function)

Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	500
2	Kamal	25	Pune	1500.00	500
3	Karan	23	Pune	2000.00	500
4	Chirag	25	Mumbai	6500.00	501
5	Harsh	32	Mumbai	8500.00	501
6	Kajal	22	Bilaspur	4500.00	502
7	Mahi	24	Patna	10000.00	502

Write a query to find the sum of salaries of employees from each department

```
SELECT Dept_id, SUM(Salary) AS 'SumSalary'
FROM Emp
GROUP BY Dept_id;
```

Dept_ID	SumSalary
500	5500.00
501	15000.00
502	14500.00

Query 4:

(GROUP BY with AVG aggregate function)

Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	500
2	Kamal	25	Pune	1500.00	500
3	Karan	23	Pune	2000.00	500
4	Chirag	25	Mumbai	6500.00	501
5	Harsh	32	Mumbai	8500.00	501
6	Kajal	22	Bilaspur	4500.00	502
7	Mahi	24	Patna	10000.00	502

Write a query to find the average of salaries of employees from each department

```
SELECT Dept_id, ROUND(AVG(Salary))
FROM Emp
GROUP BY Dept_id;
```

Dept_ID	AVG(Salary)
500	1833
501	7500
502	7250

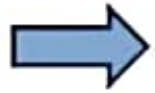
Query 5:

(GROUP BY with MIN aggregate function)

Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	500
2	Kamal	25	Pune	1500.00	500
3	Karan	23	Pune	2000.00	500
4	Chirag	25	Mumbai	6500.00	501
5	Harsh	32	Mumbai	8500.00	501
6	Kajal	22	Bilaspur	4500.00	502
7	Mahi	24	Patna	10000.00	502

Write a query to find the minimum salary of employee from each department

```
SELECT Dept_id, MIN(Salary)
FROM Emp
GROUP BY Dept_id;
```



Dept_ID	MIN(Salary)
500	1500.00
501	6500.00
502	4500.00

Query 6:

(GROUP BY with MAX aggregate function)

Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	500
2	Kamal	25	Pune	1500.00	500
3	Karan	23	Pune	2000.00	500
4	Chirag	25	Mumbai	6500.00	501
5	Harsh	32	Mumbai	8500.00	501
6	Kajal	22	Bilaspur	4500.00	502
7	Mahi	24	Patna	10000.00	502

Write a query to find the maximum salary of employee from each department

```
SELECT Dept_id, MAX(Salary)
FROM Emp
GROUP BY Dept_id;
```

Dept_ID	MAX(Salary)
500	2000.00
501	8500.00
502	10000.00

Query 7:

(GROUP BY with MIN & MAX both)

Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	500
2	Kamal	25	Pune	1500.00	500
3	Karan	23	Pune	2000.00	500
4	Chirag	25	Mumbai	6500.00	501
5	Harsh	32	Mumbai	8500.00	501
6	Kajal	22	Bilaspur	4500.00	502
7	Mahi	24	Patna	10000.00	502

Write a query to find the minimum and maximum salary of employee from each department

```
SELECT Dept_id, MIN(Salary), MAX(Salary)
FROM Emp
GROUP BY Dept_id;
```

Dept_ID	MIN(Salary)	MAX(Salary)
500	1500.00	2000.00
501	6500.00	8500.00
502	4500.00	10000.00

Query 8:

(GROUP BY with multiple columns)

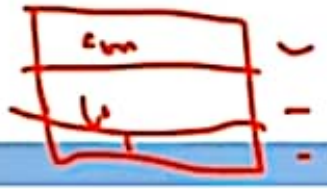
Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	500
2	Kamal	25	Pune	1500.00	500
3	Karan	23	Pune	2000.00	500
4	Chirag	25	Mumbai	6500.00	501
5	Harsh	32	Mumbai	8500.00	501
6	Kajal	22	Bilaspur	4500.00	502
7	Mahi	24	Patna	10000.00	502

Write a query to find the number of employees in each department location wise

```
SELECT Dept_id, Address, COUNT(*)
FROM Emp
GROUP BY Dept_id, Address;
```

Dept_ID	Address	COUNT(*)
500	Delhi	1
500	Pune	2
501	Mumbai	2
502	Bilaspur	1
502	Patna	1

HAVING Clause



- **HAVING** clause is used with **GROUP BY** clause & filter the groups created by the **GROUP BY** clause
- The **HAVING clause** enables you to specify **conditions** that filter which **group** results appear in the results.
- The **HAVING** clause was added to SQL because the *aggregate functions like* ***SUM, AVG, MIN, MAX,COUNT*** cannot be used with **WHERE** clause.
- The **WHERE clause** places conditions on the **selected columns,** whereas the **HAVING** clause places conditions on **groups** created by the **GROUP BY clause**
- In SQL query, **HAVING** clause is placed after the **GROUP BY** clause

Syntax:

```
SELECT column_name(s), aggregate_function(column_name)
FROM table_name
[WHERE condition]
GROUP BY column_name(s)
HAVING condition
[ORDER BY column_name(s)];
```

Order of Clauses(in SQL):

1. SELECT
2. FROM
3. WHERE
4. GROUP BY
5. HAVING
6. ORDER BY

Query 1:

(GROUP BY with COUNT aggregate function)

Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	500
2	Kamal	25	Pune	1500.00	500
3	Karan	23	Pune	2000.00	500
4	Chirag	25	Mumbai	6500.00	501
5	Harsh	32	Mumbai	8500.00	501
6	Kajal	22	Bilaspur	4500.00	502
7	Mahi	24	Patna	10000.00	502

Write a query to display departments where the number of employee is greater than 2

```
SELECT Dept_id, COUNT(*)
FROM Emp
GROUP BY Dept_id
HAVING COUNT(*) >2;
```

Example Table: Emp

Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	500
2	Kamal	25	Pune	1500.00	500
3	Karan	23	Pune	2000.00	500
4	Chirag	25	Mumbai	6500.00	501
5	Harsh	32	Mumbai	8500.00	501
6	Kajal	22	Bilaspur	4500.00	502
7	Mahi	24	Patna	10000.00	502

Query 1:

Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	500
2	Kamal	25	Pune	1500.00	500
3	Karan	23	Pune	2000.00	500
4	Chirag	25	Mumbai	6500.00	501
5	Harsh	32	Mumbai	8500.00	501
6	Kajal	22	Bilaspur	4500.00	502
7	Mahi	24	Patna	10000.00	502

Write a query to display departments where the number of employee is greater than 2

```
SELECT Dept_id, COUNT(*)
FROM Emp
GROUP BY Dept_id
HAVING COUNT(*) >2;
```

Dept_ID	COUNT(*)
500	3

Query 2:

Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	500
2	Kamal	25	Pune	1500.00	500
3	Karan	23	Pune	2000.00	500
4	Chirag	25	Mumbai	6500.00	501
5	Harsh	32	Mumbai	8500.00	501
6	Kajal	22	Bilaspur	4500.00	502
7	Mahi	24	Patna	10000.00	502

Write a query to find the department for which the sum of salaries of employees is more than 10000

```
SELECT Dept_ID, SUM(Salary)
FROM Emp
GROUP BY Dept_id
HAVING SUM(Salary)>10000;
```

Dept_ID	SUM(Salary)
501	15000.00
502	14500.00

Query 3:

Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	500
2	Kamal	25	Pune	1500.00	500
3	Karan	23	Pune	2000.00	500
4	Chirag	25	Mumbai	6500.00	501
5	Harsh	32	Mumbai	8500.00	501
6	Kajal	22	Bilaspur	4500.00	502
7	Mahi	24	Patna	10000.00	502

Write a query to find the sum of salaries of employee that lives at same location

```
SELECT Address, SUM(Salary)
FROM Emp
GROUP BY Address
HAVING COUNT(Address)>=2;
```

Address	SUM(Salary)
Mumbai	15000.00
Pune	3500.00