1. Discuss about views in relational model with example.

**Views**provide a way to represent a subset of a real table, selecting certain columns or rows from an ordinary table. They are particularly useful for restricting access to the original table, allowing users to see only a specific portion of it. The table from which a view is created is known as the **base table**. PostgreSQL supports both updatable and non-updatable views.

Views in a relational database can be categorized into **updatable views** and **non-updatable views**, depending on whether they allow modifications (INSERT, UPDATE, DELETE) on the underlying base table.

**Updatable views**

### ****Conditions for an Updatable View****

1. The view must be based on a **single table**.
2. The view should not contain **aggregations** (SUM, AVG, COUNT, etc.).
3. It should not include **JOINs, GROUP BY, DISTINCT, or HAVING**.
4. The view should contain **all NOT NULL columns** from the base table if those columns do not have default values.

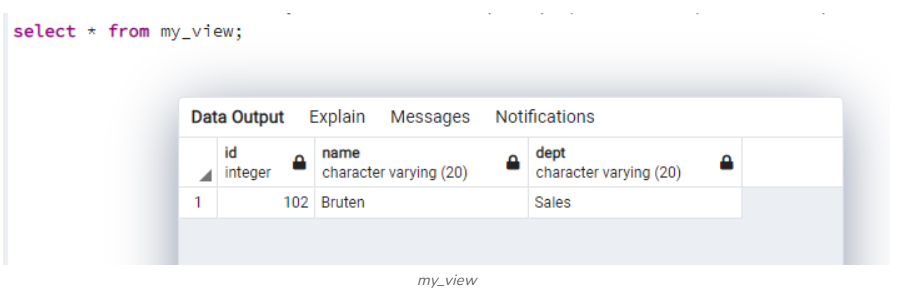
* There should be only one entry in the **FROM clause**of the defining query of the view
* The selection list must not contain any aggregate function such as ‘[**SUM**](https://www.geeksforgeeks.org/postgresql-sum-function/)**‘**, ‘[**MIN**](https://www.geeksforgeeks.org/postgresql-min-function/)**‘**, ‘[**MAX**](https://www.geeksforgeeks.org/postgresql-max-function/)**‘**, etc.
* The query defining the view must not include**‘GROUP BY'**, ‘**HAVING'**, ‘**LIMIT'**, ‘**OFFSET'**, ‘**DISTINCT'**, **'WITH'**, ‘**UNION'**, ‘**INTERSECT'**, or ‘**EXCEPT'**statements.
* **Creating an updatable view**
* Now we can create a view from the original table “**example**“, A view can take one or more than one column in the selection list depending on how much access you want to give to your users. Let us create an updatable view “**my\_view**” with three columns ‘**id’**, ‘**name’**, and ‘**dept’**respectively.
* **CREATE OR REPLACE** VIEW my\_view **AS**
* **SELECT** id,name,dept from example
* **WHERE** dept='Sales';

### Querying the results

We can simply see the results of the created view by simply running a **SELECT**query as follows:

**SELECT \* FROM** my\_view;

**Output**:

****

**Explanation:** The result should show only the rows where dept is ‘**Sales**‘. If there was a single row with ‘**dept**‘ as ‘**Sales**‘, you would get only that row in the results.

### Inserting in the created view

Now let us try to perform the **INSERT**operation in the created view using the following syntax and example as well.

**Syntax:**

**INSERT** **INTO** view\_name (column1,column2,...columnN)

**VALUES**(Val1, val2...valN);

**Example**:

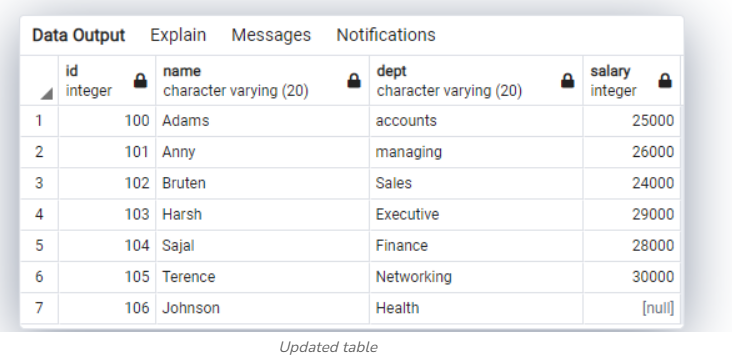
**INSERT INTO** my\_view (id,name,dept)

**VALUES**(106,'Johnson','Health');

Running the **SELECT**query on **my\_view**will show no change because the new dept value is ‘**Health’**, which does not match the ‘**WHERE'**clause condition (dept = 'Sales'). However, this new row will be added to the original table ‘**example'**.

To verify, run the query:

**SELECT \* FROM** example;

****

**Explanation:** The output will show the new row added to the ‘**example'**table, with a **NULL**value in the ‘**dept'**column for the new row and an incremented total number of rows. This demonstrates that the updatable view has been successfully created.

### Updating the created view

We can also update the created view using the following syntax :

**Syntax**:

**UPDATE** view\_name **SET** column = "New Value";

**Example**:

**UPDATE** my\_view **SET** dept = "Health";

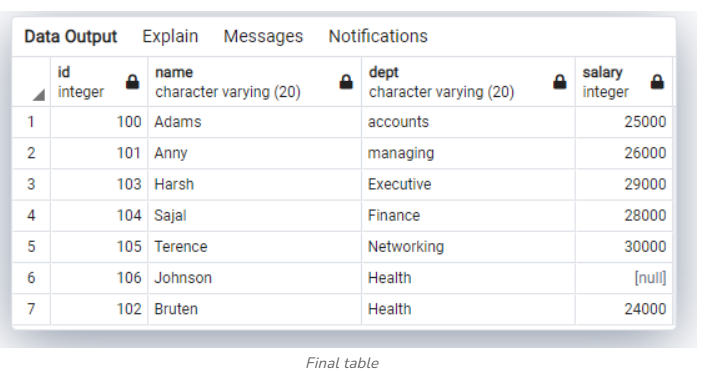
**Output**:

**UPDATE** VIEW

Query returned successfully **in** 180 msec.

To verify the update, run the query:

**SELECT \* FROM** example;

****

**Explanation:** The output will show that the row previously with dept as ‘**Sales**‘ has been updated to ‘**Health**‘. The order of the rows may have changed, with the updated row now appearing in the last position.

### ****Example 2:****

#### **Base Table: Employee**

| **EmpID** | **Name** | **Department** | **Salary** |
| --- | --- | --- | --- |
| 101 | Alice | HR | 50000 |
| 102 | Bob | IT | 70000 |
| 103 | Carol | IT | 65000 |

#### **Creating an Updatable View**

CREATE VIEW IT\_Employees AS

SELECT EmpID, Name, Salary

FROM Employee

WHERE Department = 'IT';

#### **Updating Data Through View**

UPDATE IT\_Employees

SET Salary = 75000

WHERE EmpID = 102;

**Effect:** This updates Bob's salary in the **Employee** table as well.

## **2. Non-Updatable View**

A **non-updatable view** does not allow modifications to the base table because it involves complex operations that make updates ambiguous.

### ****Conditions for a Non-Updatable View****

A view becomes **non-updatable** if it includes:

1. **Joins** between multiple tables.
2. **Aggregated data** (SUM, AVG, COUNT).
3. **DISTINCT, GROUP BY, HAVING, or UNION** clauses.
4. **Derived columns or computed expressions**.

### ****Example of a Non-Updatable View****

#### **Creating a View with Aggregation**

CREATE VIEW Avg\_Salary AS

SELECT Department, AVG(Salary) AS Avg\_Salary

FROM Employee

GROUP BY Department;

#### **Trying to Update the View**

UPDATE Avg\_Salary

SET Avg\_Salary = 60000

WHERE Department = 'IT';

**Error:** This operation is not allowed because Avg\_Salary is a derived (calculated) column.

1. What is Join? Explain different types of Joins with Examples.

* Joins are used to fetch the information from multiple tables.
* Joins are used to combine two or more tables with a common column into a single set of records.

Types of joins:

1.Inner join or equi join

2.Non-equi join

3.Left outer join

4.Right outer join

5.Full outer join

6. Cartision product join

7. self join or recursive join

**1.Inner join or equi join:** A join in whichthe joining condition is based on equality between values in the common column.

Student: Course:

|  |  |  |
| --- | --- | --- |
| **Rno** | **Sname** | **Cid** |
| 1 | Vijay | 10 |
| 2 | Ajay | 20 |
| 3 | Sanjay | 30 |

|  |  |
| --- | --- |
| **Cid** | **Cname** |
| 10 | C |
| 30 | Java |
| 40 | C++ |

Ex: Select sname,cname from student inner join course student.cid=course.cid;

|  |  |
| --- | --- |
| **Sname** | **Cname** |
| Vijay | C |
| sanjay | Java |

**2.Non-equi join:** A join in whichthe joining condition is based on non equality between values in the common column.

Ex: Select sname,cname from student inner join course student.cid!=course.cid;

|  |  |
| --- | --- |
| **Sname** | **Cname** |
| Vijay | Java |
| Vijay | C++ |
| Ajay | C |
| Ajay | Java |
| Ajay | C++ |
| Sanjay | C |
| Sanjay | C++ |

**3.Left outer join:** It gives records from left table even if there are no matching values in the right table

Ex: Select sname,cname from student left join course student.cid!=course.cid;

|  |  |
| --- | --- |
| **Sname** | **Cname** |
| Vijay | C |
| Ajay | Null |
| Sanjay | Java |

**4.Right outer join:** It gives records from left table even if there are no matching values in the right table

Ex: Select sname,cname from student right join course student.cid!=course.cid;

|  |  |
| --- | --- |
| **Sname** | **Cname** |
| Vijay | C |
| Sanjay | Java |
| Null | C++ |

**4. Full outer join:** It gives records from left table even if ther are no matching values in the right table

Ex: Select sname,cname from student full join course student.cid!=course.cid;

|  |  |
| --- | --- |
| **Sname** | **Cname** |
| Vijay | C |
| Ajay | Null |
| Sanjay | Java |
| Null | C++ |

**6.Cartision product join:** A join in which all possible combinations of all the rows of first table with each row of the second table.

Ex: select sname,cname from student,course;

|  |  |
| --- | --- |
| **Sname** | **Cname** |
| Vijay | C |
| Vijay | Java |
| Vijay | C++ |
| Ajay | C |
| Ajay | Java |
| Ajay | C++ |
| Sanjay | C |
| Sanjay | Java |
| Sanjay | C++ |

**7.Self join or recursive join:** It is a join in which a table is joined to itself where joining condition is based on column of same table.

Emp E1: Emp E2:

|  |  |  |
| --- | --- | --- |
| **Eid** | **Ename** | **Mid** |
| 10 | John | 40 |
| 20 | Karthi | 30 |
| 30 | Virat | 20 |
| 40 | Arun | 10 |

|  |  |  |
| --- | --- | --- |
| **Eid** | **Ename** | **Mid** |
| 10 | John | 40 |
| 20 | Karthi | 30 |
| 30 | Virat | 20 |
| 40 | Arun | 10 |

Ex:select E1.ename as ename, E2.ename as mname from Emp E1, Emp E2 where E1.Eid=E.Mid;

|  |  |
| --- | --- |
| **Ename** | **Mname** |
| John | Arun |
| Karthi | Virat |
| Virat | Karthi |
| Arun | John |

1. What SQL construct enables the definition of a relation? What constructs allow modification of relation instances?

## **1. SQL Construct for Defining a Relation**

A **relation** in SQL is defined using the **CREATE TABLE** statement. This statement specifies the structure of the table, including:

* Column names and data types
* Primary and foreign keys
* Constraints (e.g., NOT NULL, UNIQUE)

### ****Syntax of**** CREATE TABLE

sql

Copy

CREATE TABLE table\_name (

column1 datatype constraints,

column2 datatype constraints,

...

);

### ****Example: Defining a Relation (Employee Table)****

sql

Copy

CREATE TABLE Employee (

EmpID INT PRIMARY KEY, -- Unique identifier for each employee

Name VARCHAR(50) NOT NULL, -- Employee name (cannot be NULL)

Department VARCHAR(50), -- Department name

Salary DECIMAL(10,2) CHECK (Salary > 0), -- Salary with a constraint

JoinDate DATE DEFAULT CURRENT\_DATE -- Default to the current date

);

### ****Explanation:****

* **EmpID INT PRIMARY KEY** → Defines EmpID as the primary key, ensuring uniqueness.
* **Name VARCHAR(50) NOT NULL** → Ensures that Name cannot be NULL.
* **Department VARCHAR(50)** → Stores the department name.
* **Salary DECIMAL(10,2) CHECK (Salary > 0)** → Ensures salary is positive.
* **JoinDate DATE DEFAULT CURRENT\_DATE** → Defaults JoinDate to today’s date.

## **2. SQL Constructs for Modifying Relation Instances**

Once a relation (table) is defined, SQL provides three key constructs to modify relation instances (records).

### ****(a)**** INSERT ****– Adding New Records****

The **INSERT** statement is used to add new rows (tuples) into a table.

#### **Syntax:**

sql

Copy

INSERT INTO table\_name (column1, column2, ...)

VALUES (value1, value2, ...);

#### **Example: Adding an Employee**

sql

Copy

INSERT INTO Employee (EmpID, Name, Department, Salary, JoinDate)

VALUES (101, 'Alice', 'HR', 50000, '2024-01-15');

**Effect:** Adds a new employee, Alice, to the Employee table.

### ****(b)**** UPDATE ****– Modifying Existing Records****

The **UPDATE** statement modifies existing records in a table.

#### **Syntax:**

sql

Copy

UPDATE table\_name

SET column1 = value1, column2 = value2

WHERE condition;

#### **Example: Updating Salary**

sql

Copy

UPDATE Employee

SET Salary = 55000

WHERE EmpID = 101;

**Effect:** Updates Alice's salary from **50,000** to **55,000**.

⚠️ **Without a WHERE clause, all rows will be updated!**

### ****(c)**** DELETE ****– Removing Records****

The **DELETE** statement removes records from a table.

#### **Syntax:**

sql

Copy

DELETE FROM table\_name

WHERE condition;

#### **Example: Deleting an Employee**

sql

Copy

DELETE FROM Employee

WHERE EmpID = 101;

**Effect:** Removes Alice’s record from the Employee table.

⚠️ **Without a WHERE clause, all records will be deleted!**

1. i) Find the names of sailors who have reserved boat 103

ii) Find the sid, names of sailors who have reserved a red boat

iii) Find the colors of boats reserved by Lubber

iv) Find the names of the sailors who reserved a green boat and having age greater than 7 in descending order?

1. List out relational set operations. Explain in detail.

* The SQL Set operation is used to combine the two or more SQL SELECT statements.

## Types 1of Set Operation

1. Union
2. Union All
3. Intersect
4. Minus



### 1. Union

* The SQL Union operation is used to combine the result of two or more SQL SELECT queries.
* In the union operation, all the number of data type and columns must be same in both the tables on which UNION operation is being applied.
* The union operation eliminates the duplicate rows from its result set.

**Syntax**

SELECT column\_name FROM table1

UNION

SELECT column\_name FROM table2;

**Example:**

|  |  |  |
| --- | --- | --- |
| **ID** | **NAME** | |
| 1 | | Jack |
| 2 | | Harry |
| 3 | | Jackson |

**The First table**

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 3 | Jackson |
| 4 | Stephan |
| 5 | David |

**The Second table**

Union SQL query will be:

SELECT \* FROM First

UNION

SELECT \* FROM Second;

The result set table will look like:

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | Jack |
| 2 | Harry |
| 3 | Jackson |
| 4 | Stephan |
| 5 | David |

### 2. Union All

* Union All operation is equal to the Union operation. It returns the set without removing duplication and sorting the data.

**Syntax:**

SELECT column\_name FROM table1

UNION ALL

SELECT column\_name FROM table2;

**Example:** Using the above First and Second table.

Union All query will be like:

SELECT \* FROM First

UNION ALL

SELECT \* FROM Second;

The result set table will look like:

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | Jack |
| 2 | Harry |
| 3 | Jackson |
| 3 | Jackson |
| 4 | Stephan |
| 5 | David |

### 3. Intersect

* It is used to combine two SELECT statements. The Intersect operation returns the common rows from both the SELECT statements.
* In the Intersect operation, the number of data type and columns must be the same.
* It has no duplicates and it arranges the data in ascending order by default.

**Syntax**

SELECT column\_name FROM table1

INTERSECT

SELECT column\_name FROM table2;

**Example:**

**Using the above First and Second table.**

Intersect query will be:

SELECT \* FROM First

INTERSECT

SELECT \* FROM Second;

The resultset table will look like:

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 3ygd6w5Q@ | Jackson |

### 4. Minus

* It combines the result of two SELECT statements. Minus operator is used to display the rows which are present in the first query but absent in the second query.
* It has no duplicates and data arranged in ascending order by default.

**Syntax:**

SELECT column\_name FROM table1

MINUS

SELECT column\_name FROM table2;

**Example**

**Using the above First and Second table.**

Minus query will be:

SELECT \* FROM First

MINUS

SELECT \* FROM Second;

The result set table will look like:

|  |  |
| --- | --- |
| **5.** | **NAME** |
| 1 | Jack |
| 2 | Harry |

1. Explain the concept of Destroying/ altering Tables in Views.

**Destroying view**

* Destroy removes tables from the database, or integrity constraints and permissions from a table or view. Only the owner is allowed to destroy a table or its permissions and integrity constraints. Destroying a table destroys all views built on that table.
* If the table being destroyed has secondary indexes, the secondary indexes are also destroyed. You can destroy a secondary index separately, without affecting the base table.
* To destroy individual permissions or constraints for a table, you must use the integer argument. Use the help permit statement (for permissions) or a help integrity statement (for constraints) to display the argument values for the various individual permissions and constraints. To destroy all constraints or permissions, specify all.

Destroy accepts a maximum of 30 arguments. To destroy more than 30 objects, you must use multiple destroy statements.

**Deleting view**

* Now we know how to create simple and complex views but what if we don't need our created views anymore, So we need to delete the view so as we DROP a table in SQL, similarly, we can delete or drop a view using the DROP statement.
* The DROP statement completely deletes the structure of the view.

**Syntax:**DROP VIEW ViewName;

* Here ViewName is the name of the view to be deleted.

**Example:**

* Let's delete one of our created views, say EmpView1.

DROP VIEW EmpView1;

* Let's use the SELECT statement on Deleted View.

SELECT \* from EmpView1;

**Output:**

* Now SQL Editor will give us an error saying the table does not exist as the table is now been deleted.

**ORA-00942: table or view does not exist**

1. **Adding Columns:** Views remain unaffected unless the view needs the new column.
2. **Dropping Columns:**
   * If a dropped column is used in a view, querying the view results in an error.
   * The view must be recreated to exclude the missing column.

#### **Example: Impact of Dropping Column in a View**

##### **Existing View:**

sql

Copy

CREATE VIEW IT\_Employees AS

SELECT EmpID, Name, Salary

FROM Employee

WHERE Department = 'IT';

##### **Dropping the** Salary **Column:**

sql

Copy

ALTER TABLE Employee

DROP COLUMN Salary;

##### **Now, Querying the View:**

sql

Copy

SELECT \* FROM IT\_Employees;

💥 **Error:** "Invalid column name 'Salary'."

**Manipulating Data in a View**

* Views can also be used to manipulate data in the scope of the view table only, manipulating data in a view does not affect the data of the original table. Updateable Views are views that allow for data manipulation but there are certain conditions needed to be taken care of while manipulating the data of the view:
* The GROUP BY and ORDER BY clauses should not be included in the SELECT statement used to generate the view.
* The DISTINCT keyword should not be used in the SELECT statement.
* All NOT NULL values should be present in the View.
* Nested or complex queries should not be used to construct the view.
* A single table should be used to generate the view. We will not be able to update the view if it was constructed using several tables.
* Manipulating data in a view includes inserting or deleting a row from the view table

**Inserting a row in a view**

* Inserting a row in the view takes the same syntax as we use to insert a row in a simple table

**Syntax:**INSERT INTO ViewName(column1, column2,..)

VALUES(value1, value2,..);

* Here, **viewName** is the view in which we have to insert data and we add values according to the columns in the view table.

**Example:** let's take the above created simple view EmpView2 and we want to insert a new row.

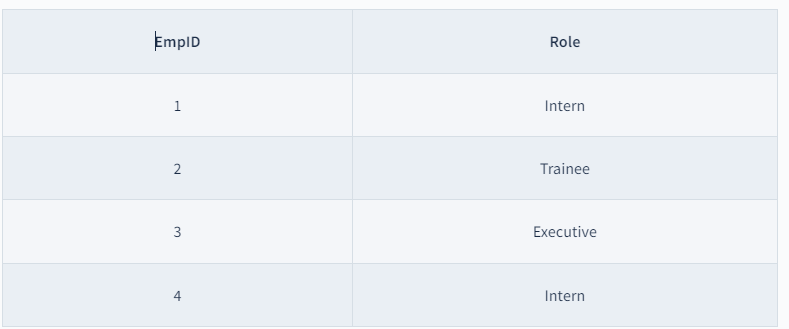
**Syntax:** INSERT INTO EmpView2(EmpID, Role) VALUES(4, 'Intern');

* Now to see the data in the EmpView2, We have to simply use the SELECT statement.

SELECT \* from EmpView2;

**Output:**

* Now the updated table EmpView2 has one more row of EmpID 4 and Role Intern.



**Deleting a row in a view**

* Deleting a row in the view takes the same syntax as we use to delete a row in a simple table.

**Syntax:**DELETE FROM ViewName WHERE condition;

* Here, **viewName** is the view from which data has been deleted, **condition** is the Condition by which we select rows to be deleted.

**Example:** let's take the above created simple view EmpView2 and we want to delete the row having EmpID = 1.

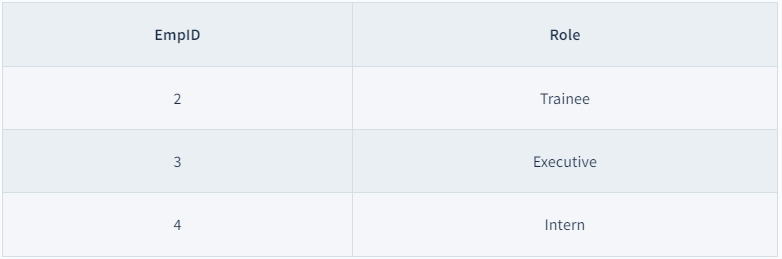
DELETE FROM EmpView2 WHERE EmpID = 1;

* Now to see the data in the EmpView2, We have to simply use the SELECT statement.

SELECT \* from EmpView2;

**Output:**

Now 1 record with data EmpID 1 and Role Intern has been deleted.



7.Explain selection and projection with examples in SQL?

Selection and projection are two major basic model operations that are used when processing queries in DBMS. Although both are associated with the operation of a database, it is used in different capacities. Selection operates by fishing out rows containing values that meet certain criteria while projection operates by fishing out certain columns of a table. It is important to know the difference between these two so as to refine queries and manipulate data in the most efficient way.

selection is an operation that allows the system to pick rows out of a table as indicated by a certain condition. The condition operates like a lens to pull out only the required columns and here it filters out all the rows which do not meet this condition. (The selection operation can be considered as row wise filtering. We can select specific rows using condition).

Syntax:

Select \*

from <table-name>

Where condition

Example:

SQL> Select \*

          from student

          Where sid=104;

**Example Table: Employees**

| **EmployeeID** | **Name** | **Department** | **Salary** |
| --- | --- | --- | --- |
| **1** | Alice | HR | 60000 |
| **2** | Bob | IT | 70000 |
| **3** | Charlie | IT | 75000 |
| **4** | David | HR | 62000 |
| **5** | Eva | Finance | 80000 |

SELECT \* FROM Employees WHERE Department = 'IT';

**Result**

| **EmployeeID** | **Name** | **Department** | **Salary** |
| --- | --- | --- | --- |
| **2** | Bob | IT | 70000 |
| **3** | Charlie | IT | 75000 |

Selection is also know as restrict, this operator yields values for all rows found in a table that satisfies given condition. Select can be used to list all of the row values or it can yield only those rows values that match a specific criteria. Select yields a horizontal subset of a table.

## **Projection**

This operation selects certain required attributes, while discarding other attributes. Projection in DBMS is a process of selecting some specific attribute/columns from a table while excluding the other attributes/columns from a selected table. It is useful for selecting a set of required attributes of the data. (The projection operation performs column wise filtering. Specific columns are selected in projection operation).

Syntax:

Select Attribute1, Atribute2

from table-name;

If all the columns of the table are selected, then it cannot be considered as projection.

Example:

 SQL> select name, dob

           from student

SELECT Name, Salary FROM Employees;

**Result**

| **Name** | **Salary** |
| --- | --- |
| **Alice** | 60000 |
| **Bob** | 70000 |
| **Charlie** | 75000 |
| **David** | 62000 |
| **Eva** | 80000 |

## **Difference Between Selection and Projection in DBMS**

| **Category** | **Selection** | **Projection** |
| --- | --- | --- |
| **Other Names** | The selection operation is also known as horizontal partitioning. | The Project operation is also known as vertical partitioning. |
| **Use** | It is used to choose the subset of tuples from the relation that satisfies the given condition mentioned in the syntax of selection. | It is used to select certain required [attributes](https://www.geeksforgeeks.org/types-of-attributes-in-er-model/), while discarding other attributes. |
| **Partitioning** | It partitions the table horizontally. | It partitions the table vertically. |
| **Which used first** | The selection operation is performed before projection (if they are to be used together). | The projection operation is performed after selection (if they are to be used together). |
| **Operator Used** | Select operator is used in Selection Operation. | Project operator is used in Projection Operation. |
| **Operator Symbol** | Select operator is denoted by Sigma symbol. | Project operator is denoted by Pi symbol. |
| **Commutative** | Selection is commutative. | Projection is not commutative. |
| **Column Selection** | Select is used to select all columns of a specific tuple. | Project is used to select specific columns. |
| **SQL Statements used** | SELECT, FROM, WHERE | SELECT, FROM |

8. Elaborate the concept of Nested queries and correlated queries.

SQL is used to manage data stored in a [**relational database**](https://www.tutorialspoint.com/sql/sql-databases.htm). SQL has the ability to nest queries. A nested [**query**](https://www.tutorialspoint.com/mysql/mysql-queries.htm) is a query within another query. A nested query allows for more complex and specific data retrieval.

a nested query involves a query that is placed within another query. The output of the inner query is used by the outer query. A nested query has two [**SELECT statements**](https://www.tutorialspoint.com/sql/sql-select-query.htm): one for the inner query and another for the outer query.

## **Syntax of Nested Queries**

The basic syntax of a nested query involves placing one query inside of another query. The inner query or subquery is executed first and returns a set of values that are then used by the outer query. The syntax for a nested query is as follows:

SELECT column1, column2, ...

FROM table1

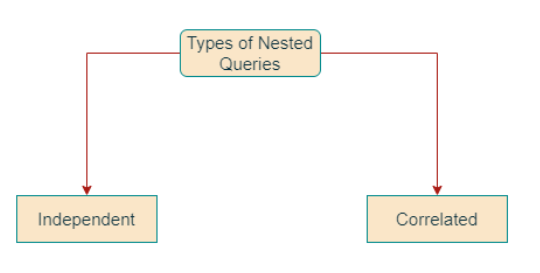
WHERE column1 IN ( SELECT column1

FROM table2

WHERE condition );

## **Types of Nested Queries in SQL**

Subqueries can be either correlated or non-correlated



### Non-correlated (or Independent) Nested Queries

Non-correlated (or Independent) Nested [**Queries**](https://www.tutorialspoint.com/mysql/mysql-queries.htm) : Non-correlated (or Independent) subqueries are executed independently of the outer query. Their results are passed to the outer query.

## **Execution Order in Independent Nested Queries**

In independent nested queries, the execution order is from the innermost query to the outer query. An outer query won't be executed until its inner query completes its execution. The outer query uses the result of the inner query.

### Operators Used in Independent Nested Queries

### IN Operator

The[**IN Operator**](https://www.tutorialspoint.com/sql/sql-in.htm) checks if a column value in the outer query's result is present in the inner query's result. The final result will have rows that satisfy the IN condition.

### NOT IN Operator

The NOT IN Operator checks if a column value in the outer query's result is not present in the inner query's result. The final result will have rows that satisfy the NOT IN condition.

### ALL Operator

The [**ALL Operator**](https://www.tutorialspoint.com/sql/sql-any-all-operator.htm)compares a value of the outer query's result with all the values of the inner query's result and returns the row if it matches all the values.

### ANY Operator

The ANY Operator compares a value of the outer query's result with all the inner query's result values and returns the row if there is a match with any value.

### Correlated Nested Queries

Correlated subqueries are executed once for each row of the outer query. They use values from the outer query to return results.

## **Execution Order in Co-related Nested Queries**

In correlated nested queries, the inner query uses values from the outer query, and the execution order is different from that of independent nested queries.

* First, the outer query selects the first row.
* Inner query uses the value of the selected row. It executes its query and returns a result set.
* Outer query uses the result set returned by the inner query. It determines whether the selected row should be included in the final output.
* Steps 2 and 3 are repeated for each row in the outer query's result set.
* This process can be resource-intensive. It may lead to performance issues if the query is not optimized properly.

### Operators Used in Co-related Nested Queries

In co-related nested queries, the following operators can be used

### EXISTS Operator

The EXISTS Operator checks whether a subquery returns any row. If it returns at least one row. EXISTS operator returns true, and the outer query continues to execute. If the subquery returns no row, the EXISTS operator returns false, and the outer query stops execution.

### NOT EXISTS Operator

The NOT EXISTS Operator checks whether a subquery returns no rows. If the subquery returns no row, the NOT EXISTS operator returns true, and the outer query continues to execute. If the subquery returns at least one row, the NOT EXISTS operator returns false, and the outer query stops execution.

### ANY Operator

The ANY Operator compares a value of the outer query's result with one or more values returned by the inner query. If the comparison is true for any one of the values returned by the inner query, the row is included in the final result.

### ALL Operator

The ALL Operator compares a value of the outer query's result with all the values returned by the inner query. Only if the comparison is true for all the values returned by the inner query, the row is included in the final result.

These operators are used to create co-related nested queries that depend on values from the outer query for execution.

Examples

Consider the following sample table to execute nested queries on these.

Table: employees table

|  |  |  |
| --- | --- | --- |
| **emp\_id** | **emp\_name** | **dept\_id** |
| 1 | John | 1 |
| 2 | Mary | 2 |
| 3 | Bob | 1 |
| 4 | Alice | 3 |
| 5 | Tom | 1 |

Table: departments table

|  |  |
| --- | --- |
| **dept\_id** | **dept\_name** |
| 1 | Sales |
| 2 | Marketing |
| 3 | Finance |

Table: sales table

|  |  |  |
| --- | --- | --- |
| **sale\_id** | **emp\_id** | **sale\_amt** |
| 1 | 1 | 1000 |
| 2 | 2 | 2000 |
| 3 | 3 | 3000 |
| 4 | 1 | 4000 |
| 5 | 5 | 5000 |
| 6 | 3 | 6000 |
| 7 | 2 | 7000 |

SELECT emp\_name FROM employees WHERE dept\_id IN (SELECT dept\_id FROM departments WHERE dept\_name = 'Sales');

Output

|  |
| --- |
| **emp\_name** |
| John |
| Bob |
| Tom |

Example 2: Find the names of all employees who have made a sale

**Required query**

SELECT emp\_name FROM employees WHERE EXISTS (SELECT emp\_id FROM sales WHERE employees.emp\_id = sales.emp\_id);

Output

|  |
| --- |
| **emp\_name** |
| John |
| Mary |
| Bob |
| Alice |
| Tom |

This query selects all employees from the "employees" table where there exists a sale record in the "sales" table for that employee.

Example 3: Find the names of all employees who have made sales greater than $1000.

**Required query**

SELECT emp\_name FROM employees WHERE emp\_id = ALL (SELECT emp\_id FROM sales WHERE sale\_amt > 1000);

Output

|  |
| --- |
| **emp\_name** |
| John |
| Mary |
| Bob |
| Alice |
| Tom |

This query selects all employees from the "employees" table. With the condition that where their emp\_id equals all the emp\_ids in the "sales" table where the sale amount is greater than $1000. Since all employees have made a sale greater than $1000, all employee names are returned.

OR

* Structured Query Language (SQL) is a programming language.
* SQL is used to manage data stored in a relational database.
* SQL has the ability of nest queries. A nested query is a query within another query. Nested query allows for more complex and specific data retrieval.
* In SQL, a nested query involves a query that is placed within another query. Output of the inner query is used by the outer query.
* A nested query has two SELECT statements: one for the inner query and another for the outer query.

## Syntax of Nested Queries

The basic syntax of a nested query involves placing one query inside of another query. Inner query or subquery is executed first and returns a set of values that are then used by the outer query. The syntax for a nested query is as follows:

SELECT column1, column2, ...

FROM table1

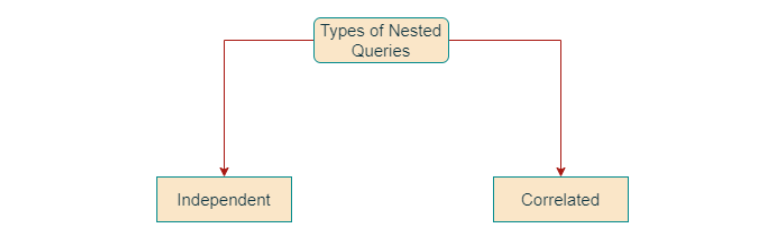
WHERE column1 IN ( SELECT column1

FROM table2

WHERE condition );

**Types of Nested Queries in SQL**

Sub queries can be either correlated or non-correlated



### Non-correlated (or Independent) Nested Queries

Non-correlated (or Independent) Nested Queries : Non-correlated (or Independent) subqueries are executed independently of the outer query. Their results are passed to the outer query.

Nested queries are a way to perform complex queries by embedding one query within another. The outer query can apply some conditions on the results of the inner query.

* A sub query typically appears with in where clause of a quert.
* A sub query is expressed inside parenthesis.
* The first query in the SQL statement in the SQL statement is known as outer query or main query.
* Query inside the SQL statement is known as outer query.
* The inner query is executed first, the output of the inner query is used as the input for the outer query.
* Nested queries is called as query with in the query.
* Instead of writing two separate queries, we can write one nested query.
* We can use various comparision operators with the sub query, such as >,<,=,IN,ANY,SOME,and ALL. A multiple row operator is very useful when the sub query returns more than one row.
* We cannot use the order by clause in a sub query, although it can be used inside the main qury.
* We can use the subqueries instead of using complex joins and unions.

**Example:**

| **E\_ID** | **E\_NAME** | **SAL** |
| --- | --- | --- |
| 1 | RAM | 25000 |
| 2 | RAMESH | 50000 |
| 3 | SUJIT | 75000 |
| 4 | SURESH | 35000 |
| 5 | HARI | 60000 |

To find the employee name whose salry is maximum.

SQL> select e\_name from Employee where sal=(select max(sal)from employee)

OUTPUT: e\_name

Sujith

Let usl use **STUDENT, COURSE, STUDENT\_COURSE** tables for understanding nested queries.

**STUDENT:**

| **S\_ID** | **S\_NAME** | **S\_ADDRESS** | **S\_PHONE** | **S\_AGE** |
| --- | --- | --- | --- | --- |
| S1 | RAM | DELHI | 9455123451 | 18 |
| S2 | RAMESH | GURGAON | 9652431543 | 18 |
| S3 | SUJIT | ROHTAK | 9156253131 | 20 |
| S4 | SURESH | DELHI | 9156768971 | 18 |

**COURSE:**

|  |  |
| --- | --- |
| **C\_ID** | **C\_NAME** |
| C1 | DSA |
| C2 | Programming |
| C3 | DBMS |

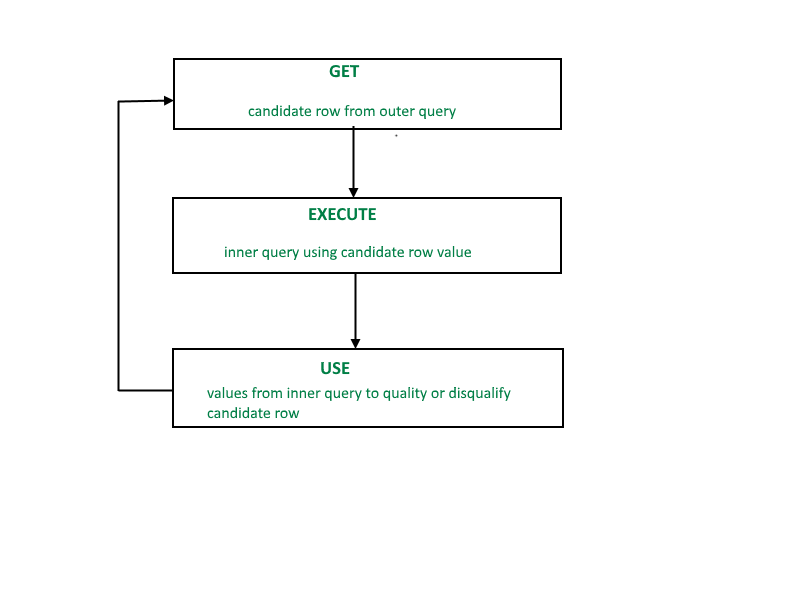
**STUDENT\_COURSE:**

| **S\_ID** | **C\_ID** |
| --- | --- |
| S1 | C1 |
| S1 | C3 |
| S2 | C1 |
| S3 | C2 |
| S4 | C2 |
| S4 | C3 |

**Independent Nested Queries:**In independent nested queries, query execution starts from innermost query to outermost queries.

* The execution of inner query is independent of outer query, but the result of inner query is used in execution of outer query.
* Various operators like IN, NOT IN, ANY, ALL etc are used in writing independent nested queries.
* **IN:** If we want to find out **S\_ID** who are enrolled in **C\_NAME** ‘DSA’ or ‘DBMS’, we can write it with the help of independent nested query and IN operator.
* From **COURSE** table, we can find out **C\_ID**for **C\_NAME** ‘DSA’ or DBMS’ and we can use these **C\_ID**s for finding **S\_ID**s from **STUDENT\_COURSE** TABLE.
* **STEP 1:** Finding **C\_ID** for **C\_NAME** =’DSA’ or ‘DBMS’ Select **C\_ID** from **COURSE** where **C\_NAME** = ‘DSA’ or **C\_NAME** = ‘DBMS’
* **STEP 2:** Using **C\_ID** of step 1 for finding **S\_ID** Select **S\_ID** from **STUDENT\_COURSE** where **C\_ID** IN (SELECT **C\_ID** from **COURSE** where **C\_NAME** = ‘DSA’ or **C\_NAME**=’DBMS’);
* The inner query will return a set with members C1 and C3 and outer query will return those **S\_ID**s for which **C\_ID** is equal to any member of set (C1 and C3 in this case).
* So, it will return S1, S2 and S4.   **Note:** If we want to find out names of **STUDENT**s who have either enrolled in ‘DSA’ or ‘DBMS’, it can be done as: Select S\_NAME from **STUDENT** where **S\_ID** IN (Select **S\_ID** from **STUDENT\_COURSE** where **C\_ID** IN (SELECT **C\_ID** from **COURSE** where **C\_NAME**=’DSA’ or **C\_NAME**=’DBMS’));
* **NOT IN:**If we want to find out **S\_ID**s of **STUDENT**s who have neither enrolled in ‘DSA’ nor in ‘DBMS’, it can be done as: Select **S\_ID** from **STUDENT** where **S\_ID** NOT IN (Select **S\_ID** from **STUDENT\_COURSE** where **C\_ID** IN (SELECT **C\_ID** from **COURSE** where **C\_NAME**=’DSA’ or **C\_NAME**=’DBMS’));
* The innermost query will return a set with members C1 and C3. Second inner query will return those **S\_ID**s for which **C\_ID** is equal to any member of set (C1 and C3 in this case) which are S1, S2 and S4. The outermost query will return those **S\_ID**s where **S\_ID** is not a member of set (S1, S2 and S4). So it will return S3.

**Co-related Nested Queries:**

Correlated subqueries are used for row-by-row processing. Each subquery is executed once for every row of the outer query.  
  
A correlated subquery is evaluated once for each row processed by the parent statement. The parent statement can be a **SELECT**, **UPDATE**, or **DELETE** statement.

SELECT column1, column2, ....

FROM table1 outer

WHERE column1 operator

(SELECT column1, column2

FROM table2

WHERE expr1 =

outer.expr2);

A correlated subquery is one way of reading every row in a table and comparing values in each row against related data. It is used whenever a subquery must return a different result or set of results for each candidate row considered by the main query. In other words, you can use a correlated subquery to answer a multipart question whose answer depends on the value in each row processed by the parent statement.

**Example**

Consider the EMPLOYEE table have the following records:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 1 | John | 20 | US | 2000.00 |
| 2 | Stephan | 26 | Dubai | 1500.00 |
| 3 | David | 27 | Bangkok | 2000.00 |
| 4 | Alina | 29 | UK | 6500.00 |
| 5 | Kathrin | 34 | Bangalore | 8500.00 |
| 6 | Harry | 42 | China | 4500.00 |
| 7 | Jackson | 25 | Mizoram | 10000.00 |

The subquery with a SELECT statement will be:

SELECT \*

    FROM EMPLOYEE

     WHERE ID IN (SELECT ID

     FROM EMPLOYEE

     WHERE SALARY > 4500);

This would produce the following result:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 4 | Alina | 29 | UK | 6500.00 |
| 5 | Kathrin | 34 | Bangalore | 8500.00 |
| 7 | Jackson | 25 | Mizoram | 10000.00 |

## Sub queries with the INSERT Statement

* SQL subquery can also be used with the Insert statement. In the insert statement, data returned from the subquery is used to insert into another table.
* In the subquery, the selected data can be modified with any of the character, date functions.

**Syntax:**

INSERT INTO table\_name (column1, column2, column3....)

SELECT \*

FROM table\_name

WHERE VALUE OPERATOR

**Example**

Consider a table EMPLOYEE\_BKP with similar as EMPLOYEE.

Now use the following syntax to copy the complete EMPLOYEE table into the EMPLOYEE\_BKP table.

INSERT INTO EMPLOYEE\_BKP

    SELECT \* FROM EMPLOYEE

    WHERE ID IN (SELECT ID

    FROM EMPLOYEE);

## Subqueries with the UPDATE Statement

The subquery of SQL can be used in conjunction with the Update statement. When a subquery is used with the Update statement, then either single or multiple columns in a table can be updated.

**Syntax**

UPDATE table

SET column\_name = new\_value

WHERE VALUE OPERATOR

    (SELECT COLUMN\_NAME

    FROM TABLE\_NAME

   WHERE condition);

**Example**

Let's assume we have an EMPLOYEE\_BKP table available which is backup of EMPLOYEE table. The given example updates the SALARY by .25 times in the EMPLOYEE table for all employee whose AGE is greater than or equal to 29.

UPDATE EMPLOYEE

   SET SALARY = SALARY \* 0.25

   WHERE AGE IN (SELECT AGE FROM CUSTOMERS\_BKP

      WHERE AGE >= 29);

This would impact three rows, and finally, the EMPLOYEE table would have the following records.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 1 | John | 20 | US | 2000.00 |
| 2 | Stephan | 26 | Dubai | 1500.00 |
| 3 | David | 27 | Bangkok | 2000.00 |
| 4 | Alina | 29 | UK | 1625.00 |
| 5 | Kathrin | 34 | Bangalore | 2125.00 |
| 6 | Harry | 42 | China | 1125.00 |
| 7 | Jackson | 25 | Mizoram | 10000.00 |

## Sub queries with the DELETE Statement

The subquery of SQL can be used in conjunction with the Delete statement just like any other statements mentioned above.

**Syntax**

DELETE FROM TABLE\_NAME

WHERE VALUE OPERATOR

 (SELECT COLUMN\_NAME

   FROM TABLE\_NAME

   WHERE condition);

**Example**Let's assume we have an EMPLOYEE\_BKP table available which is backup of EMPLOYEE table. The given example deletes the records from the EMPLOYEE table for all EMPLOYEE whose AGE is greater than or equal to 29.

DELETE FROM EMPLOYEE

   WHERE AGE IN (SELECT AGE FROM EMPLOYEE\_BKP

      WHERE AGE >= 29 );

This would impact three rows, and finally, the EMPLOYEE table would have the following records.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 1 | John | 20 | US | 2000.00 |
| 2 | Stephan | 26 | Dubai | 1500.00 |
| 3 | David | 27 | Bangkok | 2000.00 |
| 7 | Jackson | 25 | Mizoram | 10000.00 |

9. List and explain aggregate functions.

* Aggregate functions in DBMS are used to perform calculations on sets of data. They take a set of values as input and return a single value as output.
* These functions are often used to generate summary statistics on large datasets, such as the average, minimum, maximum, and sum of a set of values.
* They can also be used to count the number of rows in a dataset, and perform other complex calculations. Aggregate Functions in DBMS are used for summarizing the data.

## Types of Aggregate Functions in DBMS



* Aggregate Functions in DBMS are of different types as shown in the figure given below.

Aggregate functions are often used in databases, spreadsheets, and statistical software packages now common in the workplace. Aggregate functions are used extensively in economics and finance to provide key numbers that represent economic health or market performance.

* Aggregate functions deliver a single number to represent a larger data set. The numbers being used may themselves be products of aggregate functions.

| **PRODUCT** | **COMPANY** | **QTY** | **RATE** | **COST** |
| --- | --- | --- | --- | --- |
| Product1 | Company1 | 2 | 10 | 20 |
| Product2 | Company2 | 3 | 25 | 75 |
| Product3 | Company1 | 2 | 30 | 60 |
| Product4 | Company3 | 5 | 10 | 50 |
| Product5 | Company2 | 2 | 20 | 40 |
| Product6 | Company1 | 3 | 25 | 75 |
| Product7 | Company1 | 5 | 30 | 150 |
| Product8 | Company1 | 3 | 10 | 30 |
| Product9 | Company2 | 2 | 25 | 50 |
| Product10 | Company3 | 4 | 30 | 120 |

**Table Name: PREP\_TABLE**

* Many descriptive statistics are the result of aggregate functions.
* Economists use the outputs of data aggregation to plot changes over time and project future trends.
* The models created out of aggregated data can be used to influence policy and business decisions.

## Understanding Aggregate Function

The aggregate function simply refers to the calculations performed on a data set to get a single number that accurately represents the underlying data. The use of computers has improved how these calculations are performed, allowing aggregate functions to produce results very quickly and even adjust weightings based on the confidence the user has in the data. Thanks to computers, aggregate functions can handle ever larger and more complex data sets.

### COUNT()

This COUNT() function is used to count the number of rows in a table or a result set. It can also be used with a specific column to count the number of non-null values in that column.

**Syntax of COUNT() Function**

COUNT(\*) OR COUNT(COLUMN\_NAME)

**Example of COUNT() Function**

SQL Query:

SELECT COUNT(\*)

FROM PREP\_TABLE;

**Output:**

10

**Example of COUNT() Function with WHERE Clause**

SQL Query:

SELECT COUNT(\*)

FROM PREP\_TABLE;

WHERE RATE>=20;

**Output:**

7

**Example of COUNT() Function with DISTINCT**

SQL Query:

SELECT COUNT(DISTINCT COMPANY)

FROM PREP\_TABLE;

**Output:**

3

**Example of COUNT() Function with GROUP BY**

SQL Query:

SELECT COMPANY, COUNT(\*)

FROM PREP\_TABLE

GROUP BY COMPANY;

**Output:**

Company1 5

Company2 3

Company3 2

**Example of COUNT() Function with HAVING**

SQL Query:

SELECT COMPANY, COUNT(\*)

FROM PREP\_TABLE

GROUP BY COMPANY

HAVING COUNT(\*)>2;

**Output:**

Company1 5

Company2 3

### SUM()

The SUM() function in DBMS accepts a column name as an input and returns the total of all non-NULL values in that column. It only works on numeric fields (i.e the columns contain only numeric values). If this function is applied to columns that include both non-numeric (like, strings) and numeric values, it only considers the numeric values. If there are no numeric values, the method returns 0.

**Syntax of SUM() Function**

SUM(COLUMN\_NAME)

**Example of SUM() Function**

SQL Query:

SELECT SUM(COST)

FROM PREP\_TABLE;

**Output:**

670

**Example of SUM() Function with WHERE**

SQL Query:

SELECT SUM(COST)

FROM PREP\_TABLE

WHERE QTY>3;

**Output:**

320

**Example of SUM() Function with GROUP BY**

SQL Query:

SELECT SUM(COST)

FROM PREP\_TABLE

WHERE QTY>3

GROUP BY COMPANY;

**Output:**

Company1 150

Company2 170

**Example of SUM() Function with HAVING**

SQL Query:

SELECT COMPANY, SUM(COST)

FROM PREP\_TABLE

GROUP BY COMPANY

HAVING SUM(COST)>=170;

**Output:**

Company1 335

Company3 170

### AVG()

The AVG() aggregate function in DBMS takes the column name as an input and returns the average of all non-NULL values in that column. It only works on numeric fields (i.e the columns contain only numeric values).

**Syntax of AVG() Function**

AVG(COLUMN\_NAME)

**Example of AVG() Function**

SQL Query:

SELECT AVG(COST)

FROM PREP\_TABLE;

**Output:**

67.00

### MAX()

The MAX() function accepts the column name as a parameter and returns the maximum value in the column. When no row is specified, MAX() function returns NULL.

**Syntax of MAX() Function**

MAX(COLUMN\_NAME)

**Example of MAX() Function**

SQL Query:

SELECT MAX(RATE)

FROM PREP\_TABLE;

**Output:**

30

### MIN()

The MIN() function accepts the column name as a parameter and returns the minimum value in the column. When no row is specified, MIN() Function returns NULL as result.

**Syntax of MIN() Function:**

MIN(COLUMN\_NAME)

**Example of MIN() Function**

SQL Query:

SELECT MIN(RATE)

FROM PREP\_TABLE;

**Output:**

10

The Aggregate Functions in DBMS help us in dealing with large datasets. We have discussed different types of Aggregate Functions in DBMS which include, COUNT(), SUM(), AVG(), MAX(), and MIN().

**Write notes on Arithmetic Operators in SQL with examples.**

The SQL reserved words and characters are called operators, which are used with a WHERE clause in a SQL query. In SQL, an operator can either be a unary or binary operator. The unary operator uses only one operand for performing the unary operation, whereas the binary operator uses two operands for performing the binary operation.

## Types of Operator

SQL operators are categorized in the following categories:

1. SQL Arithmetic Operators
2. SQL Comparison Operators
3. SQL Logical Operators
4. SQL Set Operators
5. SQL Bit-wise Operators
6. SQL Unary Operators

The **Arithmetic Operators** perform the mathematical operation on the numerical data of the SQL tables. These operators perform addition, subtraction, multiplication, and division operations on the numerical operands.

**Following are the various arithmetic operators performed on the SQL data:**

1. SQL Addition Operator (+)
2. SQL Subtraction Operator (-)
3. SQL Multiplication Operator (+)
4. SQL Division Operator (-)
5. SQL Modulus Operator (+)

### SQL Addition Operator (+)

The **Addition Operator** in SQL performs the addition on the numerical data of the database table. In SQL, we can easily add the numerical values of two columns of the same table by specifying both the column names as the first and second operand. We can also add the numbers to the existing numbers of the specific column.

**Syntax of SQL Addition Operator:**

1. SELECT operand1 + operand2;

**Let's understand the below example which explains how to execute Addition Operator in SQL query:**

This example consists of an **Employee\_details** table, which has four columns **Emp\_Id, Emp\_Name, Emp\_Salary, and Emp\_Monthlybonus.**

|  |  |  |  |
| --- | --- | --- | --- |
| **Emp Id** | **Emp Name** | **Emp Salary** | **Emp Monthlybonus** |
| 101 | Tushar | 25000 | 4000 |
| 102 | Anuj | 30000 | 200 |

* Suppose, we want to add **20,000** to the salary of each employee specified in the table. Then, we have to write the following query in the SQL:

1. SELECT Emp\_Salary + 20000 as Emp\_New\_Salary FROM Employee\_details;

In this query, we have performed the SQL addition operation on the single column of the given table.

* Suppose, we want to add the Salary and monthly bonus columns of the above table, then we have to write the following query in SQL:

1. SELECT Emp\_Salary + Emp\_Monthlybonus as Emp\_Total\_Salary FROM Employee\_details;

In this query, we have added two columns with each other of the above table.

## SQL Subtraction Operator (-)

The Subtraction Operator in SQL performs the subtraction on the numerical data of the database table. In SQL, we can easily subtract the numerical values of two columns of the same table by specifying both the column names as the first and second operand. We can also subtract the number from the existing number of the specific table column.

**Syntax of SQL Subtraction Operator:**

1. SELECT operand1 - operand2;

**Let's understand the below example which explains how to execute Subtraction Operator in SQL query:**

This example consists of an **Employee\_details** table, which has four columns **Emp\_Id, Emp\_Name, Emp\_Salary, and Emp\_Monthlybonus.**

|  |  |  |  |
| --- | --- | --- | --- |
| **Emp Id** | **Emp Name** | **Emp Salary** | **Penalty** |
| 201 | Abhay | 25000 | 200 |
| 202 | Sumit | 30000 | 500 |

* Suppose we want to subtract 5,000 from the salary of each employee given in the **Employee\_details** table. Then, we have to write the following query in the SQL:

1. SELECT Emp\_Salary - 5000 as Emp\_New\_Salary FROM Employee\_details;

In this query, we have performed the SQL subtraction operation on the single column of the given table.

* If we want to subtract the penalty from the salary of each employee, then we have to write the following query in SQL:

1. SELECT Emp\_Salary - Penalty as Emp\_Total\_Salary FROM Employee\_details;

## SQL Multiplication Operator (\*)

The Multiplication Operator in SQL performs the Multiplication on the numerical data of the database table. In SQL, we can easily multiply the numerical values of two columns of the same table by specifying both the column names as the first and second operand.

**Syntax of SQL Multiplication Operator:**

1. SELECT operand1 \* operand2;

**Let's understand the below example which explains how to execute Multiplication Operator in SQL query:**

This example consists of an **Employee\_details** table, which has four columns **Emp\_Id, Emp\_Name, Emp\_Salary, and Emp\_Monthlybonus.**

|  |  |  |  |
| --- | --- | --- | --- |
| **Emp Id** | **Emp Name** | **Emp Salary** | **Penalty** |
| 201 | Abhay | 25000 | 200 |
| 202 | Sumit | 30000 | 500 |

* Suppose, we want to double the salary of each employee given in the **Employee\_details** table. Then, we have to write the following query in the SQL:

1. SELECT Emp\_Salary \* 2 as Emp\_New\_Salary FROM Employee\_details;

In this query, we have performed the SQL multiplication operation on the single column of the given table.

* If we want to multiply **the Emp\_Id** column to **Emp\_Salary** column of that employee whose **Emp\_Id** is **202,** then we have to write the following query in SQL:

1. SELECT Emp\_Id \* Emp\_Salary as Emp\_Id \* Emp\_Salary FROM Employee\_details WHERE Emp\_Id = 202;

In this query, we have multiplied the values of two columns by using the WHERE clause.

## SQL Division Operator (/)

The Division Operator in SQL divides the operand on the left side by the operand on the right side.

**Syntax of SQL Division Operator:**

1. SELECT operand1 / operand2;

In SQL, we can also divide the numerical values of one column by another column of the same table by specifying both column names as the first and second operand.

We can also perform the division operation on the stored numbers in the column of the SQL table.

**Let's understand the below example which explains how to execute Division Operator in SQL query:**

This example consists of an **Employee\_details** table, which has three columns **Emp\_Id, Emp\_Name, and Emp\_Salary.**

|  |  |  |
| --- | --- | --- |
| **Emp Id** | **Emp Name** | **Emp Salary** |
| 201 | Abhay | 25000 |
| 202 | Sumit | 30000 |

* Suppose, we want to half the salary of each employee given in the Employee\_details table. For this operation, we have to write the following query in the SQL:

1. SELECT Emp\_Salary / 2 as Emp\_New\_Salary FROM Employee\_details;

In this query, we have performed the SQL division operation on the single column of the given table.

## SQL Modulus Operator (%)

The Modulus Operator in SQL provides the remainder when the operand on the left side is divided by the operand on the right side.

**Syntax of SQL Modulus Operator:**

1. SELECT operand1 % operand2;

**Let's understand the below example which explains how to execute Modulus Operator in SQL query:**

This example consists of a **Division** table, which has three columns **Number, First\_operand, and Second\_operand.**

|  |  |  |
| --- | --- | --- |
| **Number** | **First operand** | **Second operand** |
| 1 | 56 | 4 |
| 2 | 32 | 8 |
| 3 | 89 | 9 |
| 4 | 18 | 10 |
| 5 | 10 | 5 |

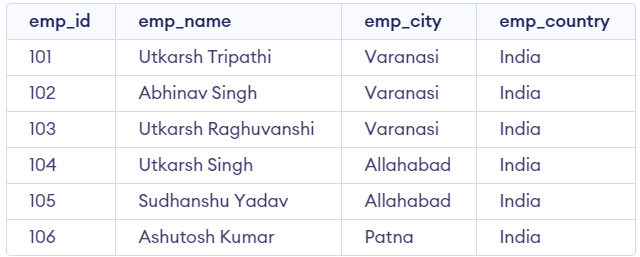
* If we want to get the remainder by dividing the numbers of First\_operand column by the numbers of Second\_operand column, then we have to write the following query in SQL:

1. SELECT First\_operand % Second\_operand as Remainder FROM Employee\_details;

# **SQL LOGICAL OPERATORS**

[Logical operators](https://www.geeksforgeeks.org/sql-logical-operators/" \t "_blank) in SQL are used to combine **multiple conditions** in a query to **control the flow of execution**. They evaluate whether these conditions are **TRUE**, **FALSE**, or **NULL**, assisting in refining query results effectively. By using these operators, developers can retrieve **highly specific data** based on given conditions.

We will use the following **employee table** throughout the examples. This table represents employee details, including their **unique ID**, **name**, **city**, and **country**.



*employee Table*

Below is the comprehensive list of **SQL Logical Operators** along with their meanings, detailed explanations, and practical examples:

**1. AND Operator**

The [AND operator](https://www.geeksforgeeks.org/sql-and-and-or-operators/) is used to combine **two**or **more conditions** in an SQL query. It returns records only when **all conditions** specified in the query are **true**. This operator is commonly used when**filtering data** that must satisfy multiple criteria simultaneously.

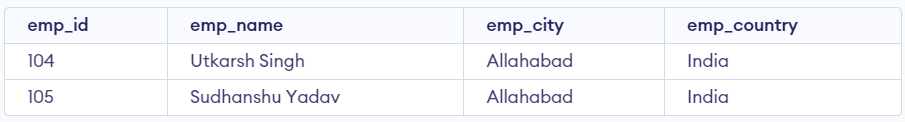
**Example**

Retrieve the records of employees from the employees table who are located in '**Allahabad**' and belong to '**India**', ensuring that both conditions are met.

**Query:**

SELECT \* FROM employee WHERE emp\_city = 'Allahabad' AND emp\_country = 'India';

**Output**



*output*

**Explanation:**

In the output, both conditions (emp\_city = 'Allahabad' and emp\_country = 'India') are satisfied for the listed employees, so these records are returned by the query.

**2. IN Operator**

The [IN operator](https://www.geeksforgeeks.org/sql-in-operator/) simplifies the process of checking if a **value matches any value in a list**, making it more efficient and readable compared to using multiple **OR** conditions. This operator is especially helpful when we need to filter results based on **multiple possible values** for a given column, reducing the complexity of the query.

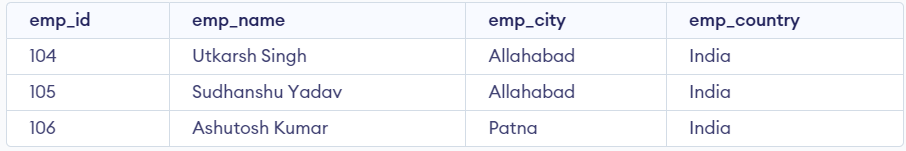
**Example**

Retrieve the records of employees from the employee table who are located in either '**Allahabad**' or '**Patna**'.

**Query:**

SELECT \* FROM employee WHERE emp\_city IN ('Allahabad', 'Patna');

**Output**



*output*

**Explanation:**

In this query, the **IN** operator checks if the value of the emp\_city column matches any value in the list ('Allahabad', 'Patna'). The query returns all employees who are located in either of these two cities.

**3. NOT Operator**

The[NOT operator](https://www.geeksforgeeks.org/sql-not-operator/) is used to reverse the result of a condition, returning **TRUE** when the condition is **FALSE**. It is typically used to exclude records that match a specific condition, making it useful for filtering out unwanted data.

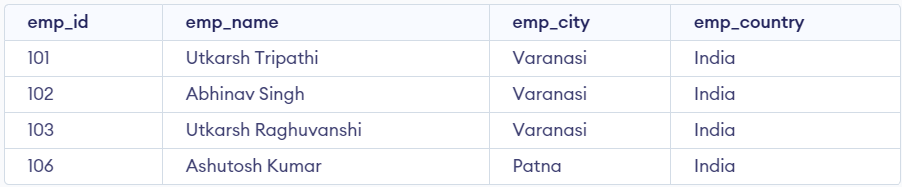
**Example**

Retrieve the records of employees from the employee table whose city names do not start with the letter 'A'.

**Query:**

SELECT \* FROM employee WHERE emp\_city NOT LIKE 'A%';

**Output**



*output*

**Explanation:**

In this query, the **NOT** operator negates the **LIKE** condition. The **LIKE** operator is used to match patterns in string data, and the 'A%' pattern matches any city name that starts with the letter 'A'. By using the **NOT** operator, we exclude cities starting with 'A' from the result set.

**4. OR Operator**

The [OR operator](https://www.geeksforgeeks.org/sql-and-and-or-operators/) combines multiple conditions in a SQL query and returns **TRUE** if at least one of the conditions is satisfied. It is ideal for situations where you want to retrieve records that meet any of several possible conditions.

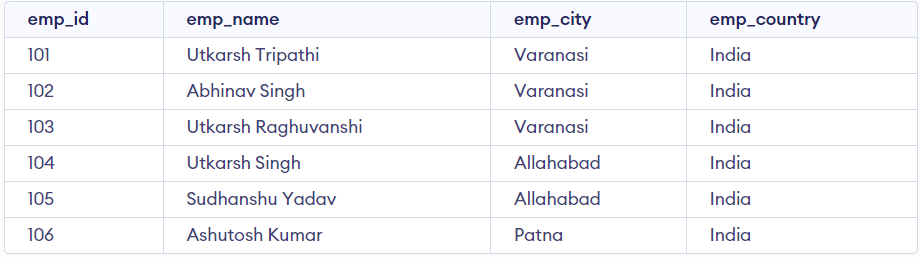
**Example**

Retrieve the records of employees from the employee table who are either from '**Varanasi**' or have '**India**' as their country.

**Query**

SELECT \* FROM employee WHERE emp\_city = 'Varanasi' OR emp\_country = 'India';

**Output**



*output*

**Explanation:**

In this case, the output includes employees from '**Varanasi**' as well as those who have '**India**' as their country, even if they are from different cities. The query returns all records where at least one of the conditions is true.

**5. LIKE Operator**

The[LIKE operator](https://www.geeksforgeeks.org/sql-like/) in SQL is used in the [WHERE clause](https://www.geeksforgeeks.org/sql-where-clause/) to search for a specified pattern in a column. It is particularly useful when we want to perform pattern matching on string data. The **LIKE** operator works with two main wildcards:

* **%**: Represents zero or more characters. It allows matching any sequence of characters in the string.
* **\_**: Represents exactly one character. It is used when you want to match a specific number of characters at a given position.

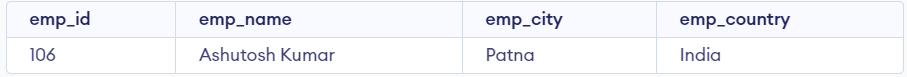
**Example**

Retrieve the records of employees from the employee table whose city names start with the letter 'P'.

**Query:**

SELECT \* FROM employee WHERE emp\_city LIKE 'P%';

**Output**



*output*

**Explanation:**

In this case, the output includes only those employees whose emp\_city starts with 'P'. The % wildcard ensures that the query matches any city name starting with the specified letter, regardless of how many additional characters follow it.

**6. BETWEEN Operator**

The [BETWEEN operator](https://www.geeksforgeeks.org/sql-between/) in SQL allows us to test if a value or expression lies within a **specified range**. The **BETWEEN** condition is inclusive, meaning it includes both the **lower**and in the results. This operator is particularly useful when we need to filter records based on a range of values, such as numerical ranges, dates, or even text values.

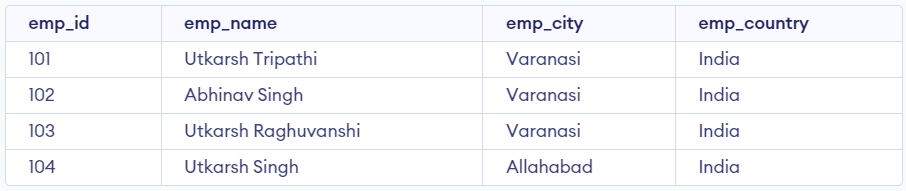
**Example**

Retrieve the records of employees from the employee table whose emp\_id values fall within the range of 101 to 104 (inclusive).

**Query:**

SELECT \* FROM employee WHERE emp\_id BETWEEN 101 AND 104;

**Output**



*output*

**Explanation:**

In this query, the **BETWEEN** operator is used to filter employees with emp\_id values ranging from 101 to 104. Since the **BETWEEN** operator is inclusive, employees with emp\_id values of 101, 102, 103, and 104 will be included in the result set.

**7. ALL Operator**

The [ALL operator](https://www.geeksforgeeks.org/sql-all-and-any/) in SQL is used to compare a value to all values returned by a subquery. It returns **TRUE** if the condition specified is **TRUE** for all values retrieved by the subquery. The **ALL** operator is commonly used with **SELECT**, **WHERE**, and [HAVING](https://www.geeksforgeeks.org/sql-having-clause-with-examples/) clauses to ensure that a value satisfies a condition when compared to a set of values.

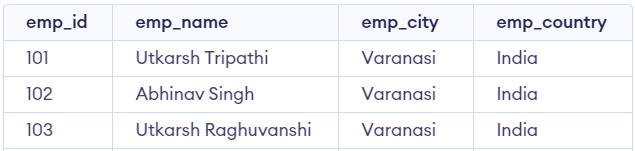
**Example**

Retrieve the records of employees whose emp\_id is equal to all **emp\_id**values in the employees table where the **emp\_city** is '**Varanasi**'.

**Query:**

SELECT \* FROM employee WHERE emp\_id = ALL   
 (SELECT emp\_id FROM employee WHERE emp\_city = 'Varanasi');

**Output**



*output*

**Explanation:**

The query checks whether **emp\_id**in the outer query is equal to every emp\_id from the **subquery**(which retrieves emp\_id values from employees in 'Varanasi'). In this case, the output will include employees whose emp\_id matches all the values in the subquery, i.e., employees with emp\_id values 101, 102, and 103 who are in 'Varanasi'.

**8. ANY Operator**

The [ANY operator](https://www.geeksforgeeks.org/sql-all-and-any/) in SQL is used to compare a value with the results of a [subquery](https://www.geeksforgeeks.org/sql-subquery/). It returns **TRUE** if the value satisfies the condition with **any** of the values returned by the subquery. This operator allows for greater flexibility when you want to check if a value matches **at least one** of the results in a set of values.

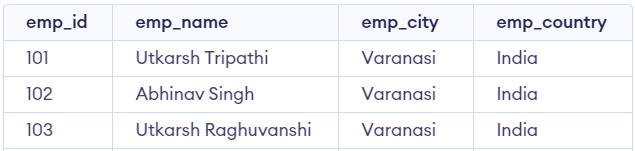
**Example**

Retrieve the records of employees whose emp\_id matches **any** of the emp\_id values in the employees table where the emp\_city is '**Varanasi**'.

**Query:**

SELECT \* FROM employee WHERE emp\_id = ANY  
 (SELECT emp\_id FROM employee WHERE emp\_city = 'Varanasi');

**Output**



**Explanation:**

The output includes employees whose emp\_id matches **any** of the emp\_id values from the subquery. In this case, the subquery selects emp\_id values from employees in '**Varanasi**'. The outer query then returns records where emp\_id matches at least one of these values, which includes employees with emp\_id **101, 102, and 106**.

**9. EXISTS Operator**

The [EXISTS operator](https://www.geeksforgeeks.org/sql-exists/) in SQL is used to check whether a subquery returns any rows. It evaluates to **TRUE** if the subquery results in one or more rows. The **EXISTS** operator is typically used with **SELECT**, **UPDATE**, **INSERT**, and **DELETE** statements to determine if any rows exist that meet a specified condition. It is often used in **correlated subqueries** where the subquery references columns from the outer query.

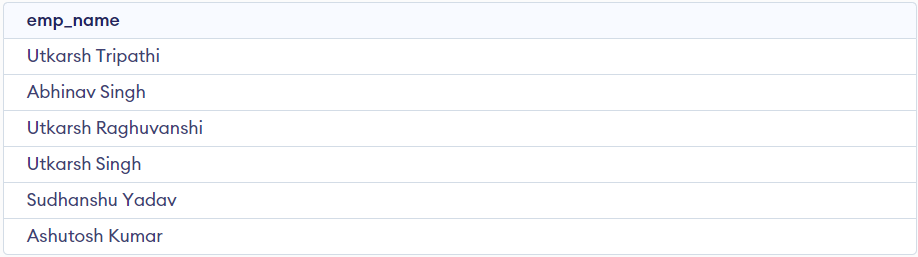
**Example**

Retrieve the names of employees from the employee table if there are any employees in the employee table who are located in 'Patna'.

**Query**

SELECT emp\_name FROM employee WHERE EXISTS  
 (SELECT emp\_id FROM employee WHERE emp\_city = 'Patna');

**Output**



*output*

**Explanation:**

The output includes the employee names because the **EXISTS** operator checks if there are any employees from Patna. If any rows are returned from the subquery, the **EXISTS** operator returns **TRUE**, and the employee names are included in the result. The query will return all employees as long as there are employees from Patna.

**10. SOME Operator**

The [SOME operator](https://www.geeksforgeeks.org/sql-some/) in SQL is used in conjunction with comparison operators such as <, >, =, <=, etc., to compare a value with the results of a subquery. It returns **TRUE** if the condition is met with **any** value returned by the subquery. The **SOME** operator allows us to **perform comparisons**with any of the values returned by a subquery, and it is particularly useful when we want to match a value against a set of values rather than a single value.

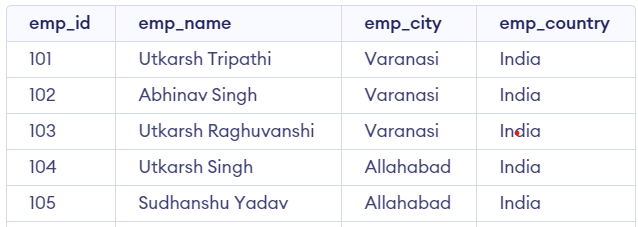
**Example**

Retrieve the records of employees from the employee table where the emp\_id is less than **any** of the emp\_id values from employees located in 'Patna'.

**Query:**

SELECT \* FROM employee WHERE emp\_id < SOME   
 (SELECT emp\_id FROM employee WHERE emp\_city = 'Patna');

**Output**



*output*

**Explanation:**

The output includes employees whose emp\_id is less than **any** of the emp\_id values of employees located in '**Patna**'. In this case, the query checks if the emp\_id valuesare less than the corresponding **emp\_id** values from the 'Patna' employees. If the condition is satisfied for at least one of the values in the subquery, those employees are included in the result.

**Explain a) Group by clause, b) Order by clause and c) Having clause .**

**a) Group by clause:**

* The group by clause is utilized in sql with the select statement to organize similar data into groups. (OR) The group by clause used on collaboration with select statement to arrange identical data into groups.
* It combines the multiple records in single or more columns using some functions.
* Generally group by clause used on aggregate functions. The main purpose of grouping the records of a table based on particular columns is to perform calculations on these groups. Therefore, The GROUP BY clause is typically used with aggregate functions such as SUM(), COUNT(), AVG(), MAX(), or MIN() etc.
* It returns only one result per group of data.
* It always precedes the order by clause and after the where clause (If they exist).

**Syntax:**

Select <columnlist> from <tablename> group by columnname;

Example:

Table name: Employee

|  |  |  |  |
| --- | --- | --- | --- |
| **EID** | **ENAME** | **ESAL** | **DEPT** |
| 10 | Arun | 10000 | Sales |
| 20 | Vikas | 20000 | Accounts |
| 30 | Ajay | 25000 | Sales |
| 40 | Charan | 35000 | Accounts |
| 50 | Deepak | 15000 | Product |

For example: 1)If you want to display total salaries of each department.

Sql>Select dept,sum(esal) from employee group by dept;

|  |  |
| --- | --- |
| DEPT | ESAL |
| Sales | 35000 |
| Account | 50000 |
| Product | 15000 |

b)Order by:

* Order by clause is used to sort data in either ascending order or descending order based on one or more columns.
* The **ORDER BY** statement in **SQL** is used to **sort the fetched data** in either ascending or descending according to one or more columns.

Here are some basic rules of Order By statement in [SQL](https://www.geeksforgeeks.org/sql-tutorial/).

* By default ORDER BY sorts the data in ascending order.
* We can use the keyword DESC to sort the data in descending order and the keyword ASC to sort in ascending order.

Syntax:

*SELECT \* FROM table\_name ORDER BY column\_name ASC | DESC*

OR

Select <columnlist> from <tablename> where <condition> order by <columnname>

**Explanation of the Syntax:**

* **table\_name**: name of the table.
* **column\_name**: name of the column according to which the data is needed to be arranged.
* **ASC**: to sort the data in ascending order.
* **DESC**: to sort the data in descending order.
* use either ASC or DESC to sort in ascending or descending order

Example: 1) If you want retrieve the employee names in ascending order.

Sql> Select \* from employee order by ename asc;

|  |  |  |  |
| --- | --- | --- | --- |
| **EID** | **ENAME** | **ESAL** | **DEPT** |
| 30 | Ajay | 25000 | Sales |
| 10 | Arun | 10000 | Sales |
| 40 | Charan | 35000 | Accounts |
| 50 | Deepak | 15000 | Product |
| 20 | Vikas | 20000 | Accounts |

2) If you want to retrieve employee salary in descending order.

Sql>Select \* from employee order by asc;

|  |  |  |  |
| --- | --- | --- | --- |
| **EID** | **ENAME** | **ESAL** | **DEPT** |
| 10 | Arun | 10000 | Sales |
| 50 | Deepak | 15000 | Product |
| 20 | Vikas | 20000 | Accounts |
| 30 | Ajay | 25000 | Sales |
| 40 | Charan | 35000 | Accounts |

## GROUP BY with ORDER BY Clause

* We can use the ORDER BY clause with GROUP BY in SQL to sort the grouped data by one or more columns.

### Syntax

Following is the syntax for using ORDER BY clause with GROUP BY clause in SQL −

SELECT column1, column2, ..., aggregate\_function(columnX) AS alias

FROM table

GROUP BY column1, column2, ...

ORDER BY column1 [ASC | DESC], column2 [ASC | DESC], ...;

### Example

* In here, we are finding the highest salary for each age, sorted by high to low −

SELECT DEPT, MIN(SALARY) AS MIN\_SALARY

FROM EMPLOYEE

GROUP BY DEPT ORDER BY MIN\_SALARY DESC;

### Output

|  |  |
| --- | --- |
| **DEPT** | **ESAL** |
| Accounts | 20000 |
| Product | 15000 |
| Sales | 10000 |

c) Having

* The HAVING clause was introduced in SQL to allow the filtering of query results based on aggregate functions and groupings, which cannot be achieved using the [WHERE](https://www.geeksforgeeks.org/sql-where-clause/)clause that is used to filter individual rows.
* The HAVING clause is used to apply a filter on the result of [GROUP BY](https://www.geeksforgeeks.org/sql-group-by/) based on the specified condition. The conditions are Boolean type i.e. *use of logical operators  (AND, OR).*
* This clause was included in SQL as the [WHERE](https://www.geeksforgeeks.org/sql-where-clause/) keyword failed when we use it with aggregate expressions. Having is a very generally used clause in [SQL](https://www.geeksforgeeks.org/sql-tutorial/).
* Similar to WHERE it helps to apply conditions, but HAVING works with groups. If you wish to filter a group, the HAVING clause comes into action.
* It is used to specify the condition that filtered which group results appears in the final result

**Some important points:**

* Having clause is used to filter data according to the conditions provided.
* Having a clause is generally used in reports of large data.
* Having clause is only used with the SELECT clause.
* The expression in the syntax can only have constants.
* In the query, [ORDER BY](https://www.geeksforgeeks.org/sql-order-by/)  is to be placed after the HAVING clause, if any.
* HAVING Clause is implemented in column operation.
* Having clause is generally used after [GROUP BY](https://www.geeksforgeeks.org/sql-group-by/).

Syntax:

Sql> Select <column list> from <table name> group by <column list> having <condition>

Example: 1)If you want to display the total salaries of sales department.

Sql> Select dept, sum(esal) from employee group by dept having dept=’sales’;

|  |  |
| --- | --- |
| **DEPT** | **ESAL** |
| Sales | 35000 |

2) If you want to retrieve the data average salary greater than 35000;

Sql> Select dept,avg(sal) from employee group by dept having avg(sal)>35000;

|  |  |
| --- | --- |
| **DEPT** | **AVGSAL** |
| Accounts | 27500 |
| Product | 15000 |
| Sales | 17500 |