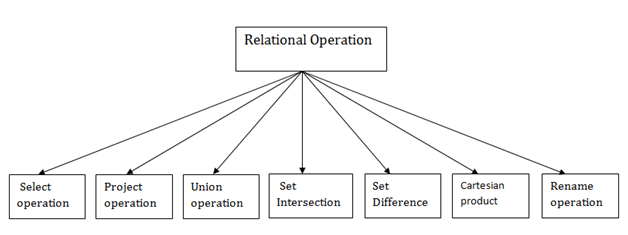
**Unit 2**

**Relational Algebra**

Relational algebra is a procedural query language. It gives a step by step process to obtain the result of the query. It uses operators to perform queries.

**Types of Relational operation**



* The select operation selects tuples that satisfy a given predicate.
* It is denoted by sigma (σ).

**Notation:   σ p(r)**

Where:

σ is used for selection prediction  
r is used for relation  
p is used as a propositional logic formula which may use connectors like: AND OR and NOT. These relational can use as relational operators like =, ≠, ≥, <, >, ≤.

As it is pure mathematics, there is no use of English Keywords in Relational Algebra and operators are represented using symbols.

**Fundamental Operators**

These are the [basic/fundamental operators](https://www.geeksforgeeks.org/basic-operators-in-relational-algebra-2/) used in Relational Algebra.

1. [Selection(σ)](https://www.geeksforgeeks.org/select-operation-in-relational-algebra/)
2. [Projection(π)](https://www.geeksforgeeks.org/difference-between-selection-and-projection-in-dbms/)
3. [Union(U)](https://www.geeksforgeeks.org/sql-union-operator/)
4. [Set Difference(-)](https://www.geeksforgeeks.org/set-theory-operations-in-relational-algebra/)
5. [Set Intersection(∩)](https://www.geeksforgeeks.org/sql-intersect-clause/)
6. [Rename(ρ)](https://www.geeksforgeeks.org/rename-operation-in-relational-algebra/)
7. [Cartesian Product(X)](https://www.geeksforgeeks.org/cartesian-product-operation-in-relational-algebra/)

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

**Example:**

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

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

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

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

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

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

π(B,C)R will show following columns.

| **B** | **C** |
| --- | --- |
| 2 | 4 |
| 2 | 3 |
| 3 | 4 |

**Note:**By Default, projection removes duplicate data.

**3. Union(U):** Union operation in relational algebra is the same as union operation in [set theory](https://www.geeksforgeeks.org/set-operations/).

**Example**   **FRENCH**

| **Student\_Name** | **Roll\_Number** |
| --- | --- |
| Ram | 01 |
| Mohan | 02 |
| Vivek | 13 |
| Geeta | 17 |

**GERMAN**

| **Student\_Name** | **Roll\_Number** |
| --- | --- |
| Vivek | 13 |
| Geeta | 17 |
| Shyam | 21 |
| Rohan | 25 |

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

π(Student\_Name)FRENCH U π(Student\_Name)GERMAN

| Student\_Name |
| --- |
| Ram |
| Mohan |
| Vivek |
| Geeta |
| Shyam |
| Rohan |

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

**4. Set Difference(-):**

 Set Difference in relational algebra is the same set difference operation as in set theory.

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

π(Student\_Name)FRENCH - π(Student\_Name)GERMAN

| **Student\_Name** |
| --- |
| Ram |
| Mohan |

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

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

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

π(Student\_Name)FRENCH ∩ π(Student\_Name)GERMAN

| **Student\_Name** |
| --- |
| Vivek |
| Geeta |

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

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

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

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

**Example:**

**A**

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

**B**

| **ID** | **Course** |
| --- | --- |
| 1 | DS |
| 2 | DBMS |

**A X B**

| **Name** | **Age** | **Sex** | **ID** | **Course** |
| --- | --- | --- | --- | --- |
| Ram | 14 | M | 1 | DS |
| Ram | 14 | M | 2 | DBMS |
| Sona | 15 | F | 1 | DS |
| Sona | 15 | F | 2 | DBMS |
| Kim | 20 | M | 1 | DS |
| Kim | 20 | M | 2 | DBMS |

**Note:** If A has ‘n’ tuples and B has ‘m’ tuples then A X B will have ‘ n\*m ‘ tuples.

**Derived Operators**

These are some of the [derived operators](https://www.geeksforgeeks.org/extended-operators-in-relational-algebra/), which are derived from the fundamental operators.

1. [Natural Join(⋈)](https://www.geeksforgeeks.org/sql-natural-join/)
2. [Conditional Join](https://www.geeksforgeeks.org/extended-operators-in-relational-algebra/)
3. **Natural Join(⋈):**Natural join is a binary operator. Natural join between two or more relations will result in a set of all combinations of tuples where they have an equal common attribute.

**Example:**

**EMP**

| **Name** | **ID** | **Dept\_Name** |
| --- | --- | --- |
| A | 120 | IT |
| B | 125 | HR |
| C | 110 | Sales |
| D | 111 | IT |

**DEPT**

| **Dept\_Name** | **Manager** |
| --- | --- |
| Sales | Y |
| Production | Z |
| IT | A |

Natural join between EMP and DEPT with condition :

**EMP.Dept\_Name = DEPT.Dept\_Name**

**EMP ⋈ DEPT**

| **Name** | **ID** | **Dept\_Name** | **Manager** |
| --- | --- | --- | --- |
| A | 120 | IT | A |
| C | 110 | Sales | Y |
| D | 111 | IT | A |

1. **Conditional Join:**Conditional join works similarly to natural join. In natural join, by default condition is equal between common attributes while in conditional join we can specify any condition such as greater than, less than, or not equal.

**Example:**

**R**

| **ID** | **Sex** | **Marks** |
| --- | --- | --- |
| 1 | F | 45 |
| 2 | F | 55 |
| 3 | F | 60 |

**S**

| **ID** | **Sex** | **Marks** |
| --- | --- | --- |
| 10 | M | 20 |
| 11 | M | 22 |
| 12 | M | 59 |

Join between R and S with condition  **R.marks >= S.marks**

| **R.ID** | **R.Sex** | **R.Marks** | **S.ID** | **S.Sex** | **S.Marks** |
| --- | --- | --- | --- | --- | --- |
| 1 | F | 45 | 10 | M | 20 |
| 1 | F | 45 | 11 | M | 22 |
| 2 | F | 55 | 10 | M | 20 |
| 2 | F | 55 | 11 | M | 22 |
| 3 | F | 60 | 10 | M | 20 |
| 3 | F | 60 | 11 | M | 22 |
| 3 | F | 60 | 12 | M | 59 |

**Union operator in SQL**

The **UNION** operator could be used to find the result set or combination of two or more tables.

**Terms and Conditions for using UNION**

* Each table used within UNION must have the same number of columns.
* The columns must have the same data types.
* The columns in each table must be in the same order.

**Syntax:**

*SELECT columnnames FROM table1*

*UNION*

*SELECT columnnames FROM table2;*

***Example***

*Let’s assume we have two tables “Emp1” and “Emp2”;*

***Table1:***

*CREATE TABLE Emp1(*

*EmpID INT PRIMARY KEY,*

*Name VARCHAR(50),*

*Country VARCHAR(50),*

*Age int(2),*

*mob int(10)*

*);*

*-- Insert some sample data into the Customers table*

*INSERT INTO Emp1 (EmpID, Name,Country, Age, mob)*

*VALUES (1, 'Shubham', 'India','23','738479734'),*

*(2, 'Aman ', 'Australia','21','436789555'),*

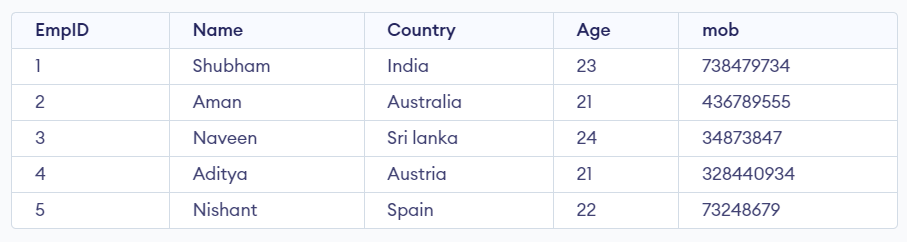
*(3, 'Naveen', 'Sri lanka','24','34873847'),*

*(4, 'Aditya', 'Austria','21','328440934'),*

*(5, 'Nishant', 'Spain','22','73248679');*

*Select \* from Emp1;*

***Output:***



***Table2***

*CREATE TABLE Emp2(*

*EmpID INT PRIMARY KEY,*

*Name VARCHAR(50),*

*Country VARCHAR(50),*

*Age int(2),*

*mob int(10)*

*);*

*-- Insert some sample data into the Customers table*

*INSERT INTO Emp2 (EmpID, Name,Country, Age, mob)*

*VALUES (1, 'Tommy', 'England','23','738985734'),*

*(2, 'Allen', 'France','21','43678055'),*

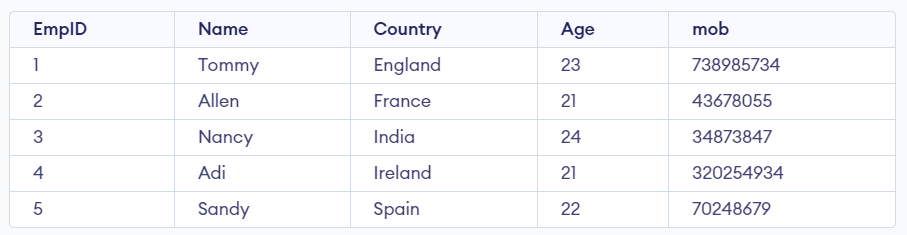
*(3, 'Nancy', 'India','24','34873847'),*

*(4, 'Adi', 'Ireland','21','320254934'),*

*(5, 'Sandy', 'Spain','22','70248679');*

*Select \* from Emp2;*

***Output:***



***SQL UNION Example***

*The below SQL statement finds the cities (only unique values) from both the “Table1” and the “Table2” tables:*

***Query:***

*SELECT Country FROM Emp1*

*UNION*

*SELECT Country FROM Emp2*

*ORDER BY Country;*

***Output:***



***SQL UNION ALL With WHERE***

*The following SQL statement returns the cities (duplicate values also) from both the “Geeks1” and the “Geeks2” tables:*

***Query:***

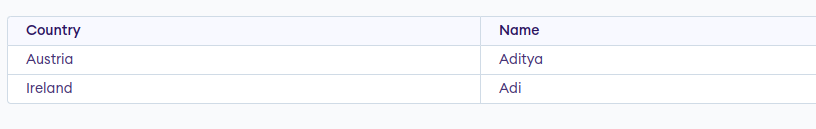
*SELECT Country, Name FROM Emp1 WHERE Name='Aditya'*

*UNION ALL*

*SELECT Country, Name FROM Emp2 WHERE Country='Ireland'*

*ORDER BY Country;*

***Output:***



**Intersect in SQL**

Intersect statement combines result-set of two or more select queries and returns only those values that are common in both the result set.

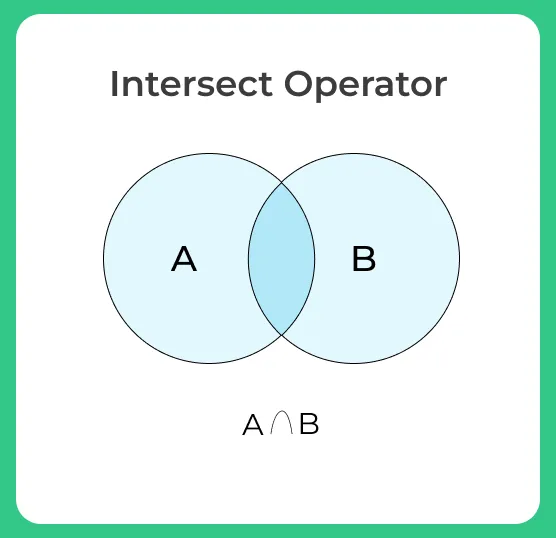
Simply INTERSECT returns the common rows are values common in both the data sets and will not consider duplicate values

**Syntax for INTERSECT**

SELECT column name(s) FROM table1  
***INTERSECT***  
SELECT column name(s) FROM table2;

Each SELECT statement within INTERSECT must have

* The same number of columns
* The columns must also have similar data types
* The columns in each SELECT statement must also be in the same order



**Example of INTERSECT**

The **First** table,

| **ID** | **NAME** |
| --- | --- |
| 1 | rishi |
| 2 | trish |
| 3 | mahi |

The **Second** table,

| **ID** | **NAME** |
| --- | --- |
| 2 | trish |
| 3 | Chester |
| 4 | mahi |

INTERSECT query will be,

SELECT name FROM First

***INTERSECT***

SELECT name FROM Second;

The result set table will look like

| **Name** |
| --- |
| trish |
| mahi |

**Another example of intersect**

Whenever ***intersect*** is applied on the select queries which contain***more than one column*** then the ***combination of both the values is considered as one tuple that is if both the values in the row are same am then only it is considered as common value***

SELECT id, name FROM First

***INTERSECT***

SELECT id, name FROM Second;

O/P

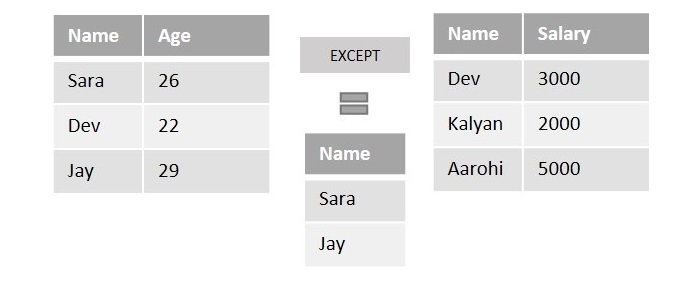
| **2** | **Trish** |
| --- | --- |

In our example combination of [2, Trish] is considered as one value while returning the common tuples

**Except**

The **EXCEPT** operator in SQL is used to retrieve the unique records that exist in the first table, not the common records of both tables. This operator acts as the opposite of the SQL UNION operator.

For better understanding consider two tables with records as shown in the following image −



If we perform the EXCEPT operator on the above two tables to retrieve the names, it will display the records only from the first table which are not in common with the records of the second table.

Here, “Dev” is common in both tables. So, the EXECPT operator will eliminate it and retrieves only “Sara” and “Jay” as output.

Syntax

Following is the syntax of the EXCEPT operator in SQL −

SELECT column1, column2,…, columnN

FROM table1, table2,…, tableN

[Conditions] //optional

EXCEPT

SELECT column1, column2,…, columnN

FROM table1, table2,…, tableN

[Conditions] //optional

**Note** − The number and order of columns in both SELECT statements should be the same.

Example

First of all, let us create a table named “**STUDENTS**” using the following query −

SQL> CREATE TABLE STUDENTS(

ID INT NOT NULL,

NAME VARCHAR(20) NOT NULL,

HOBBY VARCHAR(20) NOT NULL,

AGE INT NOT NULL,

PRIMARY KEY(ID)

);

Once the table is created, let us insert some values to the table using the query below −

SQL> INSERT INTO STUDENTS(ID, NAME, HOBBY, AGE) VALUES(1, 'Vijay', 'Cricket', 18);

INSERT INTO STUDENTS(ID, NAME, HOBBY, AGE) VALUES(2, 'Varun', 'Football', 26);

INSERT INTO STUDENTS(ID, NAME, HOBBY, AGE) VALUES(3, 'Surya', 'Cricket', 19);

INSERT INTO STUDENTS(ID, NAME, HOBBY, AGE) VALUES(4, 'Karthik', 'Cricket', 25);

INSERT INTO STUDENTS(ID, NAME, HOBBY, AGE) VALUES(5, 'Sunny', 'Football', 26);

INSERT INTO STUDENTS(ID, NAME, HOBBY, AGE) VALUES(6, 'Dev', 'Cricket', 23);

Let us verify whether the table “STUDENTS” is created or not using the following query −

SQL> SELECT \* FROM STUDENTS;

As we can see in the below output, the table has been created in the database.

+-----+----------+--------------+-------+

| ID | NAME | HOBBY | AGE |

+-----+----------+--------------+-------+

| 1 | Vijay | Cricket | 18 |

| 2 | Varun | Football | 26 |

| 3 | Surya | Cricket | 19 |

| 4 | Karthik | Cricket | 25 |

| 5 | Sunny | Football | 26 |

| 6 | Dev | Cricket | 23 |

+-----+----------+--------------+-------+

Let us create another table named “**ASSOCIATES**” using the following query −

SQL> CREATE TABLE ASSOCIATES(

ID INT NOT NULL,

NAME VARCHAR(20) NOT NULL,

SUBJECT VARCHAR(20) NOT NULL,

AGE INT NOT NULL,

HOBBY VARCHAR(20) NOT NULL,

PRIMARY KEY(ID)

);

Once the table is created, let us insert some values to the table using the query below −

SQL> INSERT INTO ASSOCIATES(ID, NAME, SUBJECT, AGE, HOBBY) VALUES(1, 'Naina', 'Maths', 24, 'Cricket');

INSERT INTO ASSOCIATES(ID, NAME, SUBJECT, AGE, HOBBY) VALUES(2, 'Varun', 'Physics', 26, 'Football');

INSERT INTO ASSOCIATES(ID, NAME, SUBJECT, AGE, HOBBY) VALUES(3, 'Dev', 'Maths', 23, 'Cricket');

INSERT INTO ASSOCIATES(ID, NAME, SUBJECT, AGE, HOBBY) VALUES(4, 'Priya', 'Physics', 25, 'Cricket');

INSERT INTO ASSOCIATES(ID, NAME, SUBJECT, AGE, HOBBY) VALUES(5, 'Aditya', 'Chemistry', 21, 'Cricket');

INSERT INTO ASSOCIATES(ID, NAME, SUBJECT, AGE, HOBBY) VALUES(6, 'Kalyan', 'Maths', 30, 'Football');

Let us verify whether the table “**ASSOCIATES**” is created or not using the following query −

SQL> SELECT \* FROM ASSOCIATES;

As we can see in the below output, the table has been created in the database.

+-----+----------+---------------+-------+----------+

| ID | NAME | SUBJECT | AGE | HOBBY |

+-----+----------+---------------+-------+----------+

| 1 | Naina | Mathematics | 24 | Cricket |

| 2 | Varun | Physics | 26 | Football |

| 3 | Dev | Mathematics | 23 | Cricket |

| 4 | Priya | Physics | 25 | Cricket |

| 5 | Adithya | Chemistry | 21 | Cricket |

| 6 | Kalyan | Mathematics | 30 | Football |

+-----+----------+--------------+-------+-----------+

Let us retrieve the records that are **only** unique in the first table using the below query −

SQL> SELECT NAME, HOBBY, AGE FROM STUDENTS

EXCEPT

SELECT NAME, HOBBY, AGE FROM ASSOCIATES

Output

When we execute the above query, the output is obtained as follows −

+-----------+--------------+-------+

| NAME | HOBBY | AGE |

+-----+----------+---------+-------+

| Karthik | Cricket | 25 |

| Sunny | Football | 26 |

| Surya | Cricket | 19 |

| Vijay | Cricket | 18 |

+-----------+--------------+-------+

**Nested queries**

* Nested query is one of the most useful functionalities of SQL
* Nested queries are useful when we want to write complex queries where one query uses the result from another query.
* Nested queries will have multiple SELECT statements nested together.
* A subquery is a SELECT statement nested within another SELECT statement.
* A nested query in SQL contains a query inside another query.
* The outer query will use the result of the inner query.
* For instance, a nested query can have two **SELECT** statements, one on the inner query and the other on the outer query.

**What are the Types of Nested Queries in SQL?**

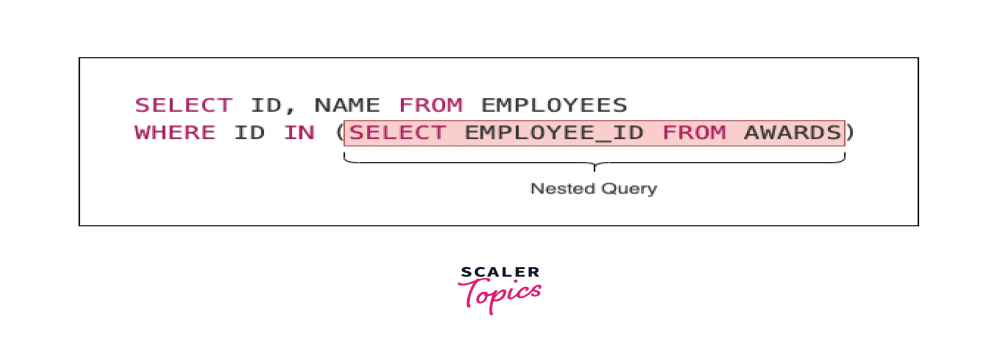
Nested queries in SQL can be classified into two different types:

* Independent Nested Queries
* Co-related Nested Queries

**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 such as **IN**, **NOT IN**, **ALL**, and **ANY** are used to write independent nested queries.

* The **IN** operator 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.
* 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.
* The **ALL** operator 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.
* 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**.



**Co-related Nested Queries**

* In co-related nested queries, the inner query uses the values from the outer query to execute the inner query for every row processed by the outer query.
* The co-related nested queries run slowly because the inner query is executed for every row of the outer query's result.

**Employees**

| **id** | **name** | **salary** | **role** |
| --- | --- | --- | --- |
| 1 | Augustine Hammond | 10000 | Developer |
| 2 | Perice Mundford | 10000 | Manager |
| 3 | Cassy Delafoy | 30000 | Developer |
| 4 | Garwood Saffen | 40000 | Manager |
| 5 | Faydra Beaves | 50000 | Developer |

**Awards**

| **id** | **employee\_id** | **award\_date** |
| --- | --- | --- |
| 1 | 1 | 2022-04-01 |
| 2 | 3 | 2022-05-01 |

**Independent Nested Queries**

**Example 1: IN**

Select all employees who won an award.

SELECT id, name FROM employees

WHERE id IN (SELECT employee\_id FROM awards);

**Output**

| **id** | **name** |
| --- | --- |
| 2 | Perice Mundford |
| 4 | Garwood Saffen |
| 5 | Faydra Beaves |

**Example 2: NOT IN**

Select all employees who never won an award.

SELECT id, name FROM employees

WHERE id NOT IN (SELECT employee\_id) FROM awards);

**Output**

| **id** | **name** |
| --- | --- |
| 1 | Augustine Hammond |
| 3 | Cassy Delafoy |

**Example 3: ALL**

Select all **Developers** who earn more than all the **Managers**

SELECT \* FROM employees

WHERE role = 'Developer'

AND salary > ALL (

SELECT salary FROM employees WHERE role = 'Manager'

);

**Output**

| **id** | **name** | **salary** | **role** |
| --- | --- | --- | --- |
| 5 | Faydra Beaves | 50000 | Developer |

**Explanation**

The developer with id **5** earns (50000) more than all the managers: **2** (10000) and **4** (40000)

**Example 4: ANY**

Select all **Developers** who earn more than any **Manager**

SELECT \* FROM employees

WHERE role = 'Developer'

AND salary > ANY (

SELECT salary FROM employees WHERE role = 'Manager'

);

**Output**

| **id** | **name** | **salary** | **role** |
| --- | --- | --- | --- |
| 5 | Faydra Beaves | 50000 | Developer |
| 3 | Cassy Delafoy | 30000 | Developer |

**Explanation**

The developers with id **3** and **5** earn more than any manager:

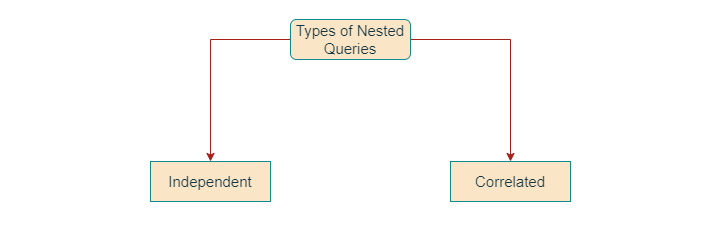
* The developer with id **3** earns (30000) more than the manager with id **2** (10000)
* The developer with id **5** earns (50000) more than the managers with id **2** (10000) and **4** (40000)

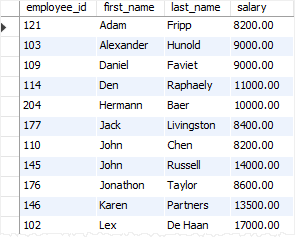
**Co-related Nested Queries**

**For Correlated Subquery:**

* SQL Correlated Sub queries are used to select data from a table referenced in the outer query.
* The sub query is known as a correlated because the sub query is related to the outer query.
* In this type of queries, a table alias (also called a correlation name) must be used to specify which table reference is to be used.
* Uncorrelated sub query executes the sub query first and provides the value to the outer query, whereas correlated sub query references a column in the outer query and executes the sub query once for each row in the outer query..

The alias is the pet name of a table which is brought about by putting directly after the table name in the FROM clause. This is suitable when anybody wants to obtain information from two separate tables.





Unlike a plain subquery, a correlated subquery is a subquery that uses the values from the outer query. Also, a correlated subquery may be evaluated once for each row selected by the outer query. Because of this, a query that uses a correlated subquery may be slow.

A correlated subquery is also known as a repeating subquery or a synchronized subquery.

SQL correlated subquery examples

Let’s see few more examples of the correlated subqueries to understand them better.

SQL correlated subquery in the WHERE clause example

The following query finds all employees whose salary is higher than the average salary of the employees in their departments:

SELECT

employee\_id,

first\_name,

last\_name,

salary,

department\_id

FROM

employees e

WHERE

salary > (SELECT

AVG(salary)

FROM

employees

WHERE

department\_id = e.department\_id)

ORDER BY

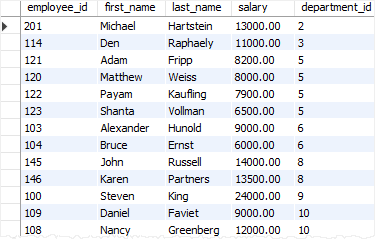
department\_id ,

first\_name ,

last\_name;

Code language: SQL (Structured Query Language) (sql)

Here is the output:



In this example, the outer query is:

SELECT

employee\_id,

first\_name,

last\_name,

salary,

department\_id

FROM

employees e

WHERE

salary >

...

Code language: SQL (Structured Query Language) (sql)

and the correlated subquery is:

SELECT

AVG( list\_price )

FROM

products

WHERE

category\_id = p.category\_id

Code language: SQL (Structured Query Language) (sql)

For each employee, the database system has to execute the correlated subquery once to calculate the average salary of the employees in the department of the current employee

**SET OPERATORS**

What are Set Operators?

SQL set operations are used for combining data from one or more tables

There are 3 set operations in SQL. They are

* UNION/UNION ALL
* INTERSECT
* MINUS

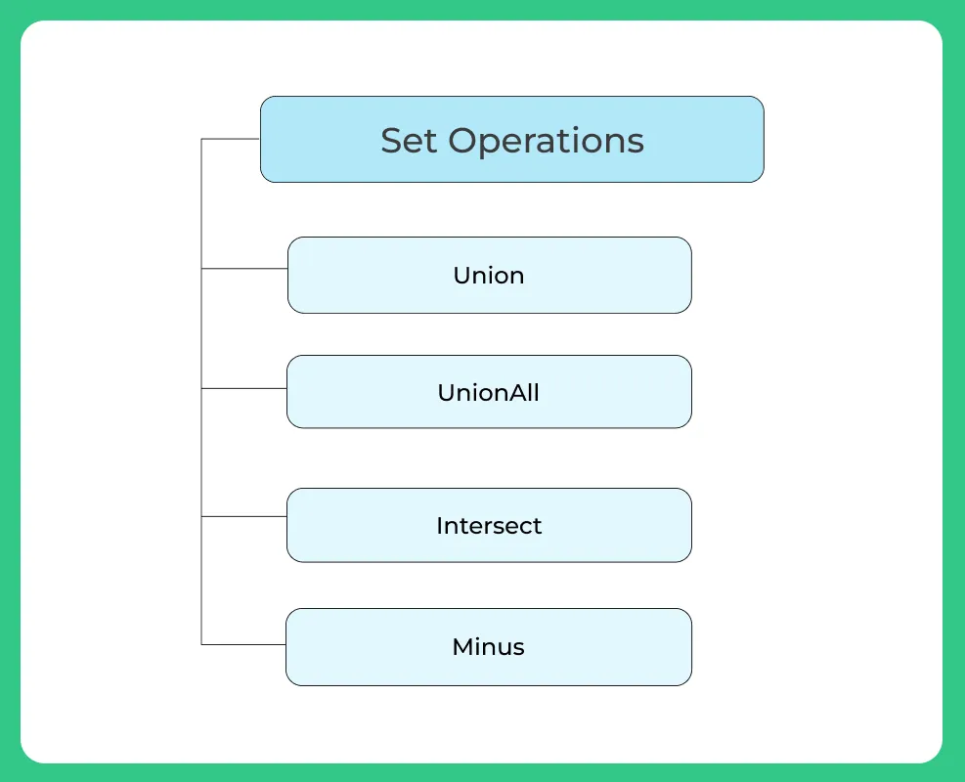
General Syntax for all SET operators

SELECT column name(s) FROM table1

*UNION/UNION ALL/INTERSECT/MINUS*

SELECT column name(s) FROM table2

Set operators combine the results of two component queries into a single result. Queries containing set operators are called compound queries.



Each SELECT statement that is used this set operators must follow these conditions

* The ***same number of columns***
* The columns must also have***similar data types***
* The columns in each SELECT statement must also be in the ***same order***
* **UNION/UNION ALL**
* Union clause used to combine the result-set of two or more select queries
* **Customers table:**

| **City** | **Country** |
| --- | --- |
| Thimpu | Germany |
| Hyderabad | India |
| Hyderabad | India |

* **Suppliers table:**

| **City** | **Country** |
| --- | --- |
| London | UK |
| California | USA |
| Texas | USA |

* **Example for UNION**
* SELECT City FROM Customers
* ***UNION***
* SELECT City FROM Suppliers
* ORDER BY City;
* **Output:**

| **City** |
| --- |
| Texas |
| Thimpu |
| London |
| Hyderabad |
| California |

**UNION all in SQL**

The only difference between UNION and UNION all Clause is that ***UNION removes duplicate values while comparing where UNION ALL allows duplicate values while combining***

**Example for UNION ALL**

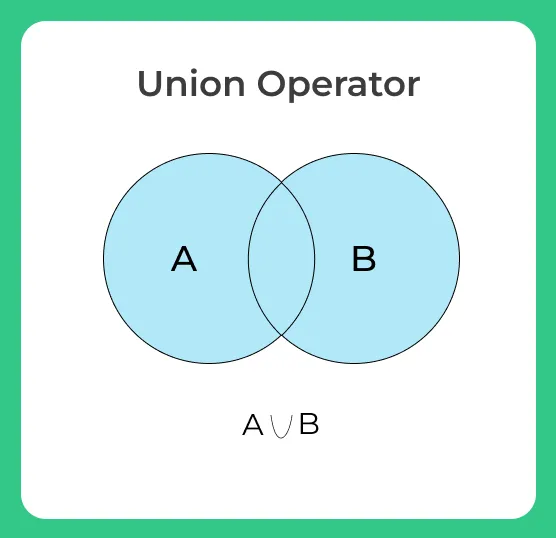
SELECT City, Country FROM Customers;

***UNION ALL***

SELECT City, Country FROM Suppliers

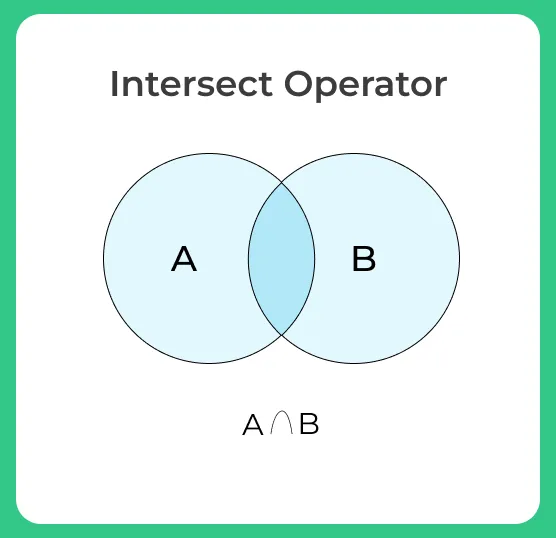
**Output:**

| **City** | **Country** |
| --- | --- |
| Texas | USA |
| Thimpu | Germany |
| Londona | UK |
| Hyderabad | India |
| California | USA |



**INTERSECT**

* Intersect statement combines result-set of two or more select queries and returns only those values that are common in both the result set
* Simply INTERSECT returns the common rows are values common in both the data sets and will not consider duplicate values



**Example of INTERSECT**

The **First** table,

| **ID** | **Name** |
| --- | --- |
| 1 | rishi |
| 2 | trish |
| 3 | mahi |

The **Second** table,

| **ID** | **Name** |
| --- | --- |
| 2 | trish |
| 3 | Chester |
| 4 | mahi |

**INTERSECT query will be,**

SELECT name FROM First

***INTERSECT***

SELECT name FROM Second;

The result set table will look like

| **Name** |
| --- |
| trish |
| mahi |

**MINUS**

* Minus operator is used to subtract the result set obtained by the first SELECT query from the result set obtained by the second SELECT query.
* Simply, we can say that MINUS operator will return only those rows which are unique in only first SELECT query and not those rows which are common to both first and second SELECT queries.

Consider the two tables

**Table1**

| **Name** | **Address** | **Age** | **Grade** |
| --- | --- | --- | --- |
| Priya | Hyderabad | 19 | A |
| Rahul | Chennai | 20 | B |
| Karthik | Mumbai | 21 | A |
| Payal | Delhi | 20 | B |

**Table2**

| **Name** | **Course** | **Age** | **Grade** |
| --- | --- | --- | --- |
| Divya | Java | 22 | B |
| Amitha | C++ | 19 | A |
| Harshitha | Python | 21 | A |
| Payal | Java | 20 | B |

**Example**

SELECT Name FROM Table1

***MINUS***

SELECT Name FROM Table2;

**Output:**

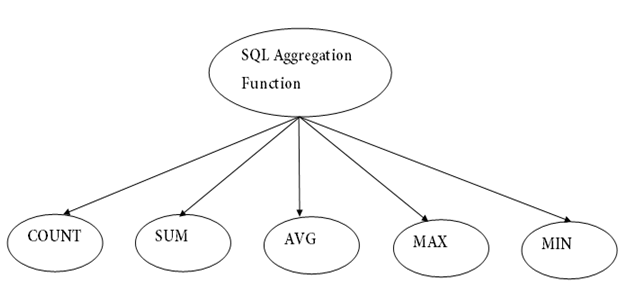
| **Name** |
| --- |
| Priya |
| Rahul |
| Karthik |

**Aggregate operators**

SQL Aggregate Functions

* SQL aggregation function is used to perform the calculations on multiple rows of a single column of a table. It returns a single value.
* It is also used to summarize the data.

Types of SQL Aggregation Function



* 1. COUNT function is used to Count the number of rows in a database table. It can work on both numeric and non-numeric data types.
* COUNT function uses the COUNT(\*) that returns the count of all the rows in a specified table. COUNT(\*) considers duplicate and Null.

COUNT FUNCTION

**Syntax**

1. COUNT(\*)
2. or
3. COUNT( [ALL|DISTINCT] expression )

**Sample table:**

**PRODUCT\_MAST**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **PRODUCT** | **COMPANY** | **QTY** | **RATE** | **COST** |
| Item1 | Com1 | 2 | 10 | 20 |
| Item2 | Com2 | 3 | 25 | 75 |
| Item3 | Com1 | 2 | 30 | 60 |
| Item4 | Com3 | 5 | 10 | 50 |
| Item5 | Com2 | 2 | 20 | 40 |
| Item6 | Cpm1 | 3 | 25 | 75 |
| Item7 | Com1 | 5 | 30 | 150 |
| Item8 | Com1 | 3 | 10 | 30 |
| Item9 | Com2 | 2 | 25 | 50 |
| Item10 | Com3 | 4 | 30 | 120 |

**count**

1. SELECT COUNT(\*)   FROM PRODUCT\_MAST;

**Output:**

10

**COUNT with WHERE**

1. SELECT COUNT(\*)   FROM PRODUCT\_MAST WHERE RATE>=20;

**Output:**

7

**COUNT() with DISTINCT**

1. SELECT COUNT(DISTINCT COMPANY)  FROM PRODUCT\_MAST;

**Output:**

3

**COUNT() with GROUP BY**

1. SELECT COMPANY, COUNT(\*)  FROM PRODUCT\_MAST  GROUP BY COMPANY;

**Output:**

Com1 5

Com2 3

Com3 2

**COUNT() with HAVING**

1. SELECT COMPANY, COUNT(\*)  FROM PRODUCT\_MAST  GROUP BY COMPANY

HAVING COUNT(\*)>2;

**Output:**

Com1 5

Com2 3

1. **SUM Function**

Sum function is used to calculate the sum of all selected columns. It works on numeric fields only.

**Syntax**

1. SUM()
2. or
3. SUM( [ALL|DISTINCT] expression )

**SUM()**

1. SELECT SUM(COST)  FROM PRODUCT\_MAST;

670

**SUM() with WHERE**

1. SELECT SUM(COST)  FROM PRODUCT\_MAST  WHERE QTY>3;

**Output:**

320

**SUM() with GROUP BY**

1. SELECT SUM(COST)  FROM PRODUCT\_MAST  WHERE QTY>3  GROUP BY COMPANY;

**Output:**

Com1 150

Com2 170

**SUM() with HAVING**

1. SELECT COMPANY, SUM(COST)  FROM PRODUCT\_MAST  GROUP BY COMPANY

HAVING SUM(COST)>=170;

**Output:**

Com1 335

Com3 170

**3. AVG function**

The AVG function is used to calculate the average value of the numeric type. AVG function returns the average of all non-Null values.

**Syntax**

1. **AVG()**

**or**

**AVG( [ALL|DISTINCT] expression )**

SELECT AVG(COST)  FROM PRODUCT\_MAST;

**Output:**

67.00

**4. MAX Function**

MAX function is used to find the maximum value of a certain column. This function determines the largest value of all selected values of a column.

**Syntax**

**MAX()**

**or**

**MAX( [ALL|DISTINCT] expression )**

SELECT MAX(RATE)  FROM PRODUCT\_MAST;

30

**5. MIN Function**

MIN function is used to find the minimum value of a certain column. This function determines the smallest value of all selected values of a column.

**Syntax**

**MIN()**

**or**

**MIN( [ALL|DISTINCT] expression )**

SELECT MIN(RATE)  FROM PRODUCT\_MAST;

**Output:**

10

**NULL**

* The term **NULL** in SQL is used to specify that a data value does not exist in the database.
* It is not the same as an empty string or a value of zero, and it signifies the absence of a value or the unknown value of a data field.

*Some common reasons why a value may be NULL −*

* The value may not be provided during the data entry.
* The value is not yet known.

*It is important to understand that you cannot use comparison operators such as “****=****”, “****<****”, or “****>****” with NULL values. This is because the NULL values are unknown and could represent any value. Instead, you must use “IS NULL” or “IS NOT NULL” operators to check if a value is NULL.*

**Syntax**

The basic syntax of **NULL** while creating a table.

SQL> CREATE TABLE CUSTOMERS( ID INT NOT NULL, NAME VARCHAR (20) NOT NULL,

AGE INT NOT NULL, ADDRESS CHAR (25) , SALARY DECIMAL (18, 2),

PRIMARY KEY (ID));

Here, **NOT NULL** signifies that column should always accept an explicit value of the given data type. There are two columns where we did not use NOT NULL, which means these columns could be NULL.

A field with a NULL value is the one that has been left blank during the record creation.

**Example**

Let us create a table with the name **CUSTOMERS** in the SQL database using the CREATE statement as shown in the query below −

SQL> CREATE TABLE CUSTOMERS( ID INT NOT NULL, NAME VARCHAR (20) NOT NULL, AGE INT NOT NULL, ADDRESS CHAR (25) , SALARY DECIMAL (18, 2), PRIMARY KEY (ID));

***Let us insert some values into the above created table using the following query −***

SQL> INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY) VALUES(1, 'Ramesh', '32', 'Ahmedabad', 2000);

INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY) VALUES(2, 'Khilan', '25', 'Delhi', 1500);

INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY) VALUES(3, 'kaushik', '23', 'Kota', 2000);

INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY) VALUES(4, 'Chaitali', '25', 'Mumbai', 6500);

INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY) VALUES(5, 'Hardik','27', 'Bhopal', 8500);

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

INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY) VALUES(7, 'Muffy', '24', 'Indore', NULL);

To verify whether the table CUSTOMERS is created or not, use the following query −

SQL> SELECT \* FROM CUSTOMERS;

The table is successfully created in the database.

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | |

| 7 | Muffy | 24 | Indore | |

+----+----------+-----+-----------+----------+

**IS NOT NULL Query**

Now, let us try to retrieve the records present in the table that are not null using the **IS NOT**

**NULL** operator −

SQL> SELECT ID, NAME, AGE, ADDRESS, SALARY FROM CUSTOMERS WHERE SALARY IS NOT NULL;

Output

The above query would produce the following result −

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

+----+----------+-----+-----------+----------+

**IS NULL Query**

Let us try to retrieve the records present in the table that are null using the **IS NULL** operator −

SQL> SELECT ID, NAME, AGE, ADDRESS, SALARY FROM CUSTOMERS WHERE SALARY IS NULL;

Output

The above query would produce the following result −

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 6 | Komal | 22 | MP | NULL |

| 7 | Muffy | 24 | Indore | NULL |

+----+----------+-----+-----------+----------+

**Updating NULL values in a table**

You can update the NULL values present in a table using the **UPDATE** statement in SQL. To do so, you can use the IS NULL operator in your WHERE clause to select the rows with NULL values and then set the new value using the SET keyword.

Example

Assume the previously created table and let us try to update the NULL value(s) in the present in the table using the **UPDATE** statement as shown below −

SQL> UPDATE CUSTOMERS SET SALARY = 9000 WHERE SALARY IS NULL;

Output

When you execute the above query, the output is obtained as follows −

Commands completed successfully.

Verification

Let us try to verify whether the specified record(s) in the table is updated or not using the following query −

SQL> SELECT \* FROM CUSTOMERS;

On executing the above query, the output is displayed as follows −

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 9000.00 |

| 7 | Muffy | 24 | Indore | 9000.00 |

+----+----------+-----+-----------+----------+

**SQL LOGICAL OPERATORS**

The Logical Operator is nothing but which returns the result in one form, i.e., either it will display the query is true, or the query is false. The results displayed to combine or merge more than one true or false data.

**The Logical Operators in SQL are as follows:**

1. SQL AND OPERATOR
2. SQL OR OPERATOR
3. SQL NOT OPERATOR
4. SQL BETWEEN OPERATOR
5. SQL IN OPERATOR
6. SQL LIKE OPERATOR

Let's understand each and every operator one by one with the help of examples. All the queries in the examples will be written using the MySQL database.

Consider we have an employees table with the following data:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **E\_ID** | **Name** | **Salary** | **City** | **Designation** | **Date\_of\_Joining** | **Age** |
| 1 | Sakshi Kumari | 50000 | Mumbai | Project Manager | 2021-06-20 | 24 |
| 2 | Tejaswini Naik | 75000 | Delhi | System Engineer | 2019-12-24 | 23 |
| 3 | Anuja Sharma | 40000 | Jaipur | Manager | 2021-08-15 | 26 |
| 4 | Anushka Tripathi | 90000 | Mumbai | Software Tester | 2021-06-13 | 24 |
| 5 | Rucha Jagtap | 45000 | Bangalore | Project Manager | 2020-08-09 | 23 |
| 6 | Rutuja Deshmukh | 60000 | Bangalore | Manager | 2019-07-17 | 26 |
| 7 | Swara Baviskar | 55000 | Jaipur | System Engineer | 2021-10-10 | 24 |
| 8 | Sana Sheik | 45000 | Pune | Software Engineer | 2020-09-10 | 26 |
| 9 | Swati Kumari | 50000 | Pune | Software Tester | 2021-01-01 | 25 |
| 10 | Mayuri Patel | 60000 | Mumbai | Project Manager | 2020-10-02 | 24 |
| 11 | Simran Khanna | 45500 | Kolhapur | HR | 2019-01-02 | 26 |
| 12 | Shivani Wagh | 50500 | Delhi | Software Developer | 2016-09-10 | 25 |
| 13 | Kiran Maheshwari | 50000 | Nashik | HR | 2013-12-12 | 23 |
| 14 | Tejal Jain | 40000 | Delhi | Project Manager | 2017-11-10 | 25 |
| 15 | Mohini Shah | 38000 | Pune | Software Developer | 2019-03-05 | 20 |

1. **SQL AND Operator**

The SQL AND operator is used with the where clause in the SQL Query. AND operator in SQL returns only those records which satisfy both the conditions in the SQL query.

Let's understand the below example, which explains how to execute AND operator in an SQL query.

Example:

Write a query to retrieve only those records of employees from the employees table where the designation is 'Project Manager' and the City to which the employee belongs to is Mumbai.

**Query:**

**SELECT** \* **FROM** employees **WHERE** City = "Mumbai" AND Designation = "Project Manager";

Here we have written a SELECT query with a WHERE clause on the City column and Designation column with 'AND' operator in between both the conditions. Any record in the employees table that meets both conditions, i.e., the city to which the employee belongs is Mumbai, and their designation is Project Manager, will only be considered in output.

You will get the following output:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **E\_ID** | **Name** | **Salary** | **City** | **Designation** | **Date\_of\_Joining** | **Age** |
| 1 | Sakshi Kumari | 50000 | Mumbai | Project Manager | 2021-06-20 | 24 |
| 10 | Mayuri Patel | 60000 | Mumbai | Project Manager | 2020-10-02 | 24 |

There are only two records in the employees table whose city name is equal to 'Mumbai' and designation name is equal to 'Project Manager'.

1. **SQL BETWEEN Operator**

This operator displays the records which fall between the given ranges in the SQL query. The results of the BETWEEN operator include begin and end values of the given range.

Let's understand the below example, which explains how to execute BETWEEN operator in an SQL query.

**Example:**

Write a query to retrieve only those records of an employee from the employees table where employee salary lies between 50000 to 90000.

**Query:**

**SELECT** \* **FROM** employees **WHERE** Salary BETWEEN 50000 AND 90000;

Here we have written a SELECT query with a WHERE clause on the Salary column with the 'BETWEEN' operator. BETWEEN operator is followed by beginning and end values 50000 and 90000 respectively with 'AND' operator in between. Any record in the employees table that meets the condition, i.e., the employee's salary is between 50000 and 90000, will only be considered in output.

You will get the following output:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **E\_ID** | **Name** | **Salary** | **City** | **Designation** | **Date\_of\_Joining** | **Age** |
| 1 | Sakshi Kumari | 50000 | Mumbai | Project Manager | 2021-06-20 | 24 |
| 2 | Tejaswini Naik | 75000 | Delhi | System Engineer | 2019-12-24 | 23 |
| 4 | Anushka Tripathi | 90000 | Mumbai | Software Tester | 2021-06-13 | 24 |
| 6 | Rutuja Deshmukh | 60000 | Bangalore | Manager | 2019-07-17 | 26 |
| 7 | Swara Baviskar | 55000 | Jaipur | System Engineer | 2021-10-10 | 24 |
| 9 | Swati Kumari | 50000 | Pune | Software Tester | 2021-01-01 | 25 |
| 10 | Mayuri Patel | 60000 | Mumbai | Project Manager | 2020-10-02 | 24 |
| 12 | Shivani Wagh | 50500 | Delhi | Software Developer | 2016-09-10 | 25 |
| 13 | Kiran Maheshwari | 50000 | Nashik | HR | 2013-12-12 | 23 |

There are nine records in the employees table whose salary falls between 50000 to 90000.

1. **SQL OR Operator**

The SQL OR operator is used with the where clause in an SQL Query. OR operator in SQL returns only those records that satisfy any of the conditions in the SQL query.

Let's understand the below example, which explains how to execute OR operator an SQL query.

**Example:**

Write a query to retrieve only those records of employees from the employees table where the employee's designation is 'System Engineer' or the city to which the employee belongs is Mumbai.

**Query:**

**SELECT** \* **FROM** employees **WHERE** Designation = "System Engineer" OR City = "Mumbai”

Here we have written a SELECT query with a WHERE clause on the City column and Designation column with the 'OR' operator in between both the conditions. Any record in the employees table that meets any of the conditions, i.e., the city to which the employee belongs is Mumbai, or their designation is System Engineer, will only be considered in output.

You will get the following output:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **E\_ID** | **Name** | **Salary** | **City** | **Designation** | **Date\_of\_Joining** | **Age** |
| 1 | Sakshi Kumari | 50000 | Mumbai | Project Manager | 2021-06-20 | 24 |
| 2 | Tejaswini Naik | 75000 | Delhi | System Engineer | 2019-12-24 | 23 |
| 4 | Anushka Tripathi | 90000 | Mumbai | Software Tester | 2021-06-13 | 24 |
| 7 | Swara Baviskar | 55000 | Jaipur | System Engineer | 2021-10-10 | 24 |
| 10 | Mayuri Patel | 60000 | Mumbai | Project Manager | 2020-10-02 | 24 |

There are only five records in the employees table whose city name is equal to 'Mumbai' or the employee's designation is equal to 'System Engineer'.

1. **SQL IN Operator**

When we want to check for one or more than one value in a single SQL query, we use IN operator with the WHERE clause in a SELECT query.

Let's understand the below example, which explains how to execute IN operator in an SQL query.

**Example:**

Write a query to retrieve only those records of employees from the employees table where the city to which the employee belongs to is either Mumbai, Bangalore, or Pune.

**Query:**

**SELECT** \* **FROM** employees **WHERE** City IN ("Mumbai", "Bangalore", "Pune");

Here we have written a SELECT query with a WHERE clause on the City column followed by IN operator. Since we wanted only those records that belongs to Mumbai, Bangalore, or Pune, we have passed Mumbai, Bangalore, and Pune as parameters to the IN operator. So, if the City value of any record matches with the places passed to the IN operator will only be considered in output.

You will get the following output:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **E\_ID** | **Name** | **Salary** | **City** | **Designation** | **Date\_of\_Joining** | **Age** |
| 1 | Sakshi Kumari | 50000 | Mumbai | Project Manager | 2021-06-20 | 24 |
| 4 | Anushka Tripathi | 90000 | Mumbai | Software Tester | 2021-06-13 | 24 |
| 5 | Rucha Jagtap | 45000 | Bangalore | Project Manager | 2020-08-09 | 23 |
| 6 | Rutuja Deshmukh | 60000 | Bangalore | Manager | 2019-07-17 | 26 |
| 8 | Sana Sheik | 45000 | Pune | Software Engineer | 2020-09-10 | 26 |
| 9 | Swati Kumari | 50000 | Pune | Software Tester | 2021-01-01 | 25 |
| 10 | Mayuri Patel | 60000 | Mumbai | Project Manager | 2020-10-02 | 24 |
| 15 | Mohini Shah | 38000 | Pune | Software Developer | 2019-03-05 | 20 |

There are only eight records in the employees table where the city to which the employee belongs is either Mumbai, Bangalore, or Pune.

1. **SQL NOT Operator**

NOT operator in SQL shows those records from the table where the criteria is not met. NOT operator is used with where clause in a SELECT query.

Let's understand the below example, which explains how to execute NOT operator in SQL query.

**Example:**

Write a query to retrieve only those records of employees from the employees table where the employee's designation is not Project Manager.

**Query:**

**SELECT** \* **FROM** employees **WHERE** NOT Designation = "Project Manager";

Here we have written a SELECT query with a WHERE clause on the Designation column followed by NOT operator. Since we wanted only those records whose designation is other than a project manager, we have given the designation value as Project Manager to the NOT operator. So, if the designation value of any record does not match with the value given to the NOT operator will only be considered in output.

You will get the following output:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **E\_ID** | **Name** | **Salary** | **City** | **Designation** | **Date\_of\_Joining** | **Age** |
| 2 | Tejaswini Naik | 75000 | Delhi | System Engineer | 2019-12-24 | 23 |
| 3 | Anuja Sharma | 40000 | Jaipur | Manager | 2021-08-15 | 26 |
| 4 | Anushka Tripathi | 90000 | Mumbai | Software Tester | 2021-06-13 | 24 |
| 6 | Rutuja Deshmukh | 60000 | Bangalore | Manager | 2019-07-17 | 26 |
| 7 | Swara Baviskar | 55000 | Jaipur | System Engineer | 2021-10-10 | 24 |
| 8 | Sana Sheik | 45000 | Pune | Software Engineer | 2020-09-10 | 26 |
| 9 | Swati Kumari | 50000 | Pune | Software Tester | 2021-01-01 | 25 |
| 11 | Simran Khanna | 45500 | Kolhapur | HR | 2019-01-02 | 26 |
| 12 | Shivani Wagh | 50500 | Delhi | Software Developer | 2016-09-10 | 25 |
| 13 | Kiran Maheshwari | 50000 | Nashik | HR | 2013-12-12 | 23 |
| 15 | Mohini Shah | 38000 | Pune | Software Developer | 2019-03-05 | 20 |

There are eleven records in the employees table whose designation is not a project manager.

1. **SQL LIKE Operator**

LIKE Operator in SQL displays only those data from the table which matches the pattern specified in the query. Percentage (%) and underscore (\_) are the two wildcard operators used with LIKE Operator to perform pattern matching tasks.

Let's understand the below example, which explains how to execute the LIKE operator in an SQL query.

**Example:**

Write a query to retrieve only those records of employees from the employees table whose salary starts with the digit 5.

**Query:**

**SELECT** \* **FROM** employees **WHERE** Salary LIKE "5%";

Here we have written a SELECT query with a WHERE clause on the Salary column followed by the LIKE operator. Since we wanted only those records whose salary starts with the digit 5, we have given the value to the LIKE operator as '5%'. So, if the salary value of any record starts with the digit 5, followed by any other digit will only be considered in output.

You will get the following output:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **E\_ID** | **Name** | **Salary** | **City** | **Designation** | **Date\_of\_Joining** | **Age** |
| 1 | Sakshi Kumari | 50000 | Mumbai | Project Manager | 2021-06-20 | 24 |
| 7 | Swara Baviskar | 55000 | Jaipur | System Engineer | 2021-10-10 | 24 |
| 9 | Swati Kumari | 50000 | Pune | Software Tester | 2021-01-01 | 25 |
| 12 | Shivani Wagh | 50500 | Delhi | Software Developer | 2016-09-10 | 25 |
| 13 | Kiran Maheshwari | 50000 | Nashik | HR | 2013-12-12 | 23 |

There are five records in the employees table whose salary starts with the digit 5.

**Outer joins in sql**

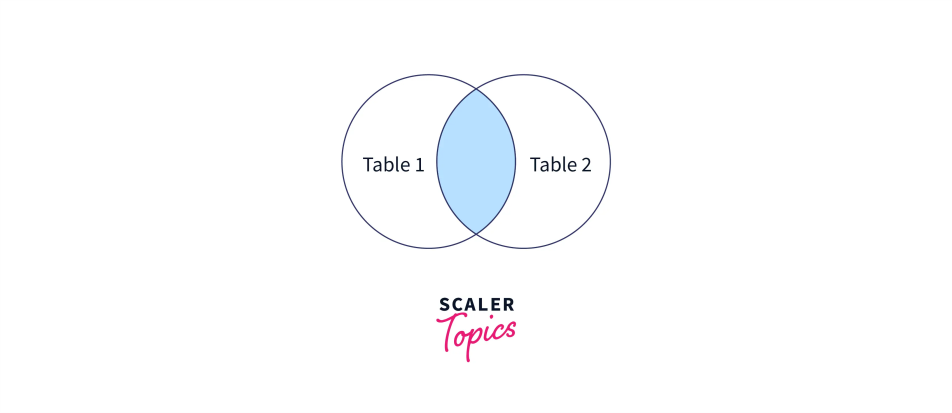
**What is a JOIN in SQL?**

Joins in SQL are used to combine data and rows from two or more tables based on a common column or a field. There are two main types of joins in SQL: inner joins and outer joins.

Inner joins return common information between tables and an outer join returns information in the resulting table that the inner join returns along with the information that is not common with the other table.

**What is INNER JOIN?**

The inner join in SQL will return the common or the matching records between the tables in the resulting table. It can be represented as:



That green area in the above diagram is the common records from the two tables, which will be resulted in an output.

The **syntax** for inner join in SQL is:

**SELECT \* FROM table\_name\_1 INNER JOIN table\_name\_2 ON table\_name\_1.common\_field = table\_name\_2.common\_field;**

**OR**

**SELECT \* FROM table\_name\_1 JOIN table\_name\_2 ON table\_name\_1.common\_field = table\_name\_2.common\_field;**

* Let's suppose we have two tables, one is storing the student roll number and student name
* The other table is storing the student roll number and the student marks.
* In this example, the roll number field is common in both tables and for the result, we need to consider the data in both tables (name and marks) based on a common field (in our case its roll number).
* So, we will apply the inner join on both tables and will return the result in the output table.

Firstly, let's check the data in both tables by running the SELECT query:

**SELECT \* FROM student\_detials\_1**

The **output** of the above table would be:

| **roll\_number** | **name** |
| --- | --- |
| 1 | Meenakshi |
| 2 | Manjili |
| 3 | Sheetal |
| 4 | Apoorva |
| 5 | Kitto |
| 6 | Nitin |
| 7 | Soumya |

**SELECT \* FROM student\_detials\_2**

The output of the above table would be:

| **roll\_number** | **marks** |
| --- | --- |
| 1 | 89 |
| 3 | 74 |
| 4 | 100 |
| 6 | 29 |

Now, let's apply inner join to both tables:

**SELECT student\_details\_1.name, student\_details\_2.marks**

**FROM student\_details\_1**

**INNER JOIN student\_details\_2**

**ON student\_details\_1.roll\_number = student\_details\_2.roll\_number;**

The **output** to the above join query is:

| **Name** | **marks** |
| --- | --- |
| Meenakshi | 89 |
| Sheetal | 74 |
| Apoorva | 100 |
| Nitin | 29 |

The table includes the names of only those students who are common in both tables.

**What is OUTER JOIN?**

An outer join returns a set of records (or rows) that includes those returned by the inner join, along with other rows for which no corresponding match was found in the other table.

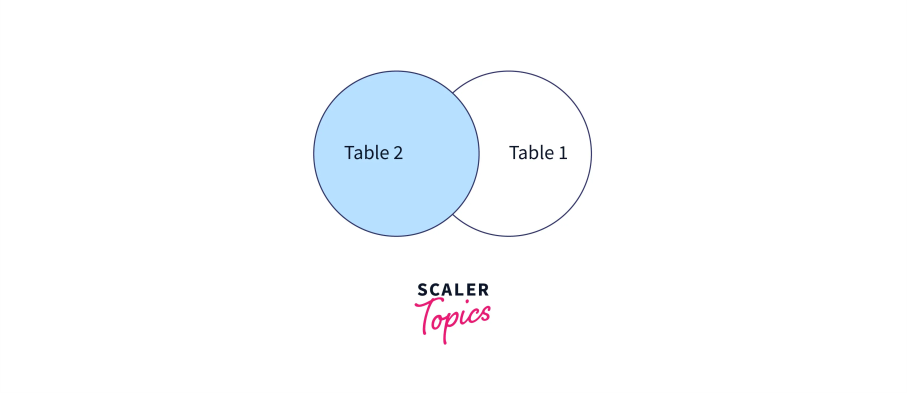
There are three types of outer joins:

1. Left outer join (or left join)
2. Right outer join (or right join)
3. Full Outer Join (or Full Join)

Outer joins are used when you want to return all data, not just the interrelated data. The next few sections of this article will describe each of the outer joins.

**Left outer join**

The left outer join is one of the types of outer join which returns all the records from the left table and only the matching and related records from the right table. The left join or the left outer join is the combination of the result of the inner join plus all the records of the left table.



The **syntax** for the left outer join is as shown below:

**SELECT \* FROM table\_name\_1**

**LEFT JOIN Table\_name\_2**

**ON Table\_name\_1.Common\_column\_Name=table\_name\_2.Common\_column\_name;**

Let's suppose we have two tables, one storing the student roll number and student name, and the other storing the student roll number and the student's hometown name. In this example, the roll number field is common in both tables and for the result, we need to consider the data in both tables (name and city) based on a common field (in our case its roll number). We will apply the left join on both tables and see what the result we get in the output table is.

Firstly, let's check the data in both tables by running the SELECT query:

SELECT \* FROM student\_detials\_1

The output of the above table would be:

| **roll\_number** | **name** |
| --- | --- |
| 1 | Meenakshi |
| 2 | Manjili |
| 3 | Sheetal |
| 4 | Apoorva |
| 5 | Kitto |
| 6 | Nitin |
| 7 | Soumya |

SELECT \* FROM student\_detials\_2

The **output** of the above table would be:

| **roll\_number** | **city** |
| --- | --- |
| 1 | Ajmer |
| 3 | Bangalore |
| 5 | Udaipur |
| 6 | Chittor |
| 10 | Raipur |

Now, let's apply the left outer join:

**SELECT student\_details\_1.name, student\_details\_2.city FROM student\_details\_1**

**LEFT JOIN student\_details\_2**

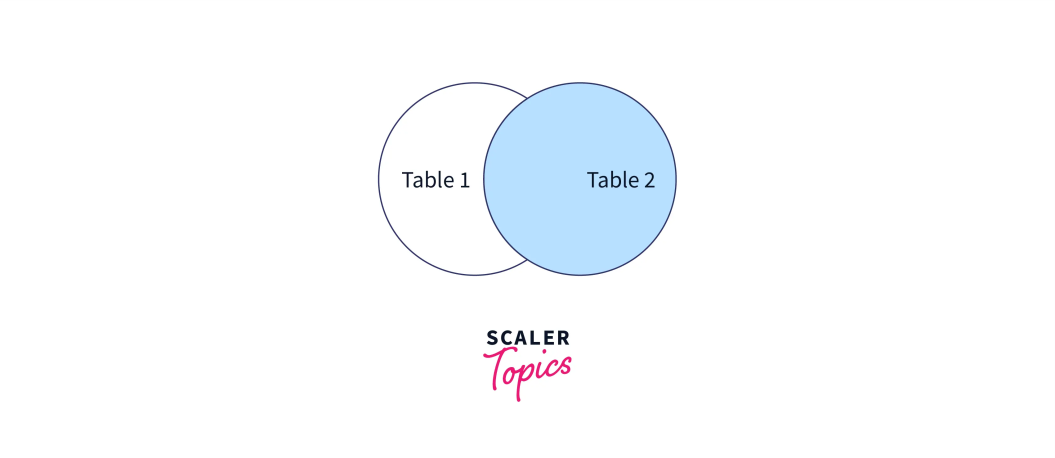
**ON student\_details\_1.roll\_number = student\_details\_2.roll\_number;**

The **output** to the above join query is:

| **name** | **city** |
| --- | --- |
| Meenakshi | Ajmer |
| Manjili | NULL |
| Sheetal | Bangalore |
| Apoorva | NULL |
| Kitto | Udaipur |
| Nitin | Chittor |
| Soumya | NULL |

**Right outer join**

Right, join is another type of outer join, which returns all the records from the right table and only the matching records from the left table. The right join or the right outer join is the combination of the result of the inner join plus all the records of the right table.



**The syntax for the right outer join is as shown below:**

**SELECT \* FROM table\_name\_1**

**RIGHT JOIN Table\_name\_2**

**ON Table\_name\_1.Common\_column\_Name=table\_name\_2.Common\_column\_name;**

Let's take the same example and apply the right outer join to see the results in the resultant output table.

**SELECT student\_details\_1.name, student\_details\_2.city FROM student\_details\_1**

**RIGHT JOIN student\_details\_2**

**ON student\_details\_1.roll\_number = student\_details\_2.roll\_number;**

The output to the above right join query is:

| **name** | **city** |
| --- | --- |
| Meenakshi | Ajmer |
| Sheetal | Bangalore |
| kitto | Udaipur |
| Nitin | Chittor |
| NULL | Raipur |

**Full Join**

The full outer join returns all the records from both tables when a match is in either table. It is the combination of the left outer join and the right outer join.

It is also known as **cross join**. Basically, a combination of two tables is provided in this join.

The syntax for full outer join is as shown below:

**SELECT \* FROM table\_name\_1**

**FULL OUTER JOIN Table\_name\_2**

**ON Table\_name\_1.Common\_column\_Name=table\_name\_2.Common\_column\_name;**

In the above example when we apply full outer join, the results will be displayed as:

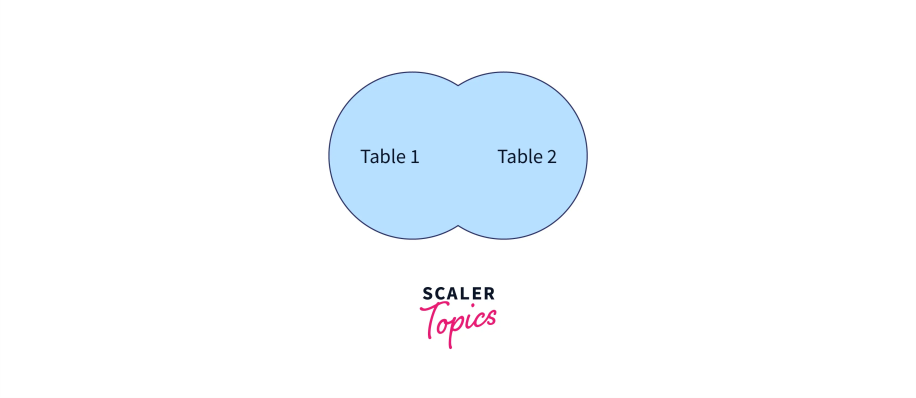
**SELECT student\_details\_1.name, student\_details\_2.City FROM student\_details\_1**

**FULL OUTER JOIN student\_details\_2**

**ON student\_details\_1.roll\_number = student\_details\_2.roll\_number;**

The **output** to the above full outer join query is:

| **name** | **city** |
| --- | --- |
| Meenakshi | Ajmer |
| Manjili | NULL |
| Sheetal | Bangalore |
| Apoorva | NULL |
| kitto | Udaipur |
| Nitin | Chittor |
| Soumya | NULL |
| NULL | Raipur |



**Disallowing NULL in SQL**

**SQL NOT NULL Constraint**

* By default, a column can hold NULL values.
* The NOT NULL constraint enforces a column to NOT accept NULL values.
* This enforces a field to always contain a value, which means that you cannot insert a new record, or update a record without adding a value to this field.

**SQL NOT NULL on CREATE TABLE**

The following SQL ensures that the "ID", "LastName", and "FirstName" columns will NOT accept NULL values when the "Persons" table is created:

**Example**

CREATE TABLE Persons ( ID int NOT NULL,  LastName varchar(255) NOT NULL,

FirstName varchar(255) NOT NULL,  Age int);

**Triggers in SQL**

* A trigger is a stored procedure in a database that automatically invokes whenever a special event in the database occurs
* . For example, a trigger can be invoked when a row is inserted into a specified table or when specific table columns are updated
* In simple words a trigger is a collection of [SQL](https://www.geeksforgeeks.org/sql-tutorial/)statements with particular names that are stored in system memory.
* It belongs to a specific class of stored procedures that are automatically invoked in response to database server events.
* Every trigger has a table attached to it.
* Because a trigger cannot be called directly, unlike a stored procedure, it is referred to as a special procedure.
* A trigger is automatically called whenever a data modification event against a table takes place, which is the main distinction between a trigger and a procedure. On the other hand, a stored procedure must be called directly.

**The following are the key differences between triggers and stored procedures:**

1. Triggers cannot be manually invoked or executed.
2. There is no chance that triggers will receive parameters.
3. A transaction cannot be committed or rolled back inside a trigger.

**Syntax:**

*create trigger [trigger\_name]*

*[before | after]*

*{insert | update | delete}*

*on [table\_name]*

*[for each row]*

*[trigger\_body]*

**Explanation of Syntax**

1. Create trigger [trigger\_name]: Creates or replaces an existing trigger with the trigger\_name.
2. [before | after]: This specifies when the trigger will be executed.
3. {insert | update | delete}: This specifies the DML operation.
4. On [table\_name]: This specifies the name of the table associated with the trigger.
5. [for each row]: This specifies a row-level trigger, i.e., the trigger will be executed for each affected row.
6. [trigger\_body]: This provides the operation to be performed as the trigger is fired

**Why Do We Employ Triggers?**

When we need to carry out some actions automatically in certain desirable scenarios, triggers will be useful. For instance, we need to be aware of the frequency and timing of changes to a table that is constantly changing. In such cases, we could create a trigger to insert the required data into a different table if the primary table underwent any changes.

**Different Trigger Types in SQL Server**

Two categories of triggers exist:

1. DDL Trigger
2. DML Trigger
3. Logon Triggers

**DDL Triggers**

The Data Definition Language (DDL) command events such as Create\_table, Create\_view, drop\_table, Drop\_view, and Alter\_table cause the DDL triggers to be activated.

**create tigger safety** **on database**

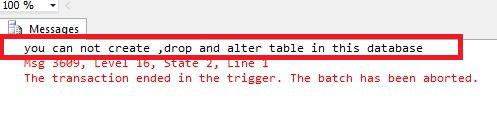
for

**create\_table,alter\_table,drop\_table**

as

**print 'you can not create,drop and alter tab**

**Output:**



**DML Triggers**

The Data uses manipulation Language (DML) command events that begin with Insert, Update, and Delete set off the DML triggers. corresponding to insert\_table, update\_view, and delete\_table.

**SQL Server**

create trigger deep

on **emp**

for

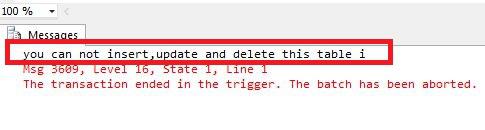
insert,update ,delete

**as**

print 'you can not insert,update and delete this table i'

rollback;

**Output:**



**Logon Triggers**

logon triggers are fires in response to a LOGON event. When a user session is created with a SQL Server instance after the authentication process of logging is finished but before establishing a user session, the LOGON event takes place. As a result, the PRINT statement messages and any errors generated by the trigger will all be visible in the SQL Server error log. Authentication errors prevent logon triggers from being used. These triggers can be used to track login activity or set a limit on the number of sessions that a given login can have in order to audit and manage server sessions.

**BEFORE and AFTER Trigger**

BEFORE triggers run the trigger action before the triggering statement is run. AFTER triggers run the trigger action after the triggering statement is run.

**Example**

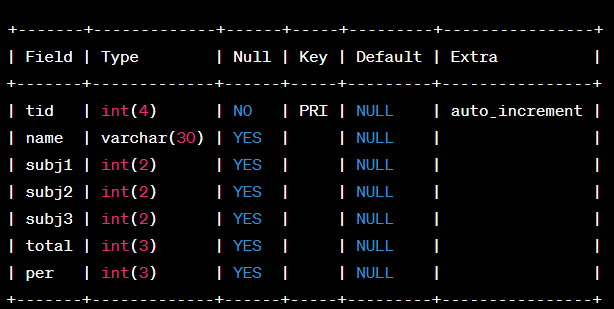
Given Student Report Database, in which student marks assessment is recorded. In such a schema, create a trigger so that the total and percentage of specified marks are automatically inserted whenever a record is inserted.

Here, a trigger will invoke before the record is inserted so BEFORE Tag can be used.

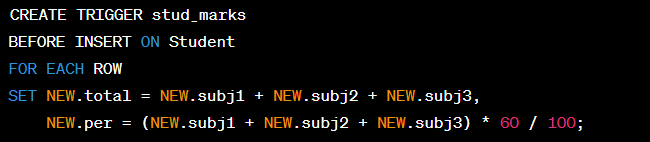
**Suppose the Database Schema**

**Query**

mysql>>desc Student;

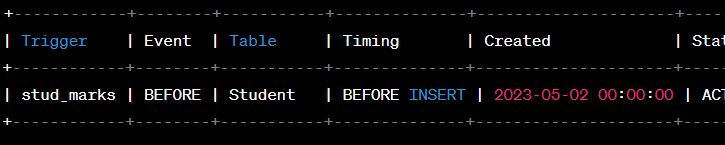


SQL Trigger to the problem statement.



Above SQL statement will create a trigger in the student database in which whenever subjects marks are entered, before inserting this data into the database, the trigger will compute those two values and insert them with the entered values. i.e.

**Output**



In this way, triggers can be created and executed in the databases.

**Advantage of Triggers**

The benefits of using triggers in SQL Server include the following:

1. Database object rules are established by triggers, which cause changes to be undone if they are not met.
2. The trigger will examine the data and, if necessary, make changes.
3. We can enforce data integrity thanks to triggers.
4. Data is validated using triggers before being inserted or updated.
5. Triggers assist us in maintaining a records log.
6. Due to the fact that they do not need to be compiled each time they are run, triggers improve the performance of SQL queries.
7. The client-side code is reduced by triggers, saving time and labor.
8. Trigger maintenance is simple.

**Disadvantage of Triggers**

The drawbacks of using triggers in SQL Server include the following:

1. Only triggers permit the use of extended validations.
2. Automatic triggers are used, and the user is unaware of when they are being executed. Consequently, it is difficult to troubleshoot issues that arise in the database layer.
3. The database server’s overhead may increase as a result of triggers.
4. In a single CREATE TRIGGER statement, we can specify the same trigger action for multiple user actions, such as INSERT and UPDATE.
5. Only the current database is available for creating triggers, but they can still make references to objects outside the database.