Module 4:
Structured Query Language (SQL)

4.2 Outline:

- Set and string operations
- Aggregate function-group by, having clause
- Views in SQL
- Joins, Nested and Complex Queries
- Triggers

SET Operations

- Set operations are supported by SQL to be performed on table data using **Set Operators**
- In order to execute set operations, two queries must be “set compatible”
  - Both relations have same number of columns
  - Corresponding column are data type compatible
- Queries which contain set operators are called **compound queries.**
- The different Set Operators are as follows:
  - Union
  - Union All
  - Intersect
  - Minus (Except)

Union Operator

- UNION combines the results of two or more SELECT statements.
- After performing the UNION operation, the **duplicate rows will be eliminated from the results.**

- **Syntax:**

```
SELECT expression_1, expression_2, ... , expression_n FROM table_1
UNION
SELECT expression_1, expression_2, ... , expression_n FROM table_2
```

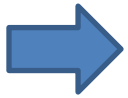
Union Operator

Customer

first_name	last_name
Stephen	Jones
Mark	Smith
Denise	King
Paula	Johnson
Richard	Archer

Employee

first_name	last_name
Christina	Jones
Michael	McDonald
Paula	Johnson
Stephen	Jones
Richard	Smith



```
SELECT first_name, last_name
FROM customer
UNION
SELECT first_name, last_name
FROM employee;
```

Result:

first_name	last_name
Stephen	Jones
Mark	Smith
Denise	King
Paula	Johnson
Richard	Archer
Christina	Jones
Michael	McDonald
Richard	Smith

Union ALL Operator

- UNION ALL is also used to combine the results of two or more SELECT statements.
- After performing the UNION ALL operation, the **duplicate rows will not be eliminated** from the results, and all the data is displayed in the result without removing the duplication.

- **Syntax**:

```
SELECT expression_1, expression_2, ... , expression_n FROM table_1
                              UNION ALL
SELECT expression_1, expression_2, ... , expression_n FROM table_2
```

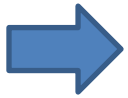
Union ALL Operator

Customer

first_name	last_name
Stephen	Jones
Mark	Smith
Denise	King
Paula	Johnson
Richard	Archer

Employee

first_name	last_name
Christina	Jones
Michael	McDonald
Paula	Johnson
Stephen	Jones
Richard	Smith



```
SELECT first_name, last_name
FROM customer
UNION ALL
SELECT first_name, last_name
FROM employee;
```

Result:

first_name	last_name
Stephen	Jones
Mark	Smith
Denise	King
Paula	Johnson
Richard	Archer
Christina	Jones
Michael	McDonald
Paula	Johnson
Stephen	Jones
Richard	Smith

Intersect Operator

- The INTERSECT operator allows you to find the results that exist in both queries.
- After performing the INTERSECT operation, the data/records which are common in both the SELECT statements are returned.

- **Syntax**:

```
SELECT expression_1, expression_2, ... , expression_n FROM table_1
INTERSECT
SELECT expression_1, expression_2, ... , expression_n FROM table_2
```

Intersect Operator

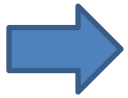
Customer

first_name	last_name
Stephen	Jones
Mark	Smith
Denise	King
Paula	Johnson
Richard	Archer

Employee

first_name	last_name
Christina	Jones
Michael	McDonald
Paula	Johnson
Stephen	Jones
Richard	Smith

```
SELECT first_name, last_name
FROM customer
INTERSECT
SELECT first_name, last_name
FROM employee;
```



Result:

first_name	last_name
Stephen	Jones
Paula	Johnson

Minus/ Except Operator

- The MINUS operator allows you to filter out the results which are **present in the first query but absent in the second query.**
- After performing the MINUS operation, the data/records which are not present in the second SELECT statement or query are displayed.
- **Syntax:**

```
SELECT expression_1, expression_2, ... , expression_n FROM table_1
                              EXCEPT
SELECT expression_1, expression_2, ... , expression_n FROM table_2
```

Note:

- The MINUS operator is supported only in Oracle databases.
- For other databases like SQLite, PostgreSQL, SQL server, you can use **EXCEPT** operator to perform similar operations.

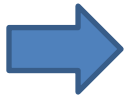
Minus Operator

Customer

first_name	last_name
Stephen	Jones
Mark	Smith
Denise	King
Paula	Johnson
Richard	Archer

Employee

first_name	last_name
Christina	Jones
Michael	McDonald
Paula	Johnson
Stephen	Jones
Richard	Smith



Our query would look like this:

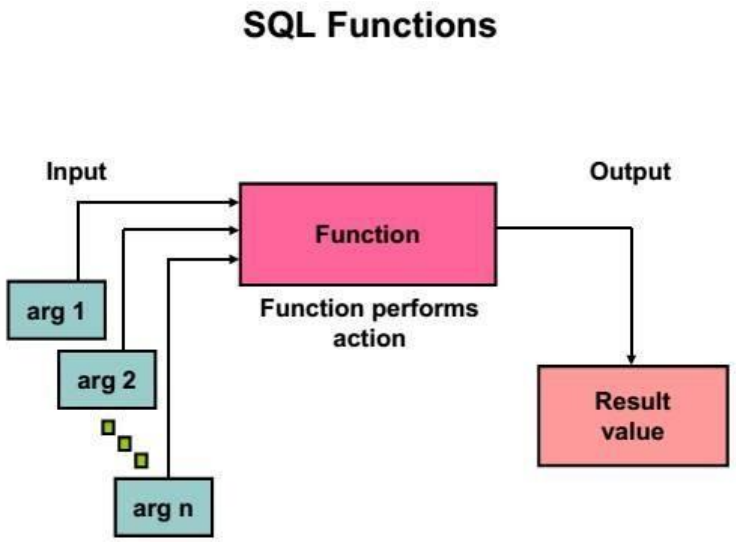
```
SELECT first_name, last_name
FROM customer
EXCEPT
SELECT first_name, last_name
FROM employee;
```

Result:

first_name	last_name
Mark	Smith
Denise	King
Richard	Archer

Built In Functions

- Functions accepts multiple input arguments and after processing it returns only one result value
- Types of Functions:
  - **MATH (NUMERIC)**
  - **STRING**
  - **DATE**



Built In Functions: MATH

- Functions applied to columns with data type number or numeric type

Function	Description
ABS	Returns the absolute value of a number.
AVG	Returns the average value of an expression/column values.
CEILING	Returns the nearest integer value which is larger than or equal to the specified decimal value.
COUNT	Returns the number of records in the SELECT query.
FLOOR	Returns the largest integer value that is less than or equal to a number. The return value is of the same data type as the input parameter.
MAX	Returns the maximum value in an expression.
MIN	Returns the minimum value in an expression.
RAND	Returns a random floating point value using an optional seed value.
ROUND	Returns a numeric expression rounded to a specified number of places right of the decimal point.
SIGN	Returns an indicator of the sign of the input integer expression.
SUM	Returns the sum of all the values or only the distinct values, in the expression. NULL values are ignored.

Built In Functions:
MATH

- **ABS()**: Return the absolute value of a number

```
SELECT Abs(-243.5) AS AbsNum;
```

Result:

Number of Records: 1

AbsNum
243.5

- **POWER(a,b):** The POWER() function returns the value of a number raised to the power of another number.

```
SELECT POWER(8, 3);
```

Result:

Number of Records: 1

512

Built In Functions:

MATH

- **ROUND()**: Return the round value to specified decimal.
  - Increasing value of previous digit by 1 if last digit is equal to or above 5
  - If last digit is below 5, don't change value of previous digit

Syntax: SELECT ROUND(45.926,2)

Output: 45.93

- **MOD(N,M) or N%M:** Function returns the remainder of N divided by number M

Syntax: SELECT MOD(4589,100)

Output: 89

Built In Functions:

MATH

- **CEIL():** It returns the smallest integer value that is greater than or equal to a number.

Syntax: SELECT CEIL(25.75)

Output: 26

- **FLOOR():** It returns the largest integer value that is less than or equal to a number.

Syntax: SELECT FLOOR(25.75)

Output: 25

Built In Functions:

MATH

- **GREATEST(exp1, exp2, exp3, exp_n):** It returns the greatest value in a list of expressions.

Syntax: SELECT GREATEST(30, 2, 36, 81, 125)

Output: 125

- **LEAST(exp1, exp2, exp3, exp_n):** It returns the smallest value in a list of expressions.

Syntax: SELECT LEAST(30, 2, 36, 81, 125)

Output: 2

- **SQRT():** It returns the square root of a number.

Syntax: SELECT SQRT(25)

Output: 5

Built In Functions: STRING

- MySQL string functions are used to manipulate string data and derive some information and analysis from tables
- These functions can be applied to column having data type CHAR, VARCHAR

String Function name		
ASCII()	CHARINDEX()	LEN()
CHAR()	LEFT()	LOWER()
NCHAR()	RIGHT()	RIGHT()
LOWER()	UPPER()	LTRIM()
RTRIM()	REPLACE()	REPLICATE()
REVERSE()	SPACE()	STUFF()
UNICODE()	QUOTENAME()	FORMAT()
CONCAT()		

Built In Functions: STRING

- **LOWER(String):** The LOWER() function converts a string to lower-case.

```
SELECT LOWER('SQL Tutorial is FUN!');
```

Result:

Number of Records: 1

sql tutorial is fun!

- **UPPER(String):** The UPPER() function converts a string to upper-case.

```
SELECT UPPER('SQL Tutorial is FUN!');
```

Result:

Number of Records: 1

SQL TUTORIAL IS FUN!

Built In Functions: STRING

- **LPAD(char1, n, char2):** This function is used to make the given string of the given size by adding the given symbol

Syntax: LPAD('geeks', 8, '0')

Output: 000geeks

- **LTRIM(string, chars):** This function is used to cut the given sub string from the original string

Syntax: LTRIM('123123geeks', '123');

Output: geeks

Built In Functions: STRING

- **RPAD(char1,n, cahr2):** This function is used to make the given string as long as the given size by adding the given symbol on the right

Syntax: RPAD('geeks', 8, '0');

Output: ‘geeks000’

- **RTRIM(string, char):** This function is used to cut the given sub string from the original string.

Syntax: RTRIM('geeksxyz', 'xyz');

Output: ‘geeks’

Built In Functions: STRING

- **TRIM():** This function is used to cut the given symbol from the string

Syntax: TRIM(LEADING '0' FROM '000123');

Output: 123

- **REPLACE(string, old_string, new_string) :** This function is used to cut the given string by removing the given sub string.

Syntax: SELECT REPLACE('SQL Tutorial', 'SQL', 'HTML');

Output: HTML Tutorial

Built In Functions: STRING

- **LENGTH(string):** This function is used to find the length of a word

Syntax: LENGTH('GeeksForGeeks');

Output: 13

- **SUBSTR(string,m,n):** This function is used to find a sub string from the a string from the given position

Syntax:SUBSTR('geeksforgeeks', 1, 5);

Output: ‘geeks’

Built In Functions:

DATE

- These functions can be applied to columns with data type Date and Time
- Date time functions can be applied in column having date and time data type

NOW(): This function will return today's date and time

- **Syntax:** SELECT NOW();
- **Output**: Today 2023-02-13 05:25:51

Function	Result
YEAR(NOW())	2023
HOUR(NOW())	5
MIN(NOW())	25
SEC(NOW())	51

Aggregate Functions

- For decision making, we need to summarize data from table like average, sum, minimum etc.
- SQL provides aggregate functions which can summarize data of given table by performing a calculation on set of values and return a single value
- Usually these functions ignore NULL values(except for COUNT)
- Different types of Aggregate Functions are:
  - Min(C): returns minimum value in column C
  - Max (C) : returns maximum value in column C
  - Sum(C) : sum of all values in column C
  - Avg(C) : average of all values in column C
  - Count(C) : number of values in column C

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

Aggregate Functions: Min( )

- **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 )
```

Example:

```
SELECT MIN(RATE)
FROM PRODUCT_MAST;
```

Output:

```
10
```

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

Aggregate Functions: Max( )

- **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 )
```

Example:

```
SELECT MAX(RATE)
FROM PRODUCT_MAST;
```

```
30
```

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

Aggregate Functions: Sum( )

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

Syntax

```
SUM()
or
SUM( [ALL|DISTINCT] expression )
```

Example: SUM()

```
SELECT SUM(COST)
FROM PRODUCT_MAST;
```

```
670
```

Aggregate Functions: Sum( )

Example: SUM() with WHERE

```
SELECT SUM(COST)
FROM PRODUCT_MAST
WHERE QTY>3;
```

Output:

```
320
```

Example: SUM() with GROUP BY

```
SELECT SUM(COST)
FROM PRODUCT_MAST
WHERE QTY>3
GROUP BY COMPANY;
```

Output:

```
Com1    150
Com2    170
```

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

Aggregate Functions: Avg( )

- 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

```
AVG()
or
AVG( [ALL|DISTINCT] expression )
```

Example:

```
SELECT AVG(COST)
FROM PRODUCT_MAST;
```

Output:

```
67.00
```

Sample

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

Aggregate Functions: Count( )

- 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.**

Example: COUNT()

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

Output:

```
10
```

Sample

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

Aggregate Functions: Count( )

Example: COUNT with WHERE

```
SELECT COUNT(*)
FROM PRODUCT_MAST;
WHERE RATE>=20;
```

Output:

```
7
```

Example: COUNT() with DISTINCT

```
SELECT COUNT(DISTINCT COMPANY)
FROM PRODUCT_MAST;
```

Output:

```
3
```

Sample

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

Aggregate Functions: Count( )

Example: COUNT() with GROUP BY

```
SELECT COMPANY, COUNT(*)
FROM PRODUCT_MAST
GROUP BY COMPANY;
```

Output:

```
Com1    5
Com2    3
Com3    2
```

Order By Clause

- The ORDER BY keyword is used to sort the result-set by a specified column.
- The ORDER BY keyword sorts the records in ascending order by default.
- If you want to sort the records in a descending order, you can use the DESC keyword.
- **Syntax**:

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

Order By Clause

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	4500.00
7	Muffy	24	Indore	10000.00

```
SQL> SELECT * FROM CUSTOMERS
   ORDER BY NAME DESC;
```

This would produce the following result −

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
7	Muffy	24	Indore	10000.00
6	Komal	22	MP	4500.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
5	Hardik	27	Bhopal	8500.00
4	Chaitali	25	Mumbai	6500.00

Order By Clause

```
SQL> SELECT * FROM CUSTOMERS
   ORDER BY NAME, SALARY;
```

This would produce the following result −

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
+----+----------+-----+-----------+----------+
```

Group By Clause

- Related rows can be grouped together by **GROUP BY clause** based on distinct values that exist for specified columns.
- A GROUP BY clause creates a set of data, containing several sets of records grouped together based on some condition.
- The GROUP BY statement is used with the SQL aggregate functions to group the retrieved data by one more columns or expression.
- GROUP BY column have to be in the SELECT clause
- Rows with same values for grouping columns are placed in distinct groups. Each group is treated as single row in query result

Group By Clause

- This GROUP BY clause follows the WHERE clause in a SELECT statement and precedes the ORDER BY clause.
- **Syntax**:

```
SELECT column1, column2
FROM table_name
WHERE [ conditions ]
GROUP BY column1, column2
```

Group By Clause

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Ramesh	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	kaushik	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

```
SQL> SELECT NAME,
SUM(SALARY)
FROM CUSTOMERS
GROUP BY NAME;
```

NAME	SUM(SALARY)
Hardik	8500.00
kaushik	8500.00
Komal	4500.00
Muffy	10000.00
Ramesh	3500.00

Having Clause

- The **HAVING Clause** enables you to specify conditions that filter which group results appear in the results.
- The WHERE clause places conditions on the selected columns, whereas the HAVING clause places conditions on groups created by the GROUP BY clause.
- A query can contain both WHERE and HAVING clause
- **Syntax**:

```
SELECT
FROM
WHERE
GROUP BY
HAVING
```

Having Clause

Table: Sales

SaleID	Product	Quantity	Price
1	Laptop	2	50000
2	Mobile	5	20000
3	Laptop	3	50000
4	Tablet	4	15000
5	Mobile	6	20000
6	Laptop	1	50000

```
SELECT Product, SUM(Quantity) AS Total_Quantity
FROM Sales
GROUP BY Product
HAVING SUM(Quantity) > 5;
```

Output:

Product	Total_Quantity
Laptop	6
Mobile	11

Views in SQL.

- In SQL, a view is a **virtual table** containing the records of one or more tables based on SQL statement executed.
- Just like a real table, view contains rows and columns.
- You can add SQL functions, WHERE, and JOIN statements to a view and present the data as if the data were coming from one single table
- The changes made in the table get automatically reflected in original table and vice versa

Views in SQL

Purpose of View:

- View is very useful in maintaining the security of database
- Consider a base table employee having following data

Emp_id	Emp_name	Salary	Address
E1	Kunal	8000	Camp
E2	Jay	7000	Tilak Road
E3	Radha	9000	Somwar Peth
E4	Sagar	7800	Warje
E5	Supriya	6700	LS Road

Views in SQL

1. Now just consider we want to give this table to any user but **don't want to show him salaries** of all the employees

In that we can create view from this table which will contain only the part of base table which we wish to show to user

Emp_id	Emp_name	Address
E1	Kunal	Camp
E2	Jay	Tilak Road
E3	Radha	Somwar Peth
E4	Sagar	Warje
E5	Supriya	LS Road

Views in SQL

2. Also in multiuser system, it may be possible that more than one user may want to update the data of some table

   - Consider two users A and B want to update the employee table.

     In such case we can give views to both these users

   - These users will make changes in their respective views, and their respective changes are done in the base table automatically

1. Creating View

Student_Detail

STU_ID	NAME	ADDRESS
1	Stephan	Delhi
2	Kathrin	Noida
3	David	Ghaziabad
4	Alina	Gurugram

- A view can be created using the **CREATE VIEW** statement.
- We can create a view from a single table or multiple tables.

1. Creating

Syntax:

```
CREATE VIEW view_name AS
SELECT column1, column2.....
FROM table_name
WHERE condition;
```

Query:

```
CREATE VIEW DetailsView AS
SELECT NAME, ADDRESS
FROM Student_Details
WHERE STU_ID < 4;
```

```
SELECT * FROM DetailsView;
```

Output:

NAME	ADDRESS
Stephan	Delhi
Kathrin	Noida
David	Ghaziabad

1. Creating View

Creating View from multiple tables

Student_Detail

STU_ID	NAME	ADDRESS
1	Stephan	Delhi
2	Kathrin	Noida
3	David	Ghaziabad
4	Alina	Gurugram

Student_Marks

STU_ID	NAME	MARKS	AGE
1	Stephan	97	19
2	Kathrin	86	21
3	David	74	18
4	Alina	90	20
5	John	96	18

1. Creating

Query:

```
CREATE VIEW MarksView AS
SELECT Student_Detail.NAME, Student_Detail.ADDRESS, Student_Marks.MARKS
FROM Student_Detail, Student_Mark
WHERE Student_Detail.NAME = Student_Marks.NAME;
```

```
SELECT * FROM MarksView;
```

NAME	ADDRESS	MARKS
Stephan	Delhi	97
Kathrin	Noida	86
David	Ghaziabad	74
Alina	Gurugram	90

2. Updating View

- Update query is used to update the records of view
- Updation in view reflects the original table also means same changes will be made in the original table

```
UPDATE < view_name > SET<column1>=<value1>,<column2>=<value2>,...
WHERE <condition>;
```

AGENT_CODE	AGENT_NAME	WORKING_AREA	COMMISSION	PHONE_NO
A007	Ramasundar	Bangalore	0.15	077-25814763
A003	Alex	London	0.18	075-12458969
A008	Alford	New York	0.12	044-25874365
A011	Ravi Kumar	Bangalore	0.15	077-45625874
A010	Santakumar	Chennai	0.14	007-22388644
A012	Lucida	San Jose	0.12	044-52981425

2. Updating View

```
UPDATE agentview
SET commission=.13
WHERE working_area='London';
```

AGENT_CODE	AGENT_NAME	WORKING_AREA	COMMISSION	PHONE_NO
A007	Ramasundar	Bangalore	0.15	077-25814763
A003	Alex	London	0.13	075-12458969
A008	Alford	New York	0.12	044-25874365
A011	Ravi Kumar	Bangalore	0.15	077-45625874
A010	Santakumar	Chennai	0.14	007-22388644
A012	Lucida	San Jose	0.12	044-52981425

2. Updating View

- In case of view containing joins between multiple tables, only insertion and updation in the view is allowed, deletion is not allowed
- Data modification is **not allowed** in the view which is based on union queries
- Data modification is **not allowed** in the view where GROUPBY or DISTINCT statements are used
- In view, text and image columns **can’t be modified**

3. Dropping View

- Drop query is used to delete a view

Syntax

```
DROP VIEW view_name;
```

- **Example**:

```
DROP VIEW MarksView;
```

Types of Views

- There are **two** types of Views:

1. **Simple View**

- The views which are based on **only one table** called as Simple view.
- Allow to perform DML (Data Manipulation Language) operations with some restrictions.
- Query defining simple view **cannot have** any join or grouping condition.

Types of Views

2. Complex View

- The views which are based on **more than one table** called as complex view.
- Do not allow DML operations to be performed.
- Query defining complex view can have join or grouping condition.

Difference: Simple and Complex View

Simple View	Complex View
Contains only one single base table or is created from only one table.	Contains more than one base table or is created from more than one table.
We cannot use group functions like MAX(), COUNT(), etc.	We can use group functions.
INSERT, DELETE and UPDATE are directly possible on a simple view.	We cannot apply INSERT, DELETE and UPDATE on complex view directly.
Example: CREATE VIEW Employee AS SELECT Empid, Empname FROM Employee WHERE Empid = '030314';	**Example:** CREATE VIEW EmployeeByDepartment AS SELECT e.emp_id, d.dept_id, e.emp_name FROM Employee e, Department d WHERE e.dept_id=d.dept_id;

Views : Advantages

1. **Security**: restrict user from accessing all data
2. **Hides Complexity**: view can be generated from complex query so result can be stored instead of writing complicated query again and again
3. **Dynamic Nature:** view definition remains unaffected if any changes are made in base table except table drop of column is altered
4. **Does not allows direct access to tables of data dictionary:** data dictionary cannot be damaged or changed

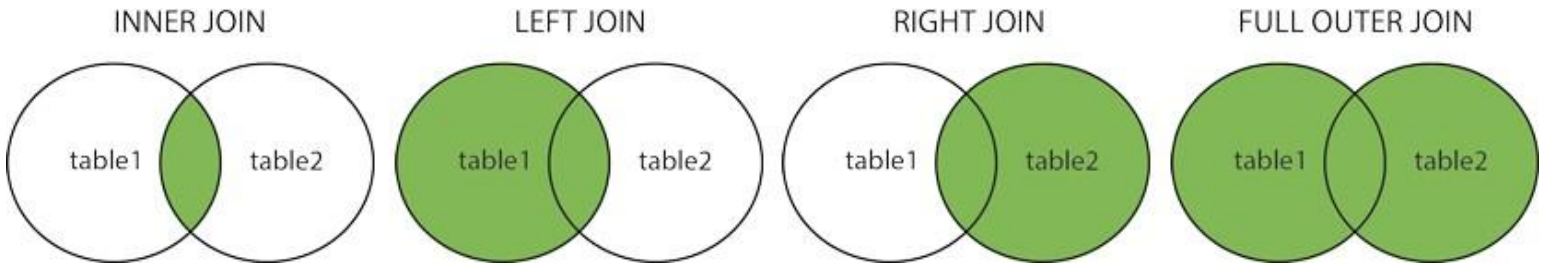
Views : Limitations

1. **Performance**: If view is defined by complex multi-table query user are not aware of how much complicated task query is performing
2. **View Management**: Views should be created as per standards so that job of DBA is simplified otherwise it becomes difficult to manage views
3. **Update Restrictions:** Whenever a user tries to update view, DBMS must translate query and apply updates on rows of underlying base table

Join

- A join operation means combining columns from one(self-table) or more tables by using values which are common
- There are six types of Joins:

1. INNER
2. LEFT OUTER
3. RIGHT OUTER
4. FULL OUTER
5. SELF

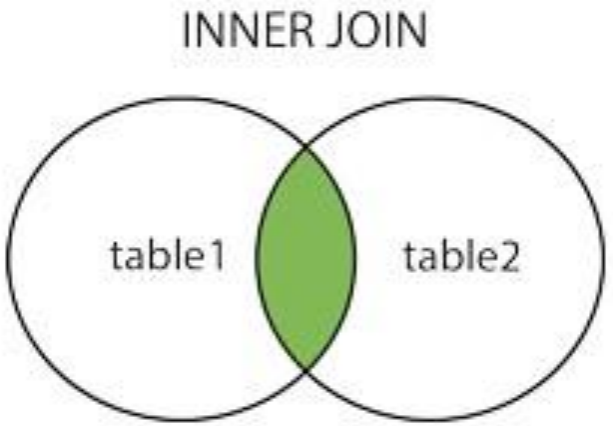


1. INNER JOIN

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

INNER JOIN Syntax

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



1. INNER JOIN

ROLL_NO	NAME	ADDRESS	PHONE	Age
1	HARSH	DELHI	XXXXXXXXXX	18
2	PRATIK	BIHAR	XXXXXXXXXX	19
3	RIYANKA	SILIGURI	XXXXXXXXXX	20
4	DEEP	RAMNAGAR	XXXXXXXXXX	18
5	SAPTARHI	KOLKATA	XXXXXXXXXX	19
6	DHANRAJ	BARABAJAR	XXXXXXXXXX	20
7	ROHIT	BALURGHAT	XXXXXXXXXX	18
8	NIRAJ	ALIPUR	XXXXXXXXXX	19

Student

COURSE_ID	ROLL_NO
1	1
2	2
2	3
3	4
1	5
4	9
5	10
4	11

StudentCourse

1. INNER JOIN

```
SELECT StudentCourse.COURSE_ID, Student.NAME, Student.AGE FROM Student
INNER JOIN StudentCourse
ON Student.ROLL_NO = StudentCourse.ROLL_NO;
```

Output

COURSE_ID	NAME	Age
1	HARSH	18
2	PRATIK	19
2	RIYANKA	20
3	DEEP	18
1	SAPTARHI	19

Output

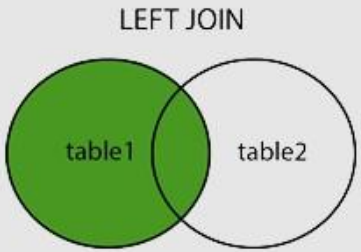
2. LEFT JOIN

- The LEFT JOIN keyword returns all records from the left table (table1), and the matching records from the right table (table2).

LEFT JOIN Syntax

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

Note: In some databases LEFT JOIN is called LEFT OUTER JOIN.



2. LEFT JOIN

ROLL_NO	NAME	ADDRESS	PHONE	Age
1	HARSH	DELHI	XXXXXXXXXX	18
2	PRATIK	BIHAR	XXXXXXXXXX	19
3	RIYANKA	SILIGURI	XXXXXXXXXX	20
4	DEEP	RAMNAGAR	XXXXXXXXXX	18
5	SAPTARHI	KOLKATA	XXXXXXXXXX	19
6	DHANRAJ	BARABAJAR	XXXXXXXXXX	20
7	ROHIT	BALURGHAT	XXXXXXXXXX	18
8	NIRAJ	ALIPUR	XXXXXXXXXX	19

Student

COURSE_ID	ROLL_NO
1	1
2	2
2	3
3	4
1	5
4	9
5	10
4	11

StudentCourse

2. LEFT JOIN

```
SELECT Student.NAME,StudentCourse.COURSE_ID
FROM Student
LEFT JOIN StudentCourse
ON StudentCourse.ROLL_NO = Student.ROLL_NO;
```

NAME	COURSE_ID
HARSH	1
PRATIK	2
RIYANKA	2
DEEP	3
SAPTARHI	1
DHANRAJ	*NULL*
ROHIT	*NULL*
NIRAJ	*NULL*

Output

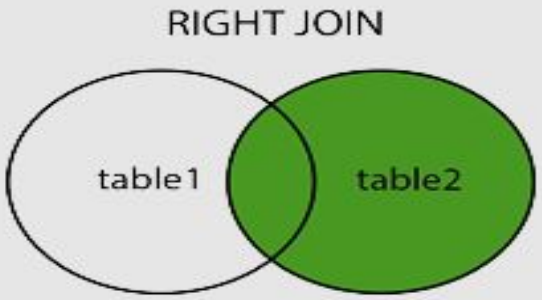
3. RIGHT JOIN

- The RIGHT JOIN keyword returns all records from the right table (table2), and the matching records from the left table (table1).

RIGHT JOIN Syntax

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

Note: In some databases `RIGHT JOIN` is called `RIGHT OUTER JOIN`.



3. RIGHT JOIN

ROLL_NO	NAME	ADDRESS	PHONE	Age
1	HARSH	DELHI	XXXXXXXXXX	18
2	PRATIK	BIHAR	XXXXXXXXXX	19
3	RIYANKA	SILIGURI	XXXXXXXXXX	20
4	DEEP	RAMNAGAR	XXXXXXXXXX	18
5	SAPTARHI	KOLKATA	XXXXXXXXXX	19
6	DHANRAJ	BARABAJAR	XXXXXXXXXX	20
7	ROHIT	BALURGHAT	XXXXXXXXXX	18
8	NIRAJ	ALIPUR	XXXXXXXXXX	19

Student

COURSE_ID	ROLL_NO
1	1
2	2
2	3
3	4
1	5
4	9
5	10
4	11

StudentCourse

3. RIGHT JOIN

```
SELECT Student.NAME,StudentCourse.COURSE_ID
FROM Student
RIGHT JOIN StudentCourse
ON StudentCourse.ROLL_NO = Student.ROLL_NO;
```

NAME	COURSE_ID
HARSH	1
PRATIK	2
RIYANKA	2
DEEP	3
SAPTARHI	1
NULL	4
NULL	5
NULL	4

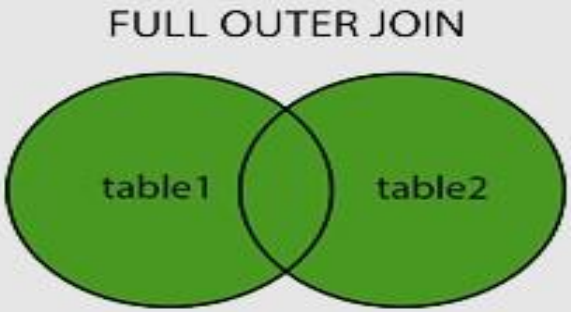
Output

4. FULL OUTER JOIN

- The FULL OUTER JOIN keyword returns all records when there is a match in left (table1) or right (table2) table records.
- **Tip:** FULL OUTER JOIN and FULL JOIN are the same.

FULL OUTER JOIN Syntax

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



4. FULL OUTER JOIN

ROLL_NO	NAME	ADDRESS	PHONE	Age
1	HARSH	DELHI	XXXXXXXXXX	18
2	PRATIK	BIHAR	XXXXXXXXXX	19
3	RIYANKA	SILIGURI	XXXXXXXXXX	20
4	DEEP	RAMNAGAR	XXXXXXXXXX	18
5	SAPTARHI	KOLKATA	XXXXXXXXXX	19
6	DHANRAJ	BARABAJAR	XXXXXXXXXX	20
7	ROHIT	BALURGHAT	XXXXXXXXXX	18
8	NIRAJ	ALIPUR	XXXXXXXXXX	19

Student

COURSE_ID	ROLL_NO
1	1
2	2
2	3
3	4
1	5
4	9
5	10
4	11

StudentCourse

4. FULL OUTER JOIN

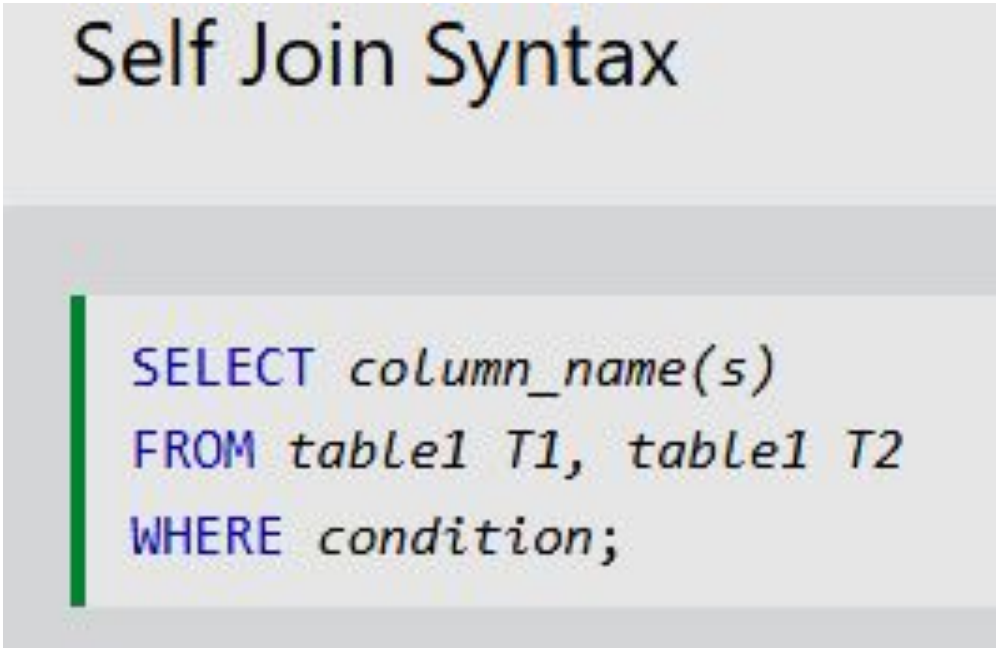
```
SELECT Student.NAME,StudentCourse.COURSE_ID
FROM Student
FULL JOIN StudentCourse
ON StudentCourse.ROLL_NO = Student.ROLL_NO;
```

Output

NAME	COURSE_ID
HARSH	1
PRATIK	2
RIYANKA	2
DEEP	3
SAPTARHI	1
DHANRAJ	NULL
ROHIT	NULL
NIRAJ	NULL
NULL	4
NULL	5
NULL	4

5. Self Join

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



5. Self Join

Id	Name	Age	City	Salary
1	Rohan	30	Mumbai	50000
2	Mohan	35	Pune	30000
3	Kriti	40	Mohali	75000
4	Riran	25	Patana	35000
5	Meenal	40	Mumbai	750000
5	Meenal	40	Mumbai	750000

Project_No	Id	Department
101	1	Testing
102	2	Development
103	3	Designing
104	4	Development

```
SELECT e.employee_name AS employee,
m.employee_name AS manager
FROM GFGemployees AS e
JOIN GFGemployees AS m
ON e.manager_id = m.employee_id;
```

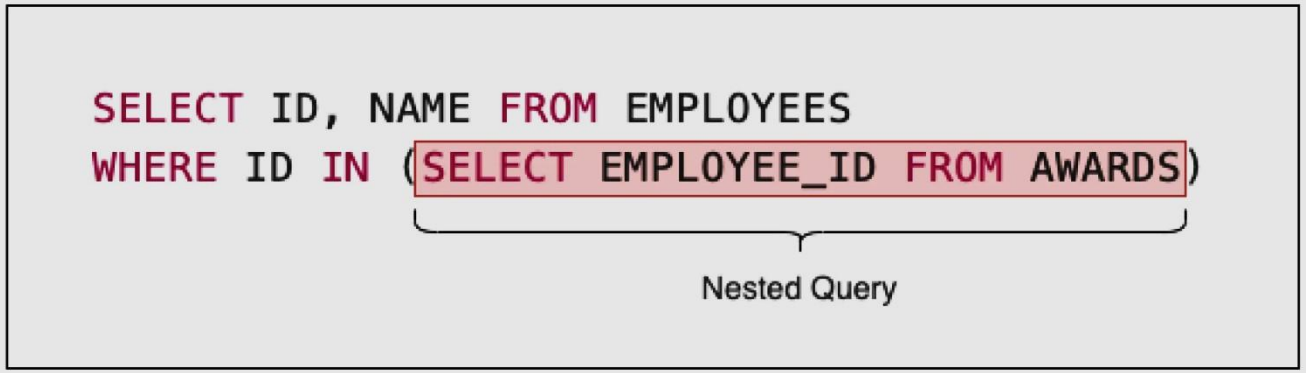
employee	manager
Zaid	Raman
Rahul	Raman
Raman	Kamran
Farhan	Kamran

Nested and Complex 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 SELECT statement nested within another SELECT statement is called a **subquery.**

Nested Query

- A nested query in SQL contains a query inside another query.
- The result of the inner query will be used by the outer query.
- For instance, a nested query can have two **SELECT** statements, one on the inner query and the other on the



Types of Nested Queries

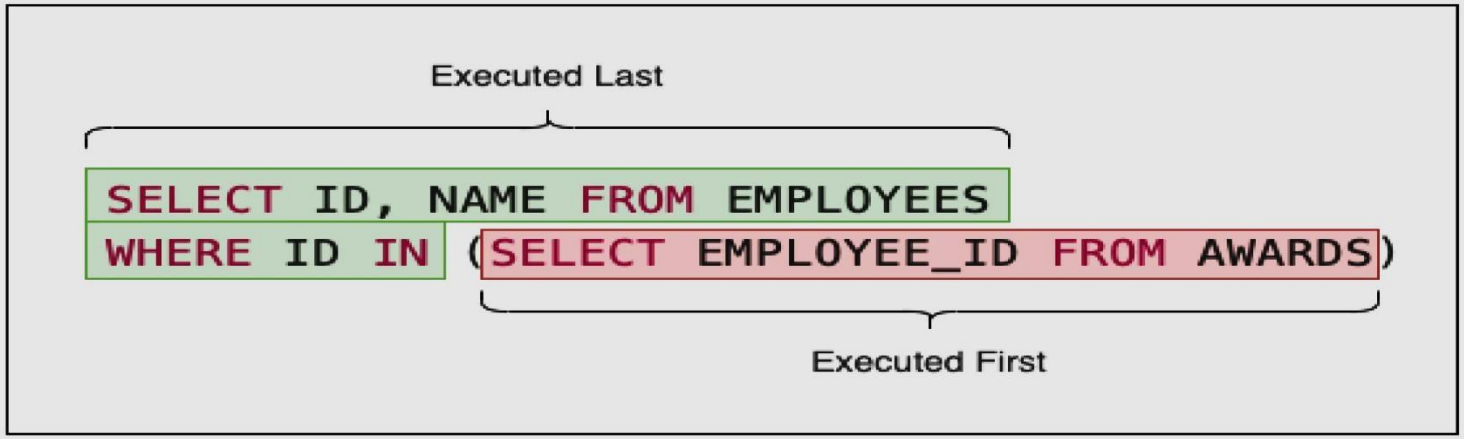
- Nested queries in SQL can be classified into two different types:
  1. Independent Nested Queries
  2. Co-related Nested Queries

1. Independent Nested Queries

- In independent nested queries, the execution order is from the innermost query to the outer query.
- The result of the inner query is used by the outer 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.

1. Independent Nested Queries

- 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**.



2. Co-related Nested Queries

- In co-related nested queries, the inner query **uses** the values from the outer query so that the inner query is executed 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.

How to Write Nested Query in SQL?

- We can write a nested query in SQL by nesting a **SELECT** statement within another **SELECT** statement.
- The outer **SELECT** statement uses the result of the inner **SELECT** statement for processing.
- The general syntax of nested queries will be:

```
SELECT column_name [, column_name ]
FROM table1 [, table2 ]
WHERE column_name OPERATOR
(
    SELECT column_name [, column_name ]
    FROM table1 [, table2 ]
    [WHERE]
)
```

Examples of Nested Query in SQL

```
CREATE TABLE employee (
    id NUMBER PRIMARY KEY,
    name VARCHAR2(100) NOT NULL,
    salary NUMBER NOT NULL,
    role VARCHAR2(100) NOT NULL
);
```

```
CREATE TABLE awards(
    id NUMBER PRIMARY KEY,
    employee_id NUMBER NOT NULL,
    award_date DATE NOT NULL
);
```

Examples of Nested Query in SQL

EMPLOYEE

- INSERT INTO employees VALUES (1, 'Augustine Hammond', 10000, 'Developer');
- INSERT INTO employees VALUES (2, 'Perice Mundford', 10000, 'Manager');
- INSERT INTO employees VALUES (3, 'Cassy Delafoy', 30000, 'Developer');
- INSERT INTO employees VALUES (4, 'Garwood Saffen', 40000, 'Manager');
- INSERT INTO employees VALUES (5, 'Faydra Beaves', 50000, 'Developer');

AWARDS:

- INSERT INTO awards VALUES(1, 1, TO_DATE('2022-04-01', 'YYYY-MM-DD'));
- INSERT INTO awards VALUES(2, 3, TO_DATE('2022-05-01', 'YYYY-MM-DD'));

Examples of Nested Query in SQL

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

Examples of Nested Query in SQL

Awards

id	employee_id	award_date
1	1	2022-04-01
2	3	2022-05-01

1. 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
1	Augustine Hammond
3	Cassy Delafoy

1. Independent Nested Queries

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
2	Perice Mundford
4	Garwood Saffen
5	Faydra Beaves

Examples of Nested Query in SQL

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

1. Independent Nested Queries

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

Examples of Nested Query in SQL

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

1. Independent Nested Queries

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

2. Co-related Nested Queries

- Select all employees whose salary is above the average salary of employees in their role.

```
SELECT * FROM employees emp1
WHERE salary > (
    SELECT AVG(salary) FROM employees emp2
    WHERE emp1.role = emp2.role
);
```

Output

id	name	salary	role
4	Garwood Saffen	40000	Manager
5	Faydra Beaves	50000	Developer

Nested and Complex Queries: Summary

- A nested query in SQL contains a query inside another query, and the outer query will use the result of the inner query.
- We can classify nested queries into independent and co-related nested queries.
- In independent nested queries, the order of execution is from the innermost query to the outermost query
- In co-related nested queries, the inner query uses the values from the outer query so that the inner query is executed for every row processed by the outer query
- Co-related nested query runs slow when compared with independent nested query.

Predicates

- A Predicate in DBMS is a condition expression which evaluates and results in Boolean value either true or false which enables decision making in retrieving and manipulating a record.
- A predicate is a condition that is specified for:
  - Filtering the data using the ***WHERE*** clause,
  - Pattern matching in ***LIKE*** operator,
  - Specifying a set of list for using ***IN*** operator,
  - Manipulating a range of values using ***BETWEEN*** operator, etc.

Predicates: Example

Consider a sample table 'emp'

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7839	KING	PRESIDENT	–	17-NOV-81	5000	–	10
7698	BLAKE	MANAGER	7839	01-MAY-81	2850	500	30
7782	CLARK	MANAGER	7839	09-JUN-81	2450	500	10
7566	JONES	MANAGER	7839	02-APR-81	2975	–	20
7788	SCOTT	ANALYST	7566	19-APR-87	3000	–	20
7902	FORD	ANALYST	7566	03-DEC-81	3000	–	20
7369	SMITH	CLERK	7902	17-DEC-80	800	500	20

Predicates: Example

Filtering the data using the ***WHERE*** clause/ Comparison Predicate

The predicate in where clause

```
select * from emp
where [job='MANAGER'];
```

O/P

```
3 rows selected.
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7698	BLAKE	MANAGER	7839	01-MAY-81	2850	–	30
7782	CLARK	MANAGER	7839	09-JUN-81	2450	–	10
7566	JONES	MANAGER	7839	02-APR-81	2975	–	20

Predicates: Example

Consider a sample table 'emp'

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7839	KING	PRESIDENT	–	17-NOV-81	5000	–	10
7698	BLAKE	MANAGER	7839	01-MAY-81	2850	500	30
7782	CLARK	MANAGER	7839	09-JUN-81	2450	500	10
7566	JONES	MANAGER	7839	02-APR-81	2975	–	20
7788	SCOTT	ANALYST	7566	19-APR-87	3000	–	20
7902	FORD	ANALYST	7566	03-DEC-81	3000	–	20
7369	SMITH	CLERK	7902	17-DEC-80	800	500	20

Predicates: Example

Specifying a set of list for using ***IN***

Example

```
select empno,job,sal,hiredate
from emp
where [ename in('SCOTT','FORD','SMITH','JONES')];
```

O/P

```
4 rows selected
```

EMPNO	JOB	SAL	HIREDATE
7566	MANAGER	2975	02-APR-81
7788	ANALYST	3000	19-APR-87
7902	ANALYST	3000	03-DEC-81
7369	CLERK	800	17-DEC-80

Predicates: Example

Consider a sample table 'emp'

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7839	KING	PRESIDENT	–	17-NOV-81	5000	–	10
7698	BLAKE	MANAGER	7839	01-MAY-81	2850	500	30
7782	CLARK	MANAGER	7839	09-JUN-81	2450	500	10
7566	JONES	MANAGER	7839	02-APR-81	2975	–	20
7788	SCOTT	ANALYST	7566	19-APR-87	3000	–	20
7902	FORD	ANALYST	7566	03-DEC-81	3000	–	20
7369	SMITH	CLERK	7902	17-DEC-80	800	500	20

Predicates: Example

Predicate in **BETWEEN** CLAUSE

Example

```
select empno,job,sal,hiredate
from emp
where [sal between 800 and 2900];
```

O/P

```
3 rows selected
```

EMPNO	JOB	SAL	HIREDATE
7698	MANAGER	2850	01-MAY-81
7782	MANAGER	2450	09-JUN-81
7369	CLERK	800	17-DEC-80

Predicates: Example

Consider a sample table 'emp'

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7839	KING	PRESIDENT	–	17-NOV-81	5000	–	10
7698	BLAKE	MANAGER	7839	01-MAY-81	2850	500	30
7782	CLARK	MANAGER	7839	09-JUN-81	2450	500	10
7566	JONES	MANAGER	7839	02-APR-81	2975	–	20
7788	SCOTT	ANALYST	7566	19-APR-87	3000	–	20
7902	FORD	ANALYST	7566	03-DEC-81	3000	–	20
7369	SMITH	CLERK	7902	17-DEC-80	800	500	20

Predicates: Example

The predicate in **LIKE** clause

Example

```
select empno,ename,hiredate,sal,job
from emp
where [ename like 'S%'];
```

O/P

```
2 rows selected
```

EMPNO	ENAME	HIREDATE	SAL	JOB
7788	SCOTT	19-APR-87	3000	ANALYST
7369	SMITH	17-DEC-80	800	CLERK

Predicates: Example

Consider a sample table 'emp'

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7839	KING	PRESIDENT	–	17-NOV-81	5000	–	10
7698	BLAKE	MANAGER	7839	01-MAY-81	2850	500	30
7782	CLARK	MANAGER	7839	09-JUN-81	2450	500	10
7566	JONES	MANAGER	7839	02-APR-81	2975	–	20
7788	SCOTT	ANALYST	7566	19-APR-87	3000	–	20
7902	FORD	ANALYST	7566	03-DEC-81	3000	–	20
7369	SMITH	CLERK	7902	17-DEC-80	800	500	20

Predicates: Example

Predicate in **IS NULL** clause

```
select * from emp
where [comm is null]
```

O/P

```
4 rows selected
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7839	KING	PRESIDENT	–	17-NOV-81	5000	–	10
7566	JONES	MANAGER	7839	02-APR-81	2975	–	20
7788	SCOTT	ANALYST	7566	19-APR-87	3000	–	20
7902	FORD	ANALYST	7566	03-DEC-81	3000	–	20

Predicates: Example

Consider a sample table 'emp'

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7839	KING	PRESIDENT	–	17-NOV-81	5000	–	10
7698	BLAKE	MANAGER	7839	01-MAY-81	2850	500	30
7782	CLARK	MANAGER	7839	09-JUN-81	2450	500	10
7566	JONES	MANAGER	7839	02-APR-81	2975	–	20
7788	SCOTT	ANALYST	7566	19-APR-87	3000	–	20
7902	FORD	ANALYST	7566	03-DEC-81	3000	–	20
7369	SMITH	CLERK	7902	17-DEC-80	800	500	20

Predicates: Example

The predicate NOT clause

Example

```
select * from emp
where [sal NOT between 800 and 2900 ];
```

O/P

```
4 rows selected
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7839	KING	PRESIDENT	–	17-NOV-81	5000	–	10
7566	JONES	MANAGER	7839	02-APR-81	2975	–	20
7788	SCOTT	ANALYST	7566	19-APR-87	3000	–	20
7902	FORD	ANALYST	7566	03-DEC-81	3000	–	20

Arithmetic Operators

- To perform various arithmetic operations, we can use these operators
- A comparison operator is a mathematical symbol used to compare two values in mathematical expression
- The result of an arithmetic operators can be any numeric value
- The various comparison operators are enlisted as given in

Operator	
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Remainder

Arithmetic Operators

- **Find all Faculty not teaching 'DT’**

SELECT * FROM Faculty

WHERE Subject <> 'DT’;

- **Find all Faculty teaching more than 10 hours.**

SELECT * FROM Faculty WHERE Hours >10;

Comparison Operators

- To compare attribute value of tuple with arithmetic comparison
- A comparison operator is a mathematical symbol used to compare two values in mathematical expression
- Comparison operators in conditions will be used to evaluate expression. The result of a comparison can be TRUE, FALSE, or UNKNOWN.

Operators	
=	Equal To
<	Less Than
<=	Less Than Equal To
>	Greater Than
>=	Greater Than Equal To
<>	Not equal to

Comparison Operators

Examples

- **Find all Faculty not teaching 'DT’**

SELECT * FROM Faculty WHERE Subject <> ‘DT’;

- **Find all Faculty teaching more than 10 hours.**

SELECT * FROM Faculty WHERE Hours>10 ;

Logical Operators

- The Logical operators will accept expression and return result as true or false to combine one or more true or false values.

Operators	
AND	Logical AND compares between two Booleans expressions to returns true when both expressions are true otherwise false.
OR	Logical OR compares between two Booleans expressions to return true when one of the expression is true otherwise false.
NOT	Not takes a single Boolean expression and changes its value from false to true or from true to false

Logical Operators

Examples

- **Find all faculty not teaching 'DT' and taught hours more than 10**

SELECT * FROM Faculty

WHERE Subject <> 'DT' **AND** Hours > 10

- **Find all Faculty teaching more than 10 hours or taught hours more than 10**

SELECT * FROM Faculty

WHERE Subject <> 'DT' **OR** Hours > 10

Triggers

- A trigger is a **procedure** that is automatically invoked by the DBMS in response to specific alteration database or a table in database.
- Triggers are stored in database as a simple database object.
- A database that has a set of associated triggers is called an **active database.**
- A database trigger enables DBA (Database Administrators) to create additional relationships between separate databases.

Triggers

- **Need/Purpose of Triggers**
  - To generate data automatically
  - Validate input data
  - Replicate data to different files to achieve data consistency

Components of Trigger (E-C-A Model)

- **Event (E):** SQL statement that causes the trigger to fire (or activate). This event may be insert, update or delete operation database table.
- **Condition (C)** : A condition that must be satisfied for execution of trigger.
- **Action (A)** : This is code or statement that execute when triggering condition is satisfied and trigger is activated on database table.

Triggers

Trigger syntax

```
CREATE [OR REPLACE] TRIGGER <Trigger Name>
[<ENABLE | DISABLE>]
<BEFORE|AFTER>
<INSERT|UPDATE | DELETE>
ON <Table_ Name>
        [FOR EACH ROW]
DECLARE
        <Variable_ Definitions>
BEGIN
        <TriggerCode> ;
END;
```

Triggers

- **Trigger Parameters**

OR REPLACE	If trigger is already present then drop and recreate the trigger
< Trigger Name>	Name of trigger to be created.
BEFORE	Indicates that trigger is to be fired before the triggering event occurs
AFTER	Indicates that trigger is to be fired After the triggering event occurs
INSERT	Indicates that trigger is to be fired whenever insert statement adds a row to table
UPDATE	Indicates that trigger is to be fired whenever Update statement modifies a row in a table
DELETE	Indicates that trigger is to be fired whenever delete statement removes a row from table

Triggers

- **Trigger Parameters**

FOR EACH ROW	Trigger will be fired only once for each row.
WHEN	Contains condition that must be satisfied to execute trigger
<trigger code>	Code to be executed whenever triggering event occurs

Trigger Types

1. **Row level Triggers**

- A row level trigger is fired each time the table is affected by the triggering statement
- For example, if an UPDATE statement changes multiple rows in a table, a row trigger is fired once for each row affected by the UPDATE statement
- If a triggering statement do not affect any row then a row trigger will not run only.
- IF **FOR EACH ROW** clause is written that means trigger is row level trigger.

Trigger Types

2. **Statement level triggers**

- A statement level trigger is fired only once on behalf of the triggering statement, irrespective of the number of rows in the table that are affected by the triggering statement.
- This trigger executes once even if no rows are affected.
- For example, if a DELETE statement deletes several rows from a table, a statement-level DELETE trigger fired only one time

Trigger Classification

- The classification of trigger is based on various parameters:

1. **Classification based on the timing**
   a) BEFORE Trigger: This trigger is fired before the occurrence of specified event.
   b) AFTER Trigger: This trigger is fired after the occurrence of specified event.
2. **Classification based on the level**
   b) STATEMENT level Trigger : This trigger is fired for once for the specified event statement.
   c) ROW level Trigger : This trigger is fired for all the records which are affected in the specified event. (only for DML)

Trigger Classification

3. **Classification based on the Event**
   a) DML Trigger : This trigger fires when the DML event such as INSERT/UPDATE / DELETE is specified
   b) DDL Trigger: This trigger fires when the DDL event such as CREATE or ALTER is specified
   c) DATABASE Trigger: This trigger fires when the database event such as LOGON/ LOGOFF , STARTUP/SHUTDOWN) is specified.

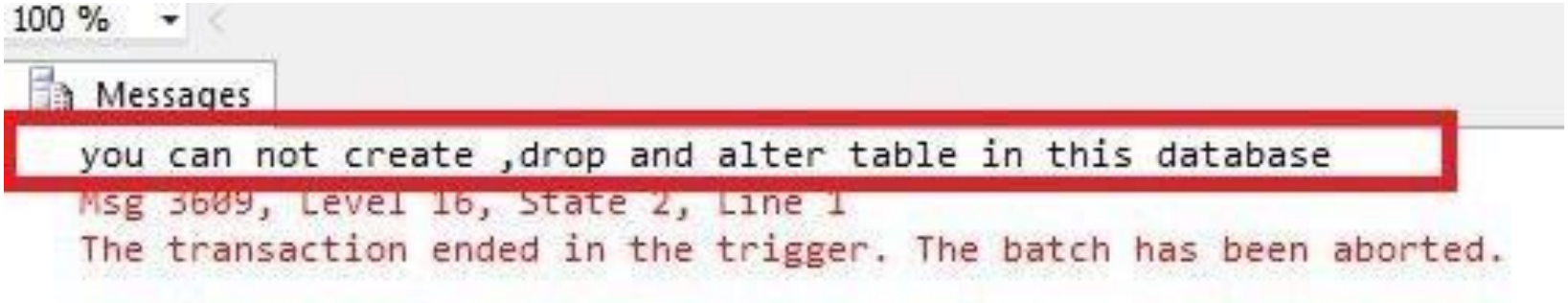
DDL Trigger

- The DDL triggers are fired in response to DDL (Data Definition Language) command events that start with Create, Alter, and Drop, such as Create_table, Create_view, drop_table, Drop_view, and Alter_table.

```
create trigger saftey
on database
for
create_table,alter_table,drop_table
as
print'you can not create ,drop and alter table in this database'
rollback;
```

DDL Trigger

- When we create, alter, or drop any table in a database, then the following message appears,



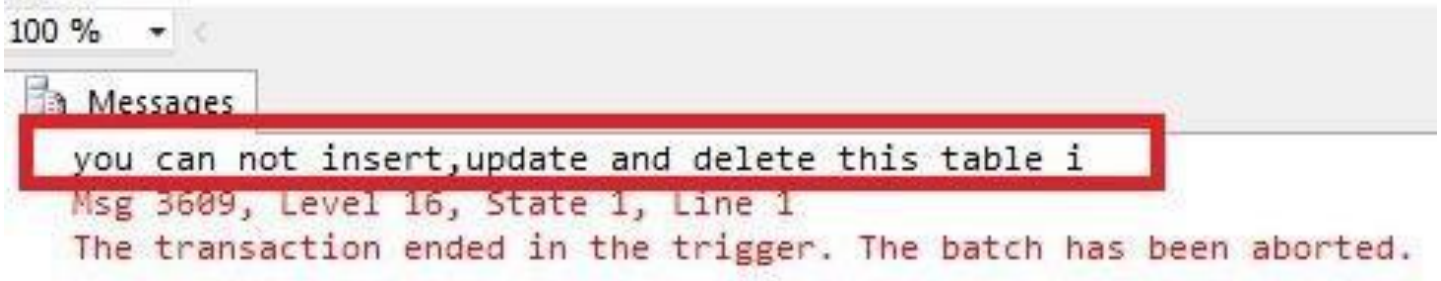
DML Trigger

- The DML triggers are fired in response to DML (Data Manipulation Language) command events that start with Insert, Update, and Delete.
- Like insert_table, Update_view and Delete_table.

```
create trigger deep
on emp
for
insert,update,delete
as
print'you can not insert,update and delete this table i'
rollback;
```

DML Trigger

- When we insert, update, or delete a table in a database, then the following message appears,



Trigger Example: Row Level Trigger

- Creating a trigger on employee table whenever a new employee added, a comment is written in EmpLog table

```
mysql> CREATE OR REPLACE TRIGGER AutoRecruit
AFTER INSERT ON EMP
FOR EACH ROW
BEGIN
        INSERT into EmpLog values(“Employee inserted”);
END;
```

Trigger created

Trigger Example

mysql>INSERT into EMP (1, ‘abc’, ‘manager’, 30000);

1 row created

mysql> SELECT * from EmpLog;

STATUS

Employee inserted

Trigger Example:2

```
CREATE TABLE Student(
studentID INT NOT NULL AUTO_INCREMENT,
FName VARCHAR(20),
LName VARCHAR(20),
Address VARCHAR(30),
City VARCHAR(15),
Marks INT,
PRIMARY KEY(studentID)
);
```

DESC Student;

StudentID	Fname	Lname	Address	City	Marks
INT	Varchar(20)	Varchar(20)	Varchar(30)	Varchar(15)	INT

Trigger Example:2

- ***Before Insert***

```
CREATE TRIGGER calculate
before INSERT
ON student
FOR EACH ROW
SET new.marks = new.marks+100;
```

- Here when we insert data into the student table automatically the trigger will be invoked.
- The trigger will add 100 to the marks column into the student column.

Trigger Example:2

After Insert

- To use this variant we need one more table i.e, Percentage where the trigger will store the results. Use the below code to create the Percentage Table.

```
create table Final_mark(
per int );
```

```
CREATE TRIGGER total_mark
after insert
ON student
FOR EACH ROW
insert into Final_mark values(new.marks);
```

Here when we insert data to the table, *total_mark trigger* will store the result in the Final_mark table.

Trigger Operations

- Data dictionary for triggers
- Dropping triggers
- Disabling Triggers

Trigger Operations

Data dictionary for triggers

- Once triggers are created their definitions can be viewed by selecting it from system tables as shown as follows:

SQL>Select * From User_Triggers

Where Trigger Name= '<Trigger_Name>' ;

- This statement will give you all properties of trigger including trigger code as well.

Dropping Triggers

- To remove trigger from database we use command DROP

SQL> Drop trigger <Trigger_Name>;

Trigger Operations

Disabling Triggers

- To deactivate trigger temporarily this can be activated again by enabling it.

MysQL> Alter trigger <Trigger_ Name> {disable | enable};

Trigger Advantages

- Triggers are useful for enforcing referential integrity, which preserves the defined relationships between tables when you add, update or delete the rows in those tables.
- Make sure that a column is filled with default information.
- After finding that the new information is inconsistent with the database, raise an error that will cause the entire transaction to roll back.

Trigger Disadvantages

- **Invisible from client applications –** Basically MySQL triggers are invoked and executed invisible from the client applications hence it is very much difficult to figure out what happens in the database layer.
- **Impose load on server –** Triggers can impose a high load on the database server.
- **Not recommended for high velocity of data –** Triggers are not beneficial for use with high-velocity data i.e. the data when a number of events per second are high. It is because in case of high-velocity data the triggers get triggered all the time.

Exercise: 1 Query

- **Employee**(Empid,Fname,Lname,Email,Phoneno,Hiredate,Jobid, Salary,Mid,Did)
- **Departments** (Did,Dname,Managerid,Locationid)
- **Locations** (Locationid,Streetadd,Postalcode,City)

- Write SQL Queries for following:

1. List employees having a manager who works for department based in U.S.
2. Display details of all employees in Finance department
3. Give 10% hike to all employees in Did 20
4. Display employee details whose salary is within range 1000 and 3000
5. Display all employee information whose first name starts with ‘R’ in descending order of their salary

Solution

1. **Mysql>**

```
SELECT * FROM Employee e
INNER JOIN Departments d ON e.did=d.did
INNER JOIN Locations l on l.locationid=d.locationid
WHERE city = 'U.S.';
```

2. **Mysql>**

```
SELECT * FROM Employee e
INNER JOIN Departments d ON e.did=d.did
WHERE Dname='Finance'
```

Solutio

3. **Mysql>**

UPDATE Employees
SET salary = 1.1* salary
WHERE did=20;

4. **Mysql>**

SELECT * FROM Employees
WHERE Salary BETWEEN 1000 AND 3000;

5. **Mysql>**

SELECT * FROM Employees
WHERE Fname LIKE ‘R%’
ORDERBY Salary DESC;

Exercise:2 Query

- **Employee**(eid,ename,address,city)
- **Works**(eid,cid,salary)
- **Company**(cid,cname,city)

- Write SQL Queries for:

1. Modify database so that John now lives in Mumbai
2. Find employees who live in same city as company for which they work
3. Give all employees of “AZ corporation” where there is increase in salary by 15%
4. Delete all tuples in works relation for employees of small bank corporation

Solution

1. **Mysql>**

   Update

   Employee SET

   city='Mumbai'

   WHERE ename='John';

2. **Mysql>**

   Select ename FROM Employee e, Company c, works w

   WHERE e.eid=w.eid

   AND w.cid=c.cid

   AND e.city=c.city;

Solution

3. **Mysql>**

```
UPDATE Works
SET Salary = 1.15*salary
WHERE eid IN(SELECT eid FROM Employee e, Works w, Company c
WHERE e.eid=w.eid AND w.cid=c.cid AND cname='AZ Corporation'));
```

4. **Mysql>**

```
DELETE FROM Employee
WHERE eid IN(SELECT eid FROM Works
WHERE cid IN(SELECT cid FROM Company WHERE cname='Small
bank corporation'));
```

Exercise:3

Query

- **For given database, write SQL queries:**

PERSON(driver_id#, name, address)

CAR(license, model, year)

ACCIDENT(report_no, date, location)

OWNS(driver_id#, license)

PARTICIPATED (driver_id, car, report_number, damage_amount)

1. Add new accident to database
2. Delete ‘Santro’ belonging to ‘John Smith’

Solution

1. Mysql>

```
INSERT INTO Accident(report_no, adate, location)
VALUES ('111','01/02/2023','PUNE');
```

2. Mysql>

```
DELETE FROM CAR
WHERE Model ='Santro'
AND
License IN(SELECT license FROM Owns
WHERE Driver_id IN
(SELECT driver_id FROM person
WHERE name= 'John Smith'));
```

Thank You!!