

SQL queries play a crucial role in database testing as they allow testers to interact with the database and verify its functionality, integrity, and performance. Here I will be showing some SQL Queries for testing different database testing purposes about retrieving the correct data and also to do it in different ways using different queries.

Database testing is the process of verifying the correctness, reliability, and performance of a database system. It involves testing various aspects of the database, such as data integrity, data consistency, data validation, and the overall functionality of the database.

Here are some key aspects of database testing:

- **Data Integrity:** Database testing ensures that the data stored in the database is accurate, complete, and consistent. It involves validating primary key constraints, foreign key relationships, unique constraints, and data validations defined in the database schema.
- **Data Manipulation:** Database testing involves testing the various operations performed on the database, such as inserting, updating, and deleting records. It ensures that these operations are executed correctly and produce the expected results.
- **Data Retrieval:** Testing the retrieval of data from the database is an important aspect of database testing. It involves verifying the accuracy and completeness of data fetched using queries and ensuring that the retrieved data matches the expected results.
- **Performance and Scalability:** Database testing includes evaluating the performance of the database under different loads and stress conditions. It involves measuring response times, throughput, and concurrency to ensure that the database can handle the expected workload efficiently. Scalability testing is performed to assess the performance of the database as the data volume and user load increase.
- **Security and Access Control:** Database testing verifies the security measures implemented in the database system. It includes testing access controls, user permissions, authentication mechanisms, and data encryption to ensure that sensitive data is protected and unauthorized access is prevented.
- **Data Migration and Integration:** Testing database migration and integration processes is crucial when transferring data from one database to another or integrating data from multiple sources. It involves validating data mappings, transformations, and ensuring the accuracy and integrity of data after migration or integration.
- **Error Handling and Recovery:** Database testing includes testing error handling mechanisms and recovery procedures. It involves simulating various error scenarios, such as network failures, system crashes, and validating the database's ability to recover data and maintain data integrity.
- **Backup and Recovery:** Database testing also involves testing backup and recovery procedures to ensure that data can be successfully backed up and restored in case of data loss or system failures. It includes testing backup schedules, recovery mechanisms, and data consistency after recovery.

To perform effective database testing, various techniques and tools are used, such as SQL queries, test data generation, database comparison tools, and automation frameworks. It is important to have a comprehensive test strategy and test cases that cover all aspects of the database system to ensure its reliability and performance.

When performing database testing, it's important to consider various scenarios, including boundary values, null values, error conditions, performance under different loads, and concurrency. By executing a variety of queries and analyzing the results, testers can assess the quality and reliability of the database system. Database testing involves verifying the accuracy, integrity, and performance of a database system. Queries are an essential part of database testing, as they are used to extract and manipulate data from the database. Here are some common types of queries used in database testing:

## **Example-1**

First, There are two tables Employee and Salary and with some values inside them:

A "CREATE TABLE" query is used to create a new table in a database. It defines the structure of the table by specifying the columns and their data types, constraints, and any additional properties. Here's the syntax and explanation of a typical "CREATE TABLE" query:

- **CREATE TABLE:** This statement indicates that a new table is being created.
- **table\_name:** The name of the table being created. Choose a meaningful name that reflects the purpose of the table.
- **column1, column2, ...:** The names of the columns in the table. Each column must have a unique name within the table.
- **datatype:** The data type of the column, which determines the type of values that can be stored in the column (e.g., VARCHAR, INTEGER, DATE, etc.).
- **constraint:** Optional constraints that define rules or conditions for the column. Constraints can include NULL/NOT NULL, PRIMARY KEY, UNIQUE, FOREIGN KEY, etc.

```
CREATE TABLE Employee (  
    EmpId int,  
    Name varchar(255),  
    ManagerId int,  
    DOJ DATETIME,  
    City varchar(255),  
    PRIMARY KEY (EmpId)  
);
```

The table needs to have some values in order for the testers to test the accuracy of the data. To insert values inside the tables we use the Insert Into query.

The "**INSERT INTO**" query is used to add new records or data into an existing table in a database. It allows you to specify the table name and provide values for the columns in the table.

Insert Queries:

- Insert into a Table: Add new records to a table.
- Insert with Data Validation: Verify that data constraints are enforced correctly.
- Insert with Identity/Auto-increment Columns: Test the generation of unique identifiers.

Here's the syntax and explanation of a typical "INSERT INTO" query:

- **INSERT INTO**: This statement indicates that you are inserting data into a table.
- **table\_name**: The name of the table where the data will be inserted.
- **(column1, column2, ...)**: The list of columns in the table for which values will be provided. This is optional, but it's good practice to explicitly specify the columns.
- **VALUES** (value1, value2, ...): The values to be inserted into the corresponding columns. The values must be in the same order as the columns specified.

**INSERT INTO Employee VALUES**

```
(121, 'John', 321, '2016/1/31', 'hyd'),  
(321, 'David', 986, '2018/1/30', 'Chennai'),  
(421, 'Scott', 876, '2020/11/27', 'Mumbai');
```

```
CREATE TABLE Salary (  
  EmpId int,  
  Project varchar(255),  
  Salary int,  
  Variable int,  
  PRIMARY KEY (EmpId)  
);
```

**INSERT INTO Salary VALUES**

```
(121, 'P1', 20000, 0),  
(321, 'P2', 35000, 1000),  
(421, 'P1', 50000, 3000);
```

And that is how the Employee and Salary tables look like:

NB: I am getting the values from both tables by using (**SELECT \* from Employee**) & (**SELECT \* from Salary**)

A "SELECT" query is used to retrieve data from a database table. It allows you to specify the columns you want to retrieve, apply filters to narrow down the results, perform calculations, and join multiple tables to combine data. Here's the syntax and explanation of a typical "SELECT" query:

- **SELECT**: This statement indicates that you are selecting data from a table.
- **column1, column2, ...**: The columns you want to retrieve from the table. You can specify multiple columns separated by commas or use the asterisk (\*) to retrieve all columns.
- **FROM table\_name**: The name of the table from which you want to retrieve the data.
- **WHERE** condition: Optional. It allows you to specify conditions to filter the rows returned. Only rows that satisfy the condition will be included in the result set.
- **GROUP BY** column(s): Optional. It is used to group the result set based on one or more columns. Typically, aggregate functions like **SUM**, **AVG**, **COUNT**, etc., are used in combination with **GROUP BY**.
- **HAVING** condition: Optional. It is used to filter the result set based on conditions after the grouping has been applied.
- **ORDER BY** column(s): Optional. It allows you to specify the order in which the result set should be sorted. You can sort by one or more columns in ascending (**ASC**) or descending (**DESC**) order.

\* Means to select all the values from certain table

```
1 SELECT * FROM Employee
```

!	Empld	Name	ManagerId	DOJ	City
121		John	321	2016-01-31 00:00:00	hyd
321		David	986	2018-01-30 00:00:00	Chennai
421		Scott	876	2020-11-27 00:00:00	Mumbai

```
1 SELECT * FROM Salary
```

EmpId	Project	Salary	Variable
121	P1	20000	0
321	P2	35000	1000
421	P1	50000	3000

There are some questions to work on these tables. They are as follows:

1. Write an SQL query to fetch the EmpID and Name of all the employees working under Manager with id - "986"  
**= SELECT empid, name, managerid FROM Employee  
where managerid = 986**

The **SELECT** statement is used to retrieve data from one or more tables. It allows us to specify the columns you want to retrieve and apply filtering conditions using the WHERE clause. In this example, we need to specify and retrieve the data of managerId '986'. This is why we are using the WHERE clause to specify the managerId.

```
1 SELECT empid, name, managerid FROM Employee
2 WHERE managerid = 986
```

empid	name	managerid
321	David	986

2. Write an SQL query to fetch the different projects available from the Salary table.  
**= SELECT DISTINCT(project) from Salary**

The "**DISTINCT**" keyword is used in a "**SELECT**" query to retrieve unique or distinct values from a specific column or set of columns in a database table. It eliminates

duplicate values and returns only unique entries. Here's the syntax and explanation of a typical "SELECT DISTINCT" query:

- **SELECT DISTINCT:** This statement indicates that you want to retrieve distinct or unique values from the specified columns.
- **column1, column2, ...:** The columns for which you want to retrieve distinct values. You can specify multiple columns separated by commas.
- **FROM table\_name:** The name of the table from which you want to retrieve the data.
- **WHERE condition:** Optional. It allows you to specify conditions to filter the rows before retrieving distinct values.

```
1 SELECT DISTINCT(project) FROM Salary
```

! project
P1
P2

3. Write an SQL query to fetch John's EmpId, Date of Joining, and the city he lives in from the Employee table.

= **SELECT name, empid, doj, city from Employee where name = 'John'**

To select a specific person's details from a database, you would need to know the criteria by which you want to identify that person, such as their unique identifier, name, or any other identifying attribute. Once you have identified the specific person, you can use a "SELECT" query to retrieve their details.

```
1 SELECT name, empid, doj, city FROM Employee
2 WHERE name = 'John'
```

! name	empid	doj	city
John	121	2016-01-31 00:00:00	hyd

4. Write an SQL query to fetch the count of employees working in Project 'P1'.

= **SELECT COUNT(\*) as DifferentProjects from Salary  
where project = 'P1'**

A "COUNT" query is used to retrieve the number of rows or records that match a specific condition in a database table. It allows you to count the occurrences of a

particular value, the total number of rows in a table, or the number of rows that satisfy a given condition. Here's the syntax and explanation of a typical "COUNT" query:

- **SELECT COUNT(column):** This statement indicates that you want to count the occurrences of values in a specific column or count the total number of rows.
- **column:** The column for which you want to count the occurrences of values. Use \* to count all rows in the table.
- **FROM table\_name:** The name of the table from which you want to retrieve the data.
- **WHERE condition:** Optional. It allows you to specify conditions to filter the rows before performing the count operation.

```
1 SELECT COUNT(*) AS DifferentProjects FROM Salary
2 WHERE project = 'P1'
```

! DifferentProjects

2

5. Write an SQL query to find the maximum, minimum, and average salary of the employees.

= **SELECT max(Salary) as HighestSalary, min(Salary) as LowestSalary, avg(Salary) as AvgSalary from Salary**

Maximum, minimum, and average queries are used to calculate specific statistical measures on numerical data in a database table. These queries help find the highest (maximum) value, lowest (minimum) value, and the average value of a column's data. Here's an explanation of each type of query:

- Maximum Query:

- **MAX(column):** This statement retrieves the maximum value from a specific column.
- **column:** The column for which you want to find the maximum value.
- **FROM table\_name:** The name of the table from which you want to retrieve the data.

Example: Finding the maximum salary from an "employees" table:

- Minimum Query:

- **MIN(column):** This statement retrieves the minimum value from a specific column.
- **column:** The column for which you want to find the minimum value.
- **FROM table\_name:** The name of the table from which you want to retrieve the data.

Example: Finding the minimum age from a "employees" table:

-Average Query:

- **AVG(column)**: This statement calculates the average (mean) value from a specific column.
- **column**: The column for which you want to find the average value.
- **FROM table\_name**: The name of the table from which you want to retrieve the data.

Example: Finding the average sales amount from an "employees" table:

```
1 SELECT max(Salary) AS HighestSalary, min(Salary) AS LowestSalary, avg(Salary) AS AvgSalary
2 FROM Salary|
```

HighestSalary	LowestSalary	AvgSalary
50000	20000	35000

6. Write an SQL query to find the employees id whose salary lies in the range of 30000 and 40000.

= **SELECT empid, Salary from Salary**  
**where Salary BETWEEN 30000 and 40000**

The "**BETWEEN**" operator is used in a SQL query to select values within a specified range. It allows you to retrieve records that fall between two values, inclusive of the specified range boundaries. Here's the syntax and explanation of a typical "BETWEEN" query:

- **SELECT column1, column2, ...**: This statement indicates the columns you want to retrieve from the table.
- **FROM table\_name**: The name of the table from which you want to retrieve the data.
- **WHERE column BETWEEN value1 AND value2**: This condition specifies the column you want to compare and the range of values you want to select. The range includes values that are equal to or between value1 and value2.



```

1 SELECT empid, Salary FROM Salary
2 WHERE Salary BETWEEN 30000 and 40000

```

empid	Salary
321	35000

7. Write an SQL query to fetch those employees who live in Chennai and work under the manager with ManagerId - 986.

= **SELECT \* from Employee**  
**where city = 'Chennai' and managerid = 986**

The "**WHERE**" clause in a SQL query is used to filter rows based on specified conditions. The "**AND**" operator is often used within the "**WHERE**" clause to combine multiple conditions and retrieve records that meet all of those conditions. Here's the syntax and explanation of a typical "**WHERE**" with "**AND**" query:

- **SELECT column1, column2, ...:** This statement indicates the columns you want to retrieve from the table.
- **FROM table\_name:** The name of the table from which you want to retrieve the data.
- **WHERE condition1 AND condition2 AND condition3 ...:** This condition specifies the criteria that each row must meet to be included in the result set. Multiple conditions can be combined using the "**AND**" operator, and all conditions must evaluate to true for a row to be selected.

```

1 SELECT * FROM Employee
2 WHERE city = 'Chennai' and managerid = 986

```

Empid	Name	Managerid	DOJ	City
321	David	986	2018-01-30 00:00:00	Chennai

8. Write an SQL query to fetch all the employees who either live in hyd or work under a manager with ManagerId - 321.

= **SELECT \* from Employee**  
**where city = 'hyd' or managerid = 321**

The "**WHERE**" clause in a SQL query is used to filter rows based on specified conditions. The "**OR**" operator is often used within the "**WHERE**" clause to combine

multiple conditions and retrieve records that meet at least one of those conditions. Here's the syntax and explanation of a typical "WHERE" with "OR" query:

- **SELECT column1, column2, ...:** This statement indicates the columns you want to retrieve from the table.
- **FROM table\_name:** The name of the table from which you want to retrieve the data.
- **WHERE condition1 OR condition2 OR condition3 ...:** This condition specifies the criteria that each row must meet to be included in the result set. Multiple conditions can be combined using the "OR" operator, and if any of the conditions evaluate to true, the row will be selected.

```
1 SELECT * FROM Employee
2 WHERE city = 'hyd' or managerid = 321
```

!	Empld	Name	ManagerId	DOJ	City
121		John	321	2016-01-31 00:00:00	hyd

9. Write an SQL query to fetch all those employees who work on Project other than P1.

= **SELECT \* from Salary**  
**where NOT project = 'P1'**

Or

**SELECT \* from Salary**  
**where project <> 'P1'**

The "WHERE" clause in a SQL query is used to filter rows based on specified conditions. The "NOT" operator can be used within the "WHERE" clause to negate a condition, retrieving records that do not meet the specified condition. Here's the syntax and explanation of a typical "WHERE NOT" query:

- **SELECT column1, column2, ...:** This statement indicates the columns you want to retrieve from the table.
- **FROM table\_name:** The name of the table from which you want to retrieve the data.
- **WHERE NOT condition:** This condition specifies the criteria that each row must not meet to be included in the result set. The "NOT" operator negates the condition, selecting rows that do not satisfy it.

```
1 SELECT * FROM Salary
2 WHERE NOT project = 'P1'
```

!	Empld	Project	Salary	Variable
	321	P2	35000	1000

10. Write an SQL query to sort the names in alphabetical order but in a descending way from the employee table.

= **SELECT \* FROM Employee order by name desc**

The "**ORDER BY DESC**" clause in a SQL query is used to sort the result set in descending order based on one or more columns. The "**DESC**" keyword specifies the descending order. Here's the syntax and explanation of a typical "**ORDER BY DESC**" query:

- **SELECT column1, column2, ...:** This statement indicates the columns you want to retrieve from the table.
- **FROM table\_name:** The name of the table from which you want to retrieve the data.
- **ORDER BY column1 DESC, column2 DESC, ...:** This clause specifies the columns by which you want to sort the result set in descending order. You can include multiple columns separated by commas to define the order of sorting.

```
1 SELECT * FROM Employee ORDER BY name DESC
```

!	Empld	Name	ManagerId	DOJ	City
	421	Scott	876	2020-11-27 00:00:00	Mumbai
	121	John	321	2016-01-31 00:00:00	hyd
	321	David	986	2018-01-30 00:00:00	Chennai

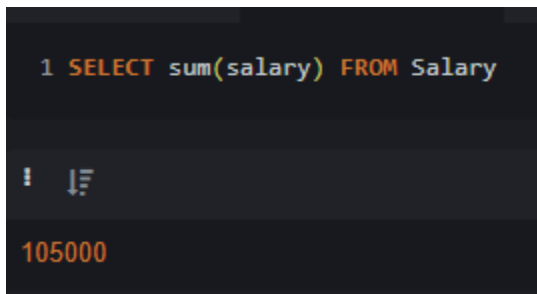
11. Write an SQL query to display the total salary of all employees.

= **SELECT sum(salary) from Salary**

A "**SUM**" query is used to calculate the sum of numerical values in a specific column or set of columns in a database table. It allows you to retrieve the total sum of

values, which can be useful for performing calculations or aggregating data. Here's the syntax and explanation of a typical "SUM" query:

- **SELECT SUM(column):** This statement indicates that you want to calculate the sum of values in a specific column.
- **column:** The column for which you want to calculate the sum.
- **FROM table\_name:** The name of the table from which you want to retrieve the data.
- **WHERE condition:** Optional. It allows you to specify conditions to filter the rows before performing the sum operation.

A screenshot of a SQL query being executed in a dark-themed environment. The query is '1 SELECT sum(salary) FROM Salary'. Below the query, there is a small icon of a document with a downward arrow, and the result '105000' is displayed in orange text.

```
1 SELECT sum(salary) FROM Salary
```

105000

12. Write an SQL query to show the salary of the employees who work in Mumbai.

= We can do it by nested queries and also joining. I will show Both.

First with joining,

```
SELECT Salary from Salary  
left join Employee on Salary.empid = Employee.empid  
where city = 'Mumbai'
```

A "LEFT JOIN" query is used to combine rows from two or more tables based on a related column, returning all the rows from the left table and the matching rows from the right table. This type of join ensures that all rows from the left table are included in the result set, even if there is no matching row in the right table. Here's the syntax and explanation of a typical "LEFT JOIN" query:

- **SELECT columns:** This statement indicates the columns you want to retrieve from the tables.
- **FROM left\_table:** The name of the left table from which you want to retrieve the data.
- **LEFT JOIN right\_table:** Specifies the right table to join with the left table.
- **ON left\_table.column = right\_table.column:** The condition that defines the relationship between the left and right tables based on a common column.

```
1 SELECT Salary FROM Salary
2 left join Employee ON Salary.empid = Employee.empid
3 WHERE city = 'Mumbai'|

! Salary

50000
```

Second by nested query,

**SELECT Salary from Salary**  
**where empid = (SELECT empid from Employee**  
**where city = 'Mumbai')**

A nested query, also known as a subquery, is a query that is embedded within another query. It allows you to use the result of one query as input or criteria for another query. The nested query is executed first, and its result is then used in the outer query. Here's an explanation of nested queries:

- The outer query selects specific columns from a table.
- The inner query, enclosed within parentheses, is executed first.
- The inner query selects a specific column from another table and applies a condition to filter the result.
- The result of the inner query, a set of values from the selected column, is then used as criteria in the outer query's "**WHERE**" clause, checking if a column value is present in the result set of the inner query.

```
1 SELECT Salary FROM Salary
2 left join Employee ON Salary.empid = Employee.empid
3 WHERE city = 'Mumbai'|

! Salary

50000
```

13. Write an SQL query to show who works under which project  
**= SELECT Employee.name,Salary.project from Employee**  
**right join Salary on Employee.empid = Salary.empid**

A "**RIGHT JOIN**" query is used to combine rows from two or more tables based on a related column, returning all the rows from the right table and the matching rows from the left table. This type of join ensures that all rows from the right table are included in

the result set, even if there is no matching row in the left table. Here's the syntax and explanation of a typical "**RIGHT JOIN**" query:

- **SELECT columns:** This statement indicates the columns you want to retrieve from the tables.
- **FROM left\_table:** The name of the left table from which you want to retrieve the data.
- **RIGHT JOIN right\_table:** Specifies the right table to join with the left table.
- **ON left\_table.column = right\_table.column:** The condition that defines the relationship between the left and right tables based on a common column.

```
1 SELECT Employee.name,Salary.project FROM Employee
2 right join Salary ON Employee.empid = Salary.empid
```

! name	project
John	P1
David	P2
Scott	P1

14. Write an SQL query to display the total salary of each employee adding the Salary with Variable value.

= **SELECT empid, salary+variable as Total from Salary**

To perform addition of two values and update a database value, you can use an "+" sign in SQL. The expression salary + variable calculates the new salary by adding the current value of the "salary" column with variable.

```
1 SELECT empid, salary+variable AS Total FROM Salary
```

! empid	Total
121	20000
321	36000
421	53000

15. Write an SQL query to fetch those employees whose name begins with any two characters, followed by a text 'vi' and ending with any sequence of characters.

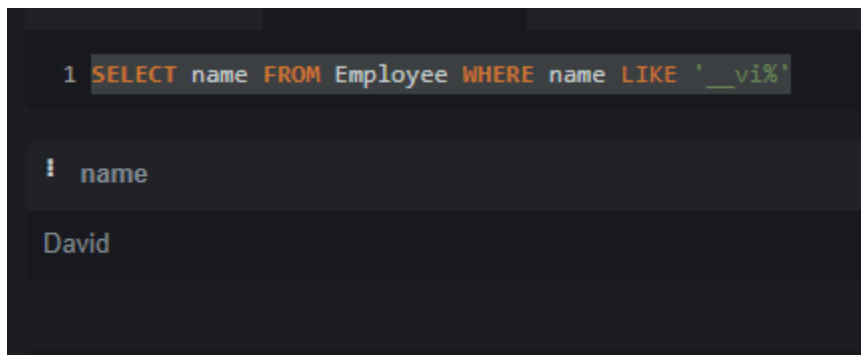
= **SELECT name from Employee WHERE name LIKE '\_\_vi%'**

The "**LIKE**" operator is used in SQL queries to perform pattern matching against a column's value in a database table. It allows you to search for records that match a specific pattern or contain a particular substring. The "**LIKE**" operator is often used with wildcard characters to represent unknown or variable parts of the pattern. Here's the syntax and explanation of a typical "**LIKE**" query:

- **SELECT columns:** This statement indicates the columns you want to retrieve from the table.
- **FROM table\_name:** The name of the table from which you want to retrieve the data.
- **WHERE column\_name LIKE pattern:** This condition specifies the column on which you want to perform pattern matching using the "**LIKE**" operator and the pattern to match against.

Here are the commonly used wildcard characters used in conjunction with the "**LIKE**" operator:

- % (percent sign): Represents any sequence of characters (including zero characters).
- \_ (underscore): Represents any single character.
- The condition name **LIKE 'J%'** filters the result set to include only records where the "name" column starts with the letter "J".



```
1 SELECT name FROM Employee WHERE name LIKE '__vi%'

! name

David
```

16. Write an SQL query to fetch those employees whose name has the letter 'o'

= **SELECT name from Employee where name LIKE '%o%'**

We can customize the pattern according to your specific needs. For example, to search for customers with names containing "o" anywhere within the name, you can use the following query. In this case, the % wildcard character is used both before and after the

search pattern "son" to indicate that the pattern can occur at any position within the name.  
%o%.

```
1 SELECT name FROM Employee WHERE name LIKE '%o%'

+-----+
| name |
+-----+
| John |
| Scott|
+-----+
```

## Example-2

To dig deep into Database Testing, I am creating 4 more Tables to show the queries in a more enhanced way.

```
CREATE TABLE EMPLOYEE (
    EMPLOYEE_ID int,
    LAST_NAME varchar(255),
    FIRST_NAME varchar(255),
    MIDDLE_NAME varchar(255),
    JOB_ID INT,
    MANAGER_ID INT,
    HIRE_DATE DATETIME,
    SALARY INT,
    COMM INT,
    DEPARTMENT_ID INT,
    PRIMARY KEY (EMPLOYEE_ID),
    FOREIGN KEY (JOB_ID) REFERENCES JOB(JOB_ID),
    FOREIGN KEY (DEPARTMENT_ID) REFERENCES
DEPARTMENT(DEPARTMENT_ID)
);
```

```
CREATE TABLE DEPARTMENT (
    DEPARTMENT_ID INT,
    NAME varchar(255),
    LOCATION_ID INT,
    PRIMARY KEY (DEPARTMENT_ID),
```



**FOREIGN KEY (LOCATION\_ID) REFERENCES LOCATION(LOCATION\_ID)**  
);

**CREATE TABLE LOCATION (**  
    **LOCATION\_ID** INT,  
    **REGIONAL\_GROUP** varchar(255),  
    **PRIMARY KEY (LOCATION\_ID)**  
);

**CREATE TABLE JOB (**  
    **JOB\_ID** INT,  
    **FUNCTION** varchar(255),  
    **PRIMARY KEY (JOB\_ID)**  
);

**INSERT INTO EMPLOYEE VALUES**  
    (7369, 'SMITH', 'JOHN', 'Q', 667, 7902, '17-Dec-84', 800, NULL, 20),  
    (7499, 'ALLEN', 'KEVIN', 'J', 670, 7698, '20-Feb-85', 1600, 300, 30),  
    (7505, 'DOYLE', 'JEAN', 'K', 671, 7839, '4-Apr-85', 2850, NULL, 30),  
    (7506, 'DENNIS', 'LYNN', 'S', 671, 7839, '15-May-85', 2750, NULL, 30),  
    (7507, 'BAKER', 'LESLIE', 'D', 671, 7839, '10-Jun-85', 2200, NULL, 40),  
    (7521, 'WARK', 'CYNTHIA', 'D', 670, 7698, '22-Feb-85', 1250, NULL, 40);

**INSERT INTO DEPARTMENT VALUES**  
    (10, 'ACCOUNTING', 122),  
    (20, 'RESEARCH', 124),  
    (30, 'SALES', 123),  
    (40, 'OPERATIONS', 167);

**INSERT INTO LOCATION VALUES**  
    (122, 'NEW YORK'),  
    (123, 'DALLAS'),  
    (124, 'CHICAGO'),  
    (167, 'BOSTON');

**INSERT INTO JOB VALUES**  
    (667, 'CLERK'),  
    (668, 'STAFF'),  
    (669, 'ANALYST'),  
    (670, 'SALESPERSION');

(671, 'MANAGER'),  
(672, 'PRESIDENT');

And that is how the Employee and Salary tables look like:

NB: I am getting the values from both tables by using (SELECT \* from Employee), (SELECT \* from Department), (SELECT \* from Job) & (SELECT \* from Location)

1 SELECT \* FROM EMPLOYEE

!	EM...	LAST...	FIRS...	MIDD...	JOB_ID	MAN...	HIRE...	SAL...	COMM	DEPARTME...
7369		SMITH	JOHN	Q	667	7902	1984-...	800	NULL	20
7499		ALLEN	KEVIN	J	670	7698	1985-...	1600	300	30
7505		DOYLE	JEAN	K	671	7839	1985-...	2850	NULL	30
7506		DENNIS	LYNN	S	671	7839	1985-...	2750	NULL	30
7507		BAKER	LESLIE	D	671	7839	1985-...	2200	NULL	40
7521		WARK	CYNT...	D	670	7698	1985-...	1250	NULL	40

1 SELECT \* FROM DEPARTMENT

!	DEPARTMENT_ID	NAME	LOCATION_ID
10		ACCOUNTING	122
20		RESEARCH	124
30		SALES	123
40		OPERATIONS	167

```
1 SELECT * FROM JOB
```

JOB_ID	FUNCTION
667	CLERK
668	STAFF
669	ANALYST
670	SALESPERSON
671	MANAGER
672	PRESIDENT

```
1 SELECT * FROM LOCATION
```

LOCATION_ID	REGIONAL_GROUP
122	NEW YORK
123	DALLAS
124	CHICAGO
167	BOSTON

I have prepared some questions to work on these tables. They are as follows:

1. List out the employees who are not receiving the commission.  
= **SELECT \* from EMPLOYEE where comm is Null**

The "IS NULL" operator is used in SQL queries to check whether a column's value is null or contains no value. It is typically used in the "WHERE" clause to filter records based on null values. Here's the syntax and explanation of a typical "IS NULL" query:

- **SELECT columns:** This statement indicates the columns you want to retrieve from the table.

- **FROM table\_name:** The name of the table from which you want to retrieve the data.
- **WHERE column\_name IS NULL:** This condition specifies the column you want to check for null values using the "IS NULL" operator.

```
1 SELECT * FROM EMPLOYEE WHERE comm is Null
```

!	EM...	LAST...	FIRS...	MIDD...	JOB_ID	MAN...	HIRE...	SAL...	COMM	DEPARTME...
7369		SMITH	JOHN	Q	667	7902	1984-...	800	NULL	20
7505		DOYLE	JEAN	K	671	7839	1985-...	2850	NULL	30
7506		DENNIS	LYNN	S	671	7839	1985-...	2750	NULL	30
7507		BAKER	LESLIE	D	671	7839	1985-...	2200	NULL	40
7521		WARK	CYNT...	D	670	7698	1985-...	1250	NULL	40

- List out the employees who are working in department 30 and draw the salaries of more than 2000

= **SELECT \* from EMPLOYEE where department\_id = 30 and Salary > 2000**

The "**WHERE**" clause in a SQL query is used to filter rows based on specified conditions. The "**AND**" operator is often used within the "**WHERE**" clause to combine multiple conditions and retrieve records that meet all of those conditions. Here's an example of a query that uses the "**WHERE**" clause with the "**AND**" operator:

- **SELECT column1, column2, ...:** This statement indicates the columns you want to retrieve from the table.
- **FROM table\_name:** The name of the table from which you want to retrieve the data.
- **WHERE condition1 AND condition2 AND condition3 ...:** This condition specifies the criteria that each row must meet to be included in the result set. Multiple conditions can be combined using the "**AND**" operator, and all conditions must evaluate to true for a row to be selected.

```
1 SELECT * FROM EMPLOYEE WHERE department_id = 30 and Salary > 2000
```

!	EM...	LAST...	FIRS...	MIDD...	JOB_ID	MAN...	HIRE...	SAL...	COMM	DEPARTME...
7505		DOYLE	JEAN	K	671	7839	1985-...	2850	NULL	30
7506		DENNIS	LYNN	S	671	7839	1985-...	2750	NULL	30

3. List out the employee id, name in descending order based on the salary column  
= **SELECT employee\_id, last\_name, salary from EMPLOYEE order by salary desc**

The "**ORDER BY DESC**" clause in a SQL query is used to sort the result set in descending order based on one or more columns. The "**DESC**" keyword specifies the descending order. Here's an example of a query that uses the "**ORDER BY DESC**" clause:

- **SELECT column1, column2, ...:** This statement indicates the columns you want to retrieve from the table.
- **FROM table\_name:** The name of the table from which you want to retrieve the data.
- **ORDER BY column1 DESC, column2 DESC, ...:** This clause specifies the columns by which you want to sort the result set in descending order. You can include multiple columns separated by commas to define the order of sorting.

```
1 SELECT employee_id, last_name, salary FROM EMPLOYEE ORDER BY salary DESC
```

employee_id	last_name	salary
7505	DOYLE	2850
7506	DENNIS	2750
7507	BAKER	2200
7499	ALLEN	1600
7521	WARK	1250
7369	SMITH	800

4. How many employees, who are working in different departments, are there in the organization.  
= **SELECT department\_id, COUNT(\*) from EMPLOYEE GROUP by department\_id**

A "**GROUP BY**" query in SQL is used to group rows based on one or more columns and perform aggregate functions on each group. It allows you to group and summarize data, providing insights into subsets of the data based on specific criteria. Here's the syntax and explanation of a typical "**GROUP BY**" query:

- **SELECT column1, column2, ...:** This statement indicates the columns you want to retrieve from the table. It can include both the columns to group by and the columns on which you want to perform aggregate functions.
- **FROM table\_name:** The name of the table from which you want to retrieve the data.
- **GROUP BY column1, column2, ...:** This clause specifies the columns by which you want to group the data.

```
1 SELECT department_id, COUNT(*) FROM EMPLOYEE GROUP BY department_id
```

department_id	
20	1
30	3
40	2

5. List out the department id having at least 3 employees  
 = **SELECT department\_id, COUNT(\*) from EMPLOYEE GROUP by department\_id HAVING COUNT(\*)>=3**

The "**GROUP BY HAVING**" clause in SQL is used to filter the result set of a grouped query based on specified conditions. It allows you to apply conditions to groups of data formed by the "**GROUP BY**" clause. The "**HAVING**" clause is similar to the "**WHERE**" clause, but it operates on the grouped data rather than individual rows. Here's the syntax and explanation of a typical "**GROUP BY HAVING**" query:

- **SELECT column1, column2, ...:** This statement indicates the columns you want to retrieve from the table. It can include both the columns to group by and the columns on which you want to perform aggregate functions.
- **FROM table\_name:** The name of the table from which you want to retrieve the data.
- **GROUP BY column1, column2, ...:** This clause specifies the columns by which you want to group the data.
- **HAVING condition:** This condition filters the grouped data based on specific criteria. It operates on the aggregated values calculated by the aggregate functions.

```
1 SELECT department_id, COUNT(*) FROM EMPLOYEE GROUP BY department_id HAVING COUNT(*)>=3
```

department_id	
30	3

6. Display the employees who got the maximum salary.

= **SELECT \* FROM EMPLOYEE where salary = (SELECT max(salary) from EMPLOYEE)**

In a nested query, the "**MAX**" function can be used to find the maximum value within a subquery. A nested query, also known as a subquery, is a query embedded within another query. Here's an example of how to use the "**MAX**" function in a nested query:

Let's break down the components of this query:

- The outer query retrieves the values from the specified column in the table.
- The inner query, enclosed within parentheses, calculates the maximum value from the specified column in another table using the "**MAX**" function.
- The outer query's "**WHERE**" clause compares the column values to the result of the subquery using the equality operator (=).

```
1 SELECT * FROM EMPLOYEE WHERE salary = (SELECT max(salary) FROM EMPLOYEE)
```

EM...	LAST...	FIRS...	MIDD...	JOB_ID	MAN...	HIRE...	SAL...	COMM	DEPARTME...
7505	DOYLE	JEAN	K	671	7839	1985-...	2850	NULL	30

7. Display the employees who are working in the Sales department.

= **SELECT \* from EMPLOYEE where department\_id IN (SELECT department\_id from DEPARTMENT where name = 'SALES')**

A nested query, also known as a subquery, can involve two different tables to retrieve data based on conditions or criteria from one table and use it in the query for the other table. Here's an example of a nested query with two different tables:

- The inner query (**SELECT department\_id from DEPARTMENT where name = 'SALES'**) selects the "**department\_id** " values from the "**DEPARTMENT** " table, filtering the Employees who work in **SALES**.

- The outer query “**SELECT \* from EMPLOYEE where department\_id** retrieves the columns "department\_id" the "EMPLOYEE" table.
- The **WHERE department\_id IN (SELECT department\_id from DEPARTMENT where name = 'SALES')** condition in the outer query filters the result set to include only the employees who work in the SALES department in the nested query.

```

1 SELECT * FROM EMPLOYEE WHERE department_id in (SELECT department_id FROM DEPARTMENT WHERE
2 name = 'SALES')

```

#	EM...	LAST...	FIRS...	MIDD...	JOB_ID	MAN...	HIRE...	SAL...	COMM	DEPARTME...
7499	ALLEN	KEVIN	J		670	7698	1985-...	1600	300	30
7505	DOYLE	JEAN	K		671	7839	1985-...	2850	NULL	30
7506	DENNIS	LYNN	S		671	7839	1985-...	2750	NULL	30

8. Display the employees who are working in “New York”  
 = We can do that with both nested queries like th previous one and also with joining the tables. I will show both.

**Nested Query:**

**SELECT \* from EMPLOYEE where department\_id = (SELECT department\_id FROM DEPARTMENT where location\_id = (SELECT location\_id from LOCATION where regional\_group = 'DALLAS'))**

```

1 SELECT * FROM EMPLOYEE WHERE department_id = (SELECT department_id FROM
2 DEPARTMENT WHERE location_id =
3 (SELECT location_id FROM LOCATION
4 WHERE regional_group = 'DALLAS'))

```

#	EM...	LAST...	FIRS...	MIDD...	JOB_ID	MAN...	HIRE...	SAL...	COMM	DEPARTME...
7499	ALLEN	KEVIN	J		670	7698	1985-...	1600	300	30
7505	DOYLE	JEAN	K		671	7839	1985-...	2850	NULL	30
7506	DENNIS	LYNN	S		671	7839	1985-...	2750	NULL	30

**Joining:**

**SELECT e.last\_name, d.department\_id, l.location\_id, l.regional\_group from EMPLOYEE e  
 RIGHT JOIN DEPARTMENT d on e.department\_id=d.department\_id**



**RIGHT JOIN LOCATION l on d.location\_id=l.location\_id  
WHERE l.regional\_group='DALLAS'**

```

1 SELECT e.last_name, d.department_id, l.location_id, l.regional_group FROM EMPLOYEE e
2 RIGHT JOIN DEPARTMENT d ON e.department_id=d.department_id
3 RIGHT JOIN LOCATION l ON d.location_id=l.location_id
4 WHERE l.regional_group='DALLAS'|

```

!	last_name	department_id	location_id	regional_group
	ALLEN	30	123	DALLAS
	DOYLE	30	123	DALLAS
	DENNIS	30	123	DALLAS

We can even do it in a merged way with both Joining and Nested Query.

**Merged Way:**

**SELECT e.last\_name, d.department\_id, d.location\_id from EMPLOYEE e  
RIGHT JOIN DEPARTMENT d on e.department\_id=d.department\_id  
WHERE d.location\_id = (SELECT location\_id from LOCATION WHERE  
regional\_group='DALLAS')**

```

1 SELECT e.last_name, d.department_id, d.location_id FROM EMPLOYEE e
2 RIGHT JOIN DEPARTMENT d ON e.department_id=d.department_id
3 WHERE d.location_id = (SELECT location_id FROM LOCATION WHERE regional_group='DALLAS')
4

```

!	last_name	department_id	location_id
	ALLEN	30	123
	DOYLE	30	123
	DENNIS	30	123

- Update the employees' salaries, who are working as Manager on the basis of 10%  
= Before updating the table looks like that:

**SELECT last\_name, salary, job\_id from EMPLOYEE  
where job\_id=(SELECT job\_id from JOB  
WHERE [function] = 'MANAGER')**

```

1 SELECT last_name, salary, job_id FROM EMPLOYEE
2 WHERE job_id=(SELECT job_id FROM JOB
3               WHERE [function] = 'MANAGER')
4

```

last_name	salary	job_id
DOYLE	2850	671
DENNIS	2750	671
BAKER	2200	671

After updating the salary value, the tables looks like that:

The "UPDATE" query in SQL is used to modify existing records in a database table. It allows you to update one or more columns of a table with new values based on specified conditions. Here's the syntax and explanation of a typical "UPDATE" query:

- **UPDATE table\_name:** This statement indicates that you want to update data in a specific table.
- **SET column1 = value1, column2 = value2, ...:** This clause specifies the columns you want to update and assigns new values to them.
- **WHERE condition:** This condition filters the rows that will be updated based on specific criteria. It determines which records will be affected by the update operation.

```

UPDATE EMPLOYEE set salary = salary + salary*0.1
where job_id=(SELECT job_id from JOB
               WHERE [function] = 'MANAGER')

```

```

SELECT last_name, salary, job_id from EMPLOYEE
where job_id=(SELECT job_id from JOB
               WHERE [function] = 'MANAGER')

```

```

1 SELECT last_name, salary, job_id FROM EMPLOYEE
2 WHERE job_id=(SELECT job_id FROM JOB
3               WHERE [function] = 'MANAGER')

```

last_name	salary	job_id
DOYLE	3135	671
DENNIS	3025	671
BAKER	2420	671

10. Delete the employees who are working in the accounting department

= Before deleting, the table looks like this:

Select employee\_id, last\_name, department\_id from EMPLOYEE

```

1 SELECT employee_id, last_name, department_id FROM EMPLOYEE

```

employee_id	last_name	department_id
7369	SMITH	20
7499	ALLEN	30
7505	DOYLE	30
7506	DENNIS	30
7507	BAKER	40
7521	WARK	40

After deleting, the table looks like this:

The "DELETE" query in SQL is used to remove one or more records from a database table. It allows you to delete specific rows based on specified conditions or delete all rows from a table. Here's the syntax and explanation of a typical "DELETE" query:

- **DELETE FROM table\_name:** This statement indicates that you want to delete data from a specific table.

- **WHERE** condition: This condition filters the rows that will be deleted based on specific criteria. It determines which records will be affected by the delete operation.

**DELETE from EMPLOYEE**

**where department\_id = (SELECT department\_id from DEPARTMENT  
where name = 'OPERATIONS')**

**Select employee\_id, last\_name, department\_id from EMPLOYEE**

```
1 SELECT employee_id, last_name, department_id FROM EMPLOYEE
```

employee_id	last_name	department_id
7369	SMITH	20
7499	ALLEN	30
7505	DOYLE	30
7506	DENNIS	30