

SQL queries are fundamental in database testing, enabling testers to interact with the database to verify its functionality, integrity, and performance. In this project, I have demonstrated various SQL queries for different database testing purposes, focusing on retrieving accurate data through multiple methods.

Database testing is the process of validating the correctness, reliability, and performance of a database system. This involves evaluating multiple facets of the database, such as data integrity, data consistency, data validation, and overall functionality.

Key aspects of Database Testing:

- **Data Integrity:** Database testing focuses on ensuring that the data within the database is accurate, complete, and consistent. This involves validating constraints such as primary keys, foreign key relationships, unique constraints, and any specific data validations outlined in the database schema. Testers must ensure that data adheres to these constraints during operations.
- **Data Manipulation:** Testing encompasses various database operations, including inserting, updating, and deleting records. It is essential to verify that these operations execute correctly and yield the expected results. Testers often run multiple queries to assess the impact of each operation on the database.
- **Data Retrieval:** A critical aspect of database testing is the retrieval of data. Testers must validate the accuracy and completeness of data returned by queries. This involves checking whether the retrieved results match the expected outputs and ensuring that all relevant data is accessible.
- **Performance and Scalability:** Evaluating the performance of a database under different conditions is crucial. Database testing assesses response times, throughput, and concurrent user handling to ensure the system can manage expected workloads efficiently. Scalability testing determines how the database performs as data volume and user loads increase, which is vital for long-term viability.
- **Security and Access Control:** Testing also involves verifying the security measures in place within the database system. This includes assessing access controls, user permissions, authentication mechanisms, and data encryption methods. Ensuring that sensitive data is protected and unauthorized access is prevented is paramount.
- **Data Migration and Integration:** When transferring data between databases or integrating data from multiple sources, thorough testing is essential. This involves validating data mappings and transformations to maintain accuracy and integrity post-migration or integration.
- **Error Handling and Recovery:** Database testing must include evaluating error handling mechanisms and recovery procedures. Simulating various error scenarios, such as network failures or system crashes, helps assess the database's ability to recover and maintain data integrity under adverse conditions.
- **Backup and Recovery:** Testing backup and recovery processes is crucial to ensure data can be restored in case of loss or system failures. This includes evaluating backup schedules, recovery mechanisms, and ensuring data consistency following recovery operations.

To perform effective database testing, a comprehensive test strategy and well-defined test cases are crucial. This strategy should cover all aspects of the database system, 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. Queries are an

essential part of database testing to extract and manipulate data from the database. Here are some common types of queries used in database testing:

Example-1

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

A **"CREATE TABLE"** query is utilized to establish a new table in a database, defining its structure by specifying columns, their data types, constraints, and additional properties. The query begins with the **"CREATE TABLE"** statement, indicating the creation of a new table, followed by the table's name, which should meaningfully reflect its purpose.

The columns within the table are then listed, each with a unique name, and assigned data types that determine the type of values they can store, such as **VARCHAR**, **INTEGER**, or **DATE**. Optional constraints can also be defined to set rules or conditions for the columns, including **NULL/NOT NULL**, **PRIMARY KEY**, **UNIQUE**, and **FOREIGN KEY**. These elements collectively ensure the integrity and functionality of the table within the database.

Create Employee Table:

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

Create Salary Table:

```
CREATE TABLE Salary (  
  EmpId int,  
  Project varchar(255),  
  Salary int,  
  Variable int,  
  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 using the Insert Into query. The **"INSERT INTO"** statement in SQL is used to add new records into a table within a database. It specifies the table name followed by a list of column names

(optional if inserting into all columns) and the corresponding values to be inserted into those columns. Each value provided in the statement must match the data type of the respective column it is being inserted into.

Insert data into the Employee Table:

```
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');
```

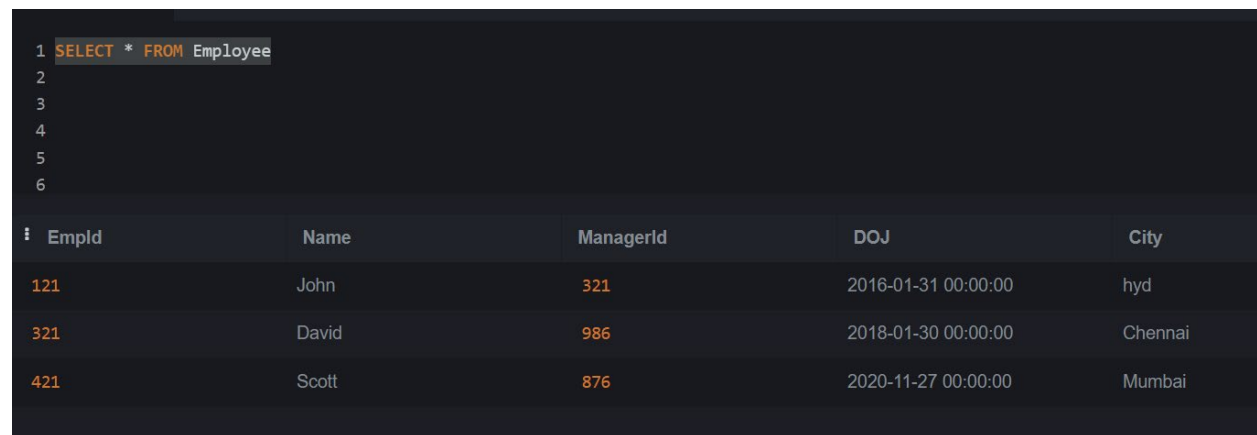
Insert data into the Salary Table:

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

I am getting the values from both tables by using (**SELECT * from Employee**), (**SELECT *from Salary**)

The "**SELECT**" query in SQL is used to retrieve data from a database. It allows users to specify which columns to retrieve and which rows should be returned based on specified conditions. The **SELECT** statement is fundamental for querying and retrieving data stored in database tables.

* Means to select all the values from certain table



The screenshot shows a SQL query editor with the command `SELECT * FROM Employee` at line 1. Below the editor, the results of the query are displayed in a table with 5 columns: EmpId, Name, ManagerId, DOJ, and City. The results show three rows of employee data.

EmpId	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			
2				
3				
4				
5				
6				
	EmpId	Project	Salary	Variable
	121	P1	20000	0
	321	P2	35000	1000
	421	P1	50000	3000

1. Write an SQL query to fetch the EmpID and Name of all the employees working under Manager with id- “876”

**SELECT empid, name, managerid from Employee
where managerid= 876;**

The query uses the **SELECT** statement to retrieve specific columns (**EmpID** and **Name**) from the **Employee** table. The **WHERE** clause filters the results based on the condition that the **ManagerID** must be equal to 876. This query is useful for retrieving a list of employees managed by a specific manager, facilitating organizational reporting and management tasks in a relational database system.

1	SELECT empid, name, managerid FROM Employee			
2	WHERE managerid= 876;			
3				
4				
5				
6				
	empid	name	managerid	
	421	Scott	876	

2. Write an SQL query to fetch the different salaries available from the Salary table.

SELECT DISTINCT(Salary) from Salary

The query uses the “**SELECT DISTINCT**” statement to retrieve unique salary values from the **Salary** table. Each unique salary value present in the table will be returned exactly once in the result set. This query is useful for obtaining a list of all unique salary levels stored in the database, which can be important for various analytical and reporting purposes within an organization.

```
1 SELECT DISTINCT(Salary) FROM Salary
2
3
4
5
6
```

Salary
20000
35000
50000

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

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

The query uses the **SELECT** statement to retrieve specific columns (**EmpId**, **DOJ**, and **City**) from the **Employee** table. The **WHERE** clause filters the results based on the condition that the **Name** must be equal to 'Scott'. This query is useful for retrieving essential information about a specific employee, identified by their name, to facilitate various operational and administrative tasks in a relational database system.

```
1 SELECT name, empid, doj,city FROM Employee
2 WHERE name='Scott'
3
4
5
6
```

name	empid	doj	city
Scott	421	2020-11-27 00:00:00	Mumbai

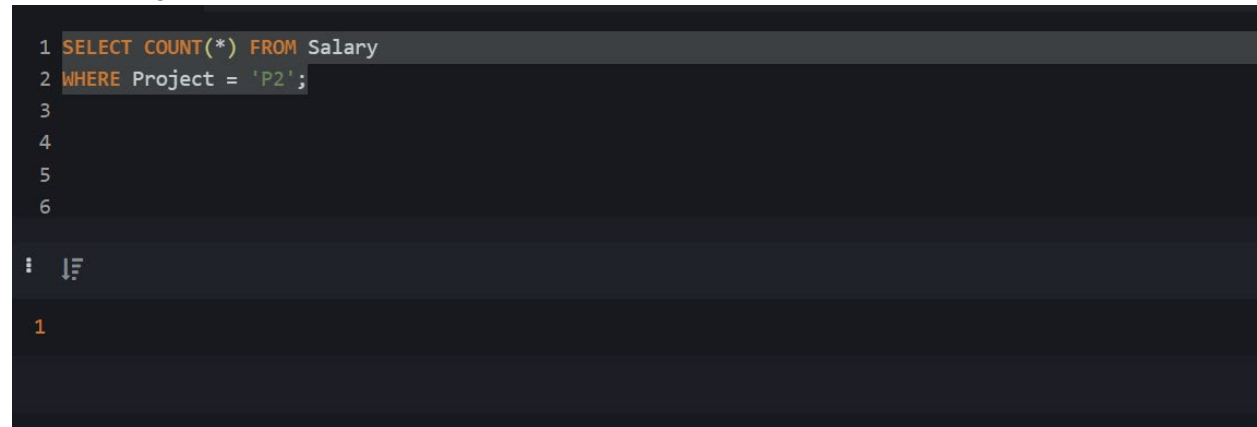
4. Write an SQL query to fetch the count of employees working in Project 'P2'.

**SELECT COUNT(*) FROM Salary
WHERE Project = 'P2';**

The query uses the **SELECT COUNT(*)** statement to retrieve the total number of employees working on **Project 'P2'** from the **Employee** table. The **WHERE** clause filters the results based on the condition that

the **Project** must be equal to 'P2'. This query is useful for determining the number of employees assigned to a specific project, which can be important for resource allocation, project management, and reporting within an organization.

```
1 SELECT COUNT(*) FROM Salary
2 WHERE Project = 'P2';
3
4
5
6
```



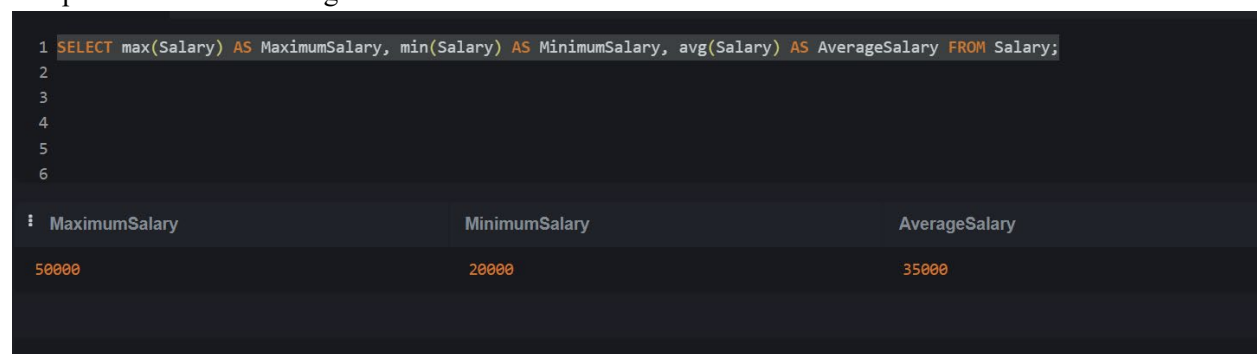
The image shows a SQL query editor with a dark background. The query is: `SELECT COUNT(*) FROM Salary WHERE Project = 'P2';`. Below the query, there is a result set icon (a small table with a downward arrow) and the number `1`, indicating that one row was returned.

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

```
SELECT max(Salary) AS MaximumSalary,
min(Salary) AS MinimumSalary,
avg(Salary) AS AverageSalary
FROM Salary;
```

The query uses the **SELECT** statement combined with aggregate functions (**MAX**, **MIN**, and **AVG**) to retrieve statistical information about the salaries in the **Salary** table. Specifically, it calculates the highest salary (**MAX**), the lowest salary (**MIN**), and the average salary (**AVG**). Each result is given a meaningful alias (**MaximumSalary**, **MinimumSalary**, **AverageSalary**) for clarity. This query is useful for summarizing salary data to understand salary distribution and inform decision-making regarding employee compensation within an organization.

```
1 SELECT max(Salary) AS MaximumSalary, min(Salary) AS MinimumSalary, avg(Salary) AS AverageSalary FROM Salary;
2
3
4
5
6
```



The image shows a SQL query editor with a dark background. The query is: `SELECT max(Salary) AS MaximumSalary, min(Salary) AS MinimumSalary, avg(Salary) AS AverageSalary FROM Salary;`. Below the query, there is a result set icon (a small table with a downward arrow) and a table with three columns: **MaximumSalary**, **MinimumSalary**, and **AverageSalary**. The values are 50000, 20000, and 35000 respectively.

MaximumSalary	MinimumSalary	AverageSalary
50000	20000	35000

6. Write an SQL query to find the employees id whose salary lies in the range of 30000 and 50000.

```
SELECT empid, salary from Salary
where Salary between 30000 and 50000;
```

The query uses the **SELECT** statement to retrieve the **EmpID** and **Salary** columns from the **Salary** table. The **WHERE** clause, with the **BETWEEN** operator, filters the results to include only those rows where the salary is within the specified range of **30,000 to 50,000**.

```
1 SELECT empid, salary FROM Salary
2 WHERE Salary between 30000 and 50000;
3
4
5
6
```

empid	salary
321	35000
421	50000

7. Write an SQL query to fetch those employees who live in Mumbai and work under the manager with ManagerId 876.

SELECT EmpId, City, ManagerId FROM Employee WHERE City='Mumbai' AND ManagerID='876';

The query uses the **SELECT** statement to retrieve the **EmpID**, **City**, and **ManagerID** columns from the **Employee** table. The **WHERE** clause includes two conditions combined with the **AND** operator. This query is useful for identifying employees based in Mumbai who report to a specific manager, facilitating localized management and organizational analysis.

```
1 SELECT EmpId, City, ManagerId FROM Employee WHERE City='Mumbai' AND ManagerID='876';
2
3
4
5
6
```

EmpId	City	ManagerId
421	Mumbai	876

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

SELECT * FROM Salary WHERE project <>'P1'

WHERE Project <> 'P1': The **<>** operator is used to select rows where the **Project** column is not equal to **'P1'**. This is a standard way to express inequality in SQL and is straightforward and commonly used.

SELECT * FROM Salary where NOT project = 'P1'

WHERE NOT Project = 'P1': The **NOT** operator negates the condition that follows it. Here, it means that rows where the **Project** column equals **'P1'** will be excluded, effectively selecting rows where the **Project** is not **'P1'**. This is another way to express inequality, although it is less commonly used compared to the **<>** operator.

```
1 SELECT * FROM Salary WHERE project <> 'P1'
2
3 SELECT * FROM Salary WHERE NOT project = 'P1'
4
5
6
```

EmpId	Project	Salary	Variable
321	P2	35000	1000

9. Write an SQL query to fetch the EmpID, Name, City, and Salary of employees by joining the Employee and Salary tables on the EmpID field.

SELECT Employee.empid, Employee.name, Employee.City, Salary.salary from Employee

Join Salary

On Employee.EmpId= Salary.EmpId

- **SELECT Employee.EmpID, Employee.Name, Employee.City, Salary.Salary:** Specifies the columns to be retrieved from the joined tables. This includes EmpID, Name, and City from the Employee table, and Salary from the Salary table.
- **FROM Employee:** Specifies the primary table from which to start the data retrieval.
- **JOIN Salary:** Indicates that the Salary table should be joined with the Employee table.
- **ON Employee.EmpID = Salary.EmpID:** Defines the join condition, which matches rows from the Employee table with rows from the Salary table where the EmpID values are equal.

```
1 SELECT Employee.empid, Employee.name, Employee.City, Salary.salary FROM Employee
2 Join Salary
3 ON Employee.EmpId= Salary.EmpId
4
5
6
```

empid	name	City	salary
121	John	hyd	20000
321	David	Chennai	35000
421	Scott	Mumbai	50000

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

SELECT sum(salary) as TotalSalary from Salary

- **SELECT SUM(Salary):** The **SUM** function calculates the total sum of the values in the **Salary** column.
- **AS TotalSalary:** This assigns an alias (**TotalSalary**) to the result, making it more readable and understandable in the output.
- **FROM Salary:** Specifies the table from which the data should be retrieved, which in this case is the **Salary** table.

```
1 SELECT sum(salary) AS TotalSalary FROM Salary
2
3
4
5
6
```

TotalSalary
105000

11. Write an SQL query to sort the names in alphabetical order but in ascending way from the employee table.

SELECT * FROM Employee order by name ASC

The query uses the **ORDER BY** clause with **ASC (ascending)** to sort the names alphabetically from the **Employee** table. Sorting is performed based on the values in the **Name** column. This query is useful for arranging data in a specified order for reporting, display, or analysis purposes, ensuring that names are listed in an organized manner from A to Z.

```
1 SELECT * FROM Employee ORDER BY name ASC
2
3
4
5
6
```

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

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

SELECT EmpId, Salary + Variable as TotalSalary FROM Salary;

- **SELECT EmpId, Salary + Variable AS TotalSalary:** This part of the query selects the **EmpId** column from the Salary table and calculates the total salary (**TotalSalary**) by adding the Salary column with the Variable column for each employee. **AS TotalSalary** assigns an alias to this calculated value for clarity in the result set.
- **FROM Salary:** Specifies the table from which data should be retrieved, which is the Salary table in this case.

```
1 SELECT EmpId, Salary + Variable AS TotalSalary FROM Salary;
2
3
4
5
6
```

EmpId	TotalSalary
121	20000
321	36000
421	53000

13. 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%';

This query fetches the names of employees whose names start with any two characters, followed by "vi", and then any number of characters afterward. The pattern matching with **LIKE** is useful in scenarios where need to search for names that fit a particular structure or pattern, especially when the exact name is not known or when handling user inputs with wildcards in search functionalities.

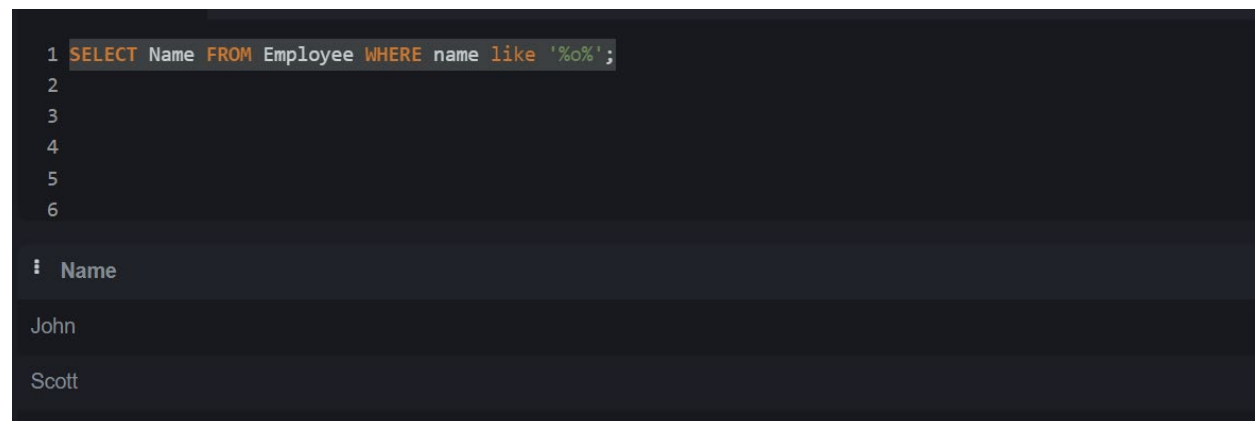
```
1 SELECT Name FROM Employee WHERE name like '__vi%';
2
3
4
5
6
```

Name
David

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

SELECT Name from Employee where name like '%o%';

This query fetches the names of employees whose names contain the letter 'o' at any position. The pattern **%o%** ensures that the letter 'o' can appear anywhere within the name, allowing for flexible and comprehensive matching. Using the **LIKE** operator with the **% wildcard** is useful in scenarios where we need to perform partial matching or search for specific characters within strings, such as filtering names, descriptions, or any text fields in a database.



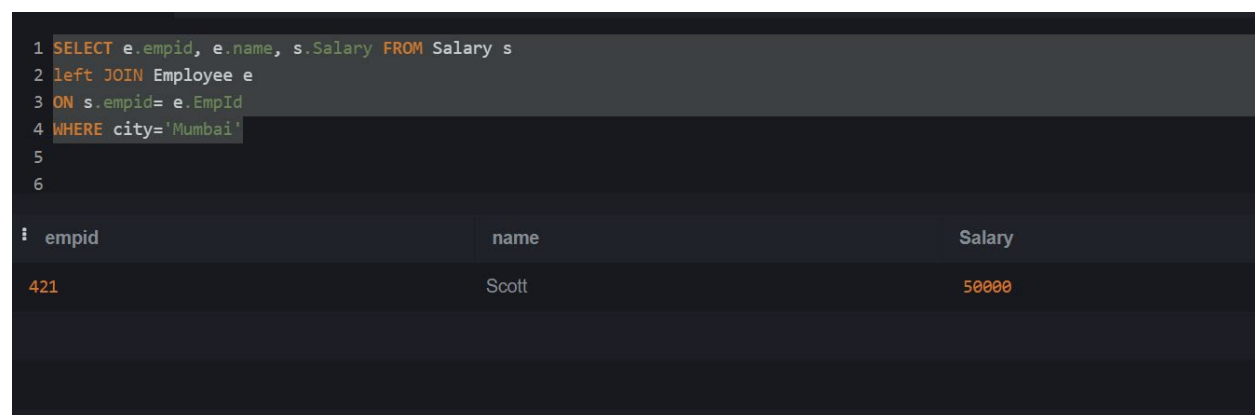
The screenshot shows a SQL query editor with the query: `1 SELECT Name FROM Employee WHERE name like 'o%';`. Below the query, the results are displayed in a table with one column, 'Name'. The results are 'John' and 'Scott'.

Name
John
Scott

15. Write an SQL query to show the salary of the employees with empid and name who work in Mumbai.

**SELECT e.empid, e.name, s.Salary from Salary s
left JOIN Employee e
ON s.empid= e.EmpId
where city='Mumbai'**

This query joins the **Salary and Employee tables** to combine the data, and it filters the results to show only those employees whose city is 'Mumbai'. The query retrieves the **empid, name, and salary** of such employees. Using the **LEFT JOIN** ensures that even if some employees in the **Salary** table do not have corresponding entries in the **Employee** table, they will still be included in the result with **NULL** values for the unmatched columns from the Employee table. However, because of the **WHERE** clause filtering for **city = 'Mumbai'**, only employees present in both tables who work in Mumbai are included in the final result.



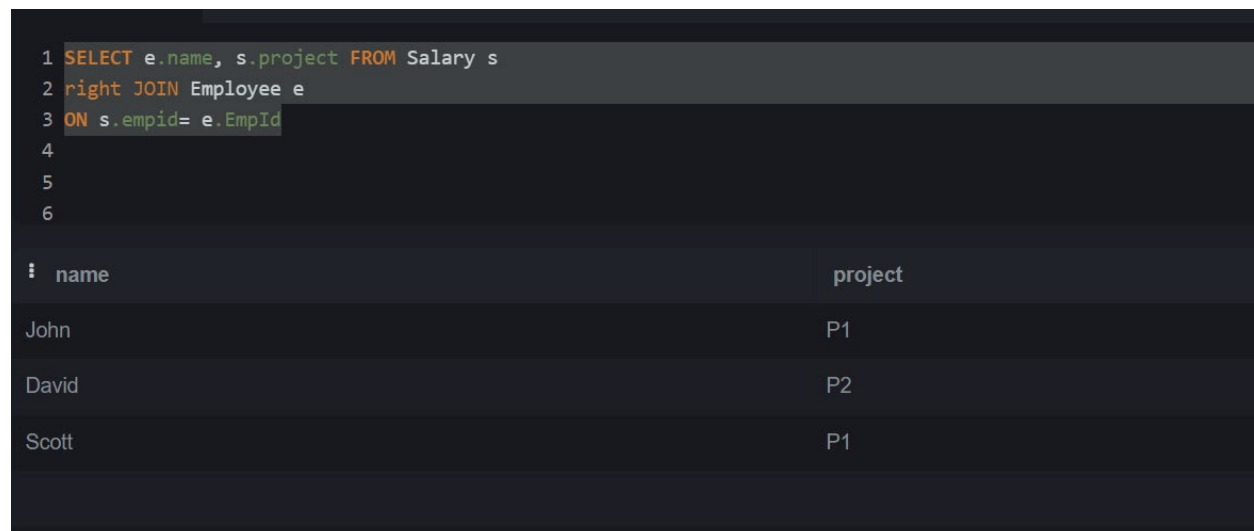
The screenshot shows a SQL query editor with the query: `1 SELECT e.empid, e.name, s.Salary FROM Salary s
2 left JOIN Employee e
3 ON s.empid= e.EmpId
4 WHERE city='Mumbai'`. Below the query, the results are displayed in a table with three columns: 'empid', 'name', and 'Salary'. The results are 421, Scott, and 50000.

empid	name	Salary
421	Scott	50000

16. Write an SQL query to show who works under which project

```
SELECT e.name, s.project from Salary s
right JOIN Employee e
ON s.empid= e.EmpId
```

This query joins the **Salary** and **Employee** tables to combine the data, and it retrieves the name of employees and the project they are working on. Using the **RIGHT JOIN** ensures that all **employees** from the **Employee** table are included in the result, even if they do not have corresponding entries in the **Salary** table. This means that every employee will be listed, and for those who do not have an associated project in the **Salary** table, the project column will have a **NULL** value. This query is useful for showing which project each employee is working on, and it also highlights employees who might not be assigned to any project.



```
1 SELECT e.name, s.project FROM Salary s
2 right JOIN Employee e
3 ON s.empid= e.EmpId
4
5
6
```

name	project
John	P1
David	P2
Scott	P1

17. Write an SQL query to fetch the empid, names and doj of employees who joined before 2018.

```
SELECT e.EmpId, e.Name, e.DOJ from Employee e
where doj< '2018-01-01'
```

- **SELECT e.EmpId, e.Name, e.DOJ:** This part of the query specifies the columns to be retrieved from the Employee table. It selects the **EmpId**, **Name**, and **DOJ (Date of Joining)** columns.
- **FROM Employee e:** This indicates the table from which to retrieve the data. The Employee table is given an **alias** e for easier reference.
- **WHERE e.DOJ < '2018-01-01':** This is the condition that filters the results to include only those employees who joined before January 1, 2018. The **DOJ** column is compared with the date '2018-01-01'.

1	SELECT e.EmpId, e.Name, e.DOJ FROM Employee e
2	WHERE doj< '2018-01-01'
3	
4	
5	
6	
7	

EmpId	Name	DOJ
121	John	2016-01-31 00:00:00

18. Write an SQL query to fetch the highest salary in each project.

SELECT Project, MAX(Salary) AS MaxSalary
FROM Salary
GROUP BY Project;

This SQL query fetches the highest salary for each project by utilizing the **GROUP BY clause** to aggregate data. Specifically, the **SELECT** statement includes the Project column and uses the MAX function to calculate the maximum salary within each project group, labeling this result as **MaxSalary**. The **FROM Salary** clause indicates that the data is sourced from the Salary table. The **GROUP BY Project** clause ensures that the data is grouped by the Project column, which allows the **MAX** function to compute the maximum salary for each distinct project.

1	SELECT Project, MAX(Salary) AS MaxSalary
2	FROM Salary
3	GROUP BY Project;
4	
5	
6	
7	

Project	MaxSalary
P1	50000
P2	35000

19. Write an SQL query to fetch the employee details along with their total salary (Salary + Variable) if the total salary is more than 25000.

```
SELECT e.EmpId, e.Name, e.City, (s.Salary + s.Variable) AS TotalSalary FROM Employee e
JOIN Salary s
ON e.EmpId = s.EmpId
WHERE (s.Salary + s.Variable) > 25000;
```

This query retrieves the details of employees whose total salary, calculated as the sum of their fixed salary (**Salary**) and variable pay (**Variable**), exceeds 25,000. The query begins by selecting the employee ID (**EmpId**), name (**Name**), and city (**City**) from the Employee table and the computed total salary (aliased as **TotalSalary**) from the Salary table.

The **JOIN** clause is used to combine the **Employee** and **Salary** tables based on the matching **EmpId** field. The **WHERE** clause filters the results to include only those employees whose combined salary (**Salary** + **Variable**) is greater than 25,000. This ensures that the query returns detailed information about employees who earn above a specified total salary threshold.

```
1 SELECT e.EmpId, e.Name, e.City, (s.Salary + s.Variable) AS TotalSalary FROM Employee e
2 JOIN Salary s
3 ON e.EmpId = s.EmpId
4 WHERE (s.Salary + s.Variable) > 25000;
5
6
7
```

EmpId	Name	City	TotalSalary
321	David	Chennai	36000
421	Scott	Mumbai	53000

20. Write an SQL query to fetch the number of employees working in each city.

```
SELECT City, COUNT(*) AS EmployeeCount FROM Employee
GROUP BY City;
```

This query aims to count the number of employees working in each city. It selects the City column and uses the **COUNT(*)** function to count the total number of employees in each city, labeling this count as **EmployeeCount** from the **Employee** table. The **GROUP BY** City clause groups the results by the **City** column, ensuring that the count of employees is calculated for each distinct city.

```

1 SELECT City, COUNT(*) AS EmployeeCount FROM Employee
2 GROUP BY City;
3
4
5
6
7

```

City	EmployeeCount
Chennai	1
hyd	1
Mumbai	1

21. Write an SQL query to fetch the employee details who have the same joining date as Scott.

```

SELECT e.EmpId, e.Name, e.ManagerId, e.DOJ, e.City
FROM Employee e
WHERE e.DOJ = (SELECT DOJ FROM Employee WHERE Name = 'Scott');

```

This query retrieves the details of employees who share the same joining date as an employee named Scott. The query selects the employee ID (EmpId), name (Name), manager ID (ManagerId), date of joining (DOJ), and city (City) from the Employee table, aliased as e. The WHERE clause filters the results to include only those employees whose date of joining (DOJ) matches the date of joining of Scott. This is achieved using a subquery (SELECT DOJ FROM Employee WHERE Name = 'Scott') that retrieves Scott's date of joining. By comparing each employee's DOJ with Scott's DOJ, the query returns a list of employees who joined the company on the same date as Scott, providing detailed information about these employees.

```

1 SELECT e.EmpId, e.Name, e.ManagerId, e.DOJ, e.City
2 FROM Employee e
3 WHERE e.DOJ = (SELECT DOJ FROM Employee WHERE Name = 'Scott');
4
5
6
7

```

EmpId	Name	ManagerId	DOJ	City
421	Scott	876	2020-11-27 00:00:00	Mumbai

22. Write an SQL query to fetch the second highest salary from the Salary table.

```
SELECT MAX(Salary) AS SecondHighestSalary  
FROM Salary  
WHERE Salary < (SELECT MAX(Salary) FROM Salary);
```

This query is designed to fetch the second highest salary from the **Salary** table. The query uses a **subquery** to achieve this. First, it selects the maximum salary from the **Salary** table using **SELECT MAX(Salary) FROM Salary** and then excludes this maximum salary in the outer query. The **outer query** again selects the maximum salary from the Salary table, but this time only considers salaries that are less than the maximum salary found in the subquery. This ensures that the highest salary is excluded, and the next highest value is returned as **SecondHighestSalary**.

```
1 SELECT MAX(Salary) AS SecondHighestSalary  
2 FROM Salary  
3 WHERE Salary < (SELECT MAX(Salary) FROM Salary);  
4  
5  
6  
7
```

SecondHighestSalary

35000

Example-2

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

Create Location Table:

```
CREATE TABLE LOCATION (  
  LOCATION_ID INT,  
  REGIONAL_GROUP varchar(255),  
  PRIMARY KEY (LOCATION_ID)  
);
```

Create Department Table:

```
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 Job Table:

```
CREATE TABLE JOB(
JOB_ID INT,
FUNCTION_JOB varchar(255),
PRIMARY KEY (JOB_ID)
);
```

Create Employee Table:

```
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)
);
```

Insert data into the Location Table:

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

Insert data into the Department Table:

```
INSERT INTO DEPARTMENT VALUES
(10, 'ACCOUNTING', 122),
```

```
(20, 'RESEARCH', 124),  
(30, 'SALES', 123),  
(40, 'OPERATIONS', 167);
```

Insert Data into the Job Table:

```
INSERT INTO JOB VALUES  
(667, 'CLERK'),  
(668, 'STAFF'),  
(669, 'ANALYST'),  
(670, 'SALESPERSON'),  
(671, 'MANAGER'),  
(672, 'PRESIDENT');
```

Insert Data into the Employee Table:

```
INSERT INTO EMPLOYEE VALUES  
(7369, 'SMITH', 'JOHN', 'Q', 667, 7902, '1984-12-17', 800, NULL, 20),  
(7499, 'ALLEN', 'KEVIN', 'J', 670, 7698, '1985-02-20', 1600, 300, 30),  
(7505, 'DOYLE', 'JEAN', 'K', 671, 7839, '1985-04-4', 2850, NULL, 30),  
(7506, 'DENNIS', 'LYNN', 'S', 671, 7839, '1985-05-15', 2750, NULL, 30),  
(7507, 'BAKER', 'LESLIE', 'D', 671, 7839, '1985-06-10', 2200, NULL, 40),  
(7521, 'WARK', 'CYNTHIA', 'D', 670, 7698, '1985-02-22', 1250, NULL, 40);
```

I am getting the values from 4 tables by using (SELECT * from Location), (SELECT * from Department), (SELECT * from Job), (SELECT * from Employee)

```
1 SELECT * FROM Location  
2  
3  
4  
5  
6
```

LOCATION_ID	REGIONAL_GROUP
122	NEW YORK
123	DALLAS
124	CHICAGO
167	BOSTON

```
1 SELECT * FROM Department
2
3
4
5
6
```

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

```
1 SELECT * FROM Job
2
3
4
5
6
```

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

```

1 SELECT * FROM Employee
2
3
4
5
6

```

EMPLO...	LAST_NA...	FIRST_N...	MIDDLE_...	JOB_ID	MANAGE...	HIRE_DATE	SALARY	COMM	DEPARTMENT_ID
7369	SMITH	JOHN	Q	667	7902	1984-12-17...	800	NULL	20
7499	ALLEN	KEVIN	J	670	7698	1985-02-20...	1600	300	30
7505	DOYLE	JEAN	K	671	7839	1985-04-04...	2850	NULL	30
7506	DENNIS	LYNN	S	671	7839	1985-05-15...	2750	NULL	30
7507	BAKER	LESLIE	D	671	7839	1985-06-10...	2200	NULL	40
7521	WARK	CYNTHIA	D	670	7698	1985-02-22...	1250	NULL	40

1. List out the employees who are not receiving the commission.

SELECT * from EMPLOYEE
where COMM is NULL;

This query retrieves all records from the **EMPLOYEE** table where the **COMM** (commission) column is **NULL**. This means it filters out employees who are not receiving any commission. **NULL** represents the absence of a value. The query uses the **SELECT *** statement to select all columns from the **EMPLOYEE** table, ensuring a comprehensive view of the data. The **WHERE** clause applies the condition **COMM IS NULL** to filter the records, ensuring only those employees whose commission value is not specified (**NULL**) are included in the result set.

```

1 SELECT * FROM EMPLOYEE
2 WHERE COMM is NULL;
3
4
5
6

```

EMPLO...	LAST_NA...	FIRST_N...	MIDDLE_...	JOB_ID	MANAGE...	HIRE_DATE	SALARY	COMM	DEPARTMENT_ID
7369	SMITH	JOHN	Q	667	7902	1984-12-17...	800	NULL	20
7505	DOYLE	JEAN	K	671	7839	1985-04-04...	2850	NULL	30
7506	DENNIS	LYNN	S	671	7839	1985-05-15...	2750	NULL	30
7507	BAKER	LESLIE	D	671	7839	1985-06-10...	2200	NULL	40
7521	WARK	CYNTHIA	D	670	7698	1985-02-22...	1250	NULL	40

2. List out the employees who are working in department 30 and draw the salaries of more than 1500.

**SELECT * from Employee
where department_id= 30 AND salary > 1500**

This query retrieves all records from the **Employee** table where the employees are working in department 30 and have a salary greater than 1500. The **SELECT *** statement selects all columns from the **Employee** table, and the **WHERE** clause applies two conditions: department_id = 30 to filter employees by their department, and salary > 1500 to filter employees by their salary.

```
1 SELECT * FROM Employee
2 WHERE department_id= 30 AND salary > 1500
3
4
5
6
```

EMPLO...	LAST_NAME	FIRST_NA...	MIDDLE_N...	JOB_ID	MANAGE...	HIRE_DATE	SALARY	COMM	DEPARTM...
7499	ALLEN	KEVIN	J	670	7698	1985-02-20 ...	1600	300	30
7505	DOYLE	JEAN	K	671	7839	1985-04-04 ...	2850	NULL	30
7506	DENNIS	LYNN	S	671	7839	1985-05-15 ...	2750	NULL	30

3. List out the employee id, name in descending order based on the salary column.

**SELECT employee_id, first_name, salary from Employee
order by salary DESC**

This query retrieves the **employee_id**, **first_name**, and **salary** columns from the **Employee** table and sorts the results in descending order based on the **salary** column. The **SELECT** statement specifies the columns to be retrieved, and the **ORDER BY salary DESC** clause sorts the data so that the highest salaries appear first.

```
1 SELECT employee_id, first_name, salary FROM Employee
2 ORDER BY salary DESC
3
4
5
6
```

employee_id	first_name	salary
7505	JEAN	2850
7506	LYNN	2750
7507	LESLIE	2200
7499	KEVIN	1600
7521	CYNTHIA	1250
7369	JOHN	800

4. How many employees, who are working in different departments, are there in the organization.

```
SELECT department_id, COUNT(*) as TotalEmployee_Department from EMPLOYEE  
GROUP BY department_id;
```

This query retrieves the count of employees in each department within the organization. The **SELECT** statement specifies the column name to retrieve the **department_id** and the count of employees **COUNT(*)** in each department. The **GROUP BY department_id** clause groups the rows based on the **department_id** column, ensuring that the count is calculated for each distinct department. The result of this query will be a list of departments along with the total number of employees in each department, which provides insight into the distribution of the workforce across different departments.

```
1 SELECT department_id, COUNT(*) AS TotalEmployee_Department FROM EMPLOYEE
2 GROUP BY department_id;
3
4
5
6
```

department_id	TotalEmployee_Department
20	1
30	3
40	2

5. List out the department id having at least 3 employees.

```
SELECT department_id, COUNT(*) as NumberOfEmployees from EMPLOYEE  
GROUP BY department_id having count(*)>=3;
```

This query retrieves the **department_id** and the number of employees in each department, but only for departments that have at least 3 employees. The **SELECT** statement specifies the columns to retrieve, and the **GROUP BY department_id** clause groups the rows based on the **department_id** column. The **HAVING COUNT(*) >= 3** clause filters the results to include only those departments where the count of employees is 3 or more.

```
1 SELECT department_id, COUNT(*) AS NumberOfEmployees FROM EMPLOYEE
2 GROUP BY department_id HAVING COUNT(*)>=3;
3
4
5
6
```

department_id	NumberOfEmployees
30	3

6. Display the employees who are working in the Research department.

**SELECT * from Employee
where department_id IN (select department_id from DEPARTMENT where name= 'Research')**

This query fetches all employees who are working in the Research department. The main **SELECT** statement fetches all columns from the **Employee** table, and the **WHERE** clause filters the results to include only those employees whose **department_id** matches any **department_id** in the **DEPARTMENT** table where the department name is 'Research'. The subquery (**SELECT department_id FROM DEPARTMENT WHERE name = 'Research'**) finds the department ID(s) corresponding to the Research department.

```
1 SELECT * FROM Employee
2 WHERE department_id IN (SELECT department_id FROM DEPARTMENT WHERE name= 'Research')
3
4
5
6
```

EMPLO...	LAST_NAME	FIRST_NA...	MIDDLE_N...	JOB_ID	MANAGE...	HIRE_DATE	SALARY	COMM	DEPARTM...
7369	SMITH	JOHN	Q	667	7902	1984-12-17 ...	800	NULL	20

7. Display the employees who got the maximum salary.

**SELECT * from Employee
where salary= (select max(salary) from Employee)**

This query fetches all employees who have the maximum salary in the **Employee** table. The main **SELECT** statement selects all columns (*) from the **Employee** table, and the **WHERE** clause filters the results to include only those employees whose salary matches the maximum salary calculated in the subquery (**SELECT MAX (salary) FROM Employee**).

```
1 SELECT * FROM Employee
2 WHERE salary= (SELECT max(salary) FROM Employee)
3
4
5
6
```

EMPLO...	LAST_NAME	FIRST_NA...	MIDDLE_N...	JOB_ID	MANAGE...	HIRE_DATE	SALARY	COMM	DEPARTM...
7505	DOYLE	JEAN	K	671	7839	1985-04-04 ...	2850	NULL	30

8. Display the employees who are working in “Boston”.

```
SELECT * from EMPLOYEE where DEPARTMENT_ID = (select DEPARTMENT_ID from  
DEPARTMENT where LOCATION_ID = (select LOCATION_ID from LOCATION where  
REGIONAL_GROUP= 'BOSTON'))
```

- **SELECT * FROM EMPLOYEE:** Selects all columns specifies the EMPLOYEE table from which data is being fetched.
- **WHERE DEPARTMENT_ID:** Filters the results to include only those employees whose DEPARTMENT_ID matches the result of the subquery (SELECT DEPARTMENT_ID FROM DEPARTMENT WHERE LOCATION_ID = (SELECT LOCATION_ID FROM LOCATION WHERE REGIONAL_GROUP = 'BOSTON')).
- **SELECT DEPARTMENT_ID FROM DEPARTMENT WHERE LOCATION ID:** This subquery selects the DEPARTMENT_ID from the DEPARTMENT table where the LOCATION_ID matches the result of another subquery.
- **SELECT LOCATION_ID FROM LOCATION WHERE REGIONAL_GROUP = 'BOSTON':** This innermost subquery selects the LOCATION_ID from the LOCATION table where the REGIONAL_GROUP is 'BOSTON'.

```
SELECT * FROM EMPLOYEE e  
JOIN Department d  
ON e.DEPARTMENT_ID = d.department_id  
JOIN Location l  
ON d.LOCATION_ID = l.Location_id where l.regional_group='BOSTON'
```

- **SELECT * FROM EMPLOYEE e:** Selects all columns specifies the EMPLOYEE table and assigns it an alias e for easier reference in the query.
- **JOIN Department d ON e.DEPARTMENT_ID = d.department_id:** Joins the Department table (d alias) to the EMPLOYEE table based on the DEPARTMENT_ID column, linking employees to their respective departments.
- **JOIN Location l ON d.LOCATION_ID = l.Location_id:** Joins the Location table (l alias) to the Department table based on the LOCATION_ID column, linking departments to their locations.
- **WHERE l.regional_group='BOSTON':** Filters the results to include only those employees whose department's location has a regional_group value of 'BOSTON'.


```

1 SELECT * FROM EMPLOYEE
2 WHERE DEPARTMENT_ID=(SELECT DEPARTMENT_ID FROM DEPARTMENT
3     WHERE LOCATION_ID=(SELECT LOCATION_ID FROM LOCATION WHERE REGIONAL_GROUP= 'BOSTON'))
4
5
6 SELECT * FROM EMPLOYEE e
7 JOIN Department d
8 ON e.DEPARTMENT_ID = d.department_id
9 JOIN Location l
10 ON d.LOCATION_ID = l.Location_id WHERE l.regional_group='BOSTON'
11

```

EMPLO...	LAST_NAME	FIRST_NA...	MIDDLE_N...	JOB_ID	MANAGE...	HIRE_DATE	SALARY	COMM	DEPARTM...
7507	BAKER	LESLIE	D	671	7839	1985-06-10 ...	2200	NULL	40
7521	WARK	CYNTHIA	D	670	7698	1985-02-22 ...	1250	NULL	40

9. Update the employees' salaries, who are working as Manager on the basis of 10%.

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

Before updating, salary in the Employee table looks like that:

```

1 SELECT first_name, job_id, salary FROM EMPLOYEE
2 WHERE job_id=(SELECT job_id FROM JOB
3     WHERE[Function_Job] = 'MANAGER')
4
5
6
7
~

```

first_name	job_id	salary
JEAN	671	2850
LYNN	671	2750
LESLIE	671	2200

**UPDATE EMPLOYEE
set salary = salary + (salary * 10/100)
where job_id= (SELECT job_id from JOB
WHERE[Function_Job] = 'MANAGER');**

**SELECT first_name, job_id, salary from EMPLOYEE
where job_id= (SELECT job_id from JOB
WHERE[Function_Job] = 'MANAGER')**

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.

- **UPDATE EMPLOYEE:** Specifies the table EMPLOYEE that you want to update.
- **SET salary = salary + (salary * 10/100):** Increases the salary of each employee who holds a job identified as 'MANAGER' by 10%.
- **WHERE job_id = (SELECT job_id FROM JOB WHERE [Function_Job] = 'MANAGER');** Filters the rows in EMPLOYEE to update only those where the job_id matches the job_id retrieved from the JOB table for jobs with [Function_Job] as 'MANAGER'.

The **SELECT** query in SQL is used to fetch the selected columns from specific table and return rows based on specified conditions.

- **SELECT first_name, job_id, salary:** Fetches the first_name, job_id, and salary columns from the EMPLOYEE table.
- **FROM EMPLOYEE:** Specifies the table from which to select data.
- **WHERE job_id = (SELECT job_id FROM JOB WHERE [Function_Job] = 'MANAGER');** Filters the results to include only those employees whose job_id matches the job_id retrieved from the JOB table where [Function_Job] is 'MANAGER'.

These SQL queries together update the salaries of employees with the job title 'MANAGER' by increasing them by 10%, and then retrieve the first_name, job_id, and updated salary of those employees. This approach ensures that employees in managerial roles receive a salary adjustment based on the specified criteria.

After updating the salary values, Employee Table looks like that:

```
1 UPDATE EMPLOYEE
2 SET salary = salary + (salary * 10/100)
3 WHERE job_id=(SELECT job_id FROM JOB
4 WHERE[Function_Job] = 'MANAGER');
5
6 SELECT first_name, job_id, salary FROM EMPLOYEE
7 WHERE job_id=(SELECT job_id FROM JOB
8 WHERE[Function_Job] = 'MANAGER')
9
10
11
```

first_name	job_id	salary
JEAN	671	3135
LYNN	671	3025
LESLIE	671	2420

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

```
SELECT e.employee_id, e.first_name, e.department_id, d.name from EMPLOYEE e
JOIN DEPARTMENT d
On e.department_id= d.department_id;
```

Before deleting, the table looks like that:

```
1 SELECT e.employee_id, e.first_name, e.department_id, d.name FROM EMPLOYEE e
2 Join DEPARTMENT d
3 ON e.department_id= d.department_id;
4
5
6
```

employee_id	first_name	department_id	name
7369	JOHN	20	RESEARCH
7499	KEVIN	30	SALES
7505	JEAN	30	SALES
7506	LYNN	30	SALES
7507	LESLIE	40	OPERATIONS
7521	CYNTHIA	40	OPERATIONS

DELETE from EMPLOYEE

```
where department_id = (SELECT department_id from DEPARTMENT
where name = 'OPERATIONS');
```

```
SELECT e.employee_id, e.first_name, e.department_id, d.name from EMPLOYEE e
Join DEPARTMENT d
On e.department_id= d.department_id;
```

The DELETE statement removes targeted employee records based on departmental criteria.

- **DELETE FROM EMPLOYEE:** Removes rows from the EMPLOYEE table.
- **WHERE department_id = (SELECT department_id FROM DEPARTMENT WHERE name = 'OPERATIONS');** Specifies the condition under which rows will be deleted. It deletes employees whose department_id matches the department_id of the 'OPERATIONS' department obtained from the DEPARTMENT table.

The SELECT query retrieves employee details post-deletion, illustrating effective data querying and manipulation practices in SQL.

- **SELECT e.employee_id, e.first_name, e.department_id, d.name:** Retrieves columns (employee_id, first_name, department_id, name) from the EMPLOYEE table (e) and the DEPARTMENT table (d).
- **FROM EMPLOYEE e:** Specifies the EMPLOYEE table with alias e as the source of data.
- **JOIN DEPARTMENT d ON e.department_id = d.department_id:** Joins the EMPLOYEE table (e) with the DEPARTMENT table (d) based on matching department_id values.

After Deleting, the table looks like that:

```

1 DELETE FROM EMPLOYEE
2 WHERE department_id = (SELECT department_id FROM DEPARTMENT
3 WHERE name = 'OPERATIONS');
4
5 SELECT e.employee_id, e.first_name, e.department_id, d.name FROM EMPLOYEE e
6 Join DEPARTMENT d
7 ON e.department_id= d.department_id;
8
9

```

employee_id	first_name	department_id	name
7369	JOHN	20	RESEARCH
7499	KEVIN	30	SALES
7505	JEAN	30	SALES
7506	LYNN	30	SALES

11. Insert the employees who belong to the previous deleted 'OPERATIONS' department from a backup source into Employee table.

```

INSERT INTO EMPLOYEE (employee_id, last_name, first_name, middle_name, job_id,
manager_id, hire_date, salary, comm, department_id)
SELECT e.employee_id, e.last_name, e.first_name, e.middle_name, e.job_id, e.manager_id,
e.hire_date, e.salary, e.comm, e.department_id
FROM Backup_EMPLOYEES e
JOIN DEPARTMENT d ON e.department_id = d.department_id
WHERE d.name = 'OPERATIONS';

```

This query is used for restoring data from a backup (**Backup_EMPLOYEES**) into the **EMPLOYEE** table specifically for employees who were previously associated with the 'OPERATIONS' department.

It ensures that only relevant employees are inserted based on the department criteria specified in the **WHERE** clause (**d.name = 'OPERATIONS'**). It adjusts column names and table references as per actual database schema.

```

1 INSERT INTO EMPLOYEE (employee_id, last_name, first_name, middle_name, job_id, manager_id, hire_date, salary, comm, department_id)
2 SELECT e.employee_id, e.last_name, e.first_name, e.middle_name, e.job_id, e.manager_id, e.hire_date, e.salary, e.comm, e.department_id
3 FROM Backup_EMPLOYEES e
4 JOIN DEPARTMENT d ON e.department_id = d.department_id WHERE d.name = 'OPERATIONS';
5
6 SELECT e.employee_id, e.first_name, e.department_id, d.name FROM EMPLOYEE e
7 JOIN DEPARTMENT d
8 ON e.department_id= d.department_id;

```

employee_id	first_name	department_id	name
7369	JOHN	20	RESEARCH
7499	KEVIN	30	SALES
7505	JEAN	30	SALES
7506	LYNN	30	SALES
7507	LESLIE	40	OPERATIONS
7521	CYNTHIA	40	OPERATIONS

12. Write an SQL query to fetch all the Department ID's which are present in either of the tables- 'Employee' and 'Department'.

```

SELECT department_id from EMPLOYEE
UNION
SELECT department_id from DEPARTMENT

```

The **UNION** operator is used to merge the results of two or more **SELECT** statements. It removes duplicates by default, so only distinct **department_id** values from both tables are returned.

```

1 SELECT department_id FROM EMPLOYEE
2 UNION
3 SELECT department_id FROM DEPARTMENT
4

```

```

department_id

```

```

10

```

```

20

```

```

30

```

```

40

```

```

SELECT department_id from EMPLOYEE
UNION ALL
SELECT department_id from DEPARTMENT

```

UNION ALL does not remove duplicates. It simply concatenates the results of both queries, including all occurrences of **department_id** from both tables.

```
1 SELECT department_id FROM EMPLOYEE
2 UNION ALL
3 SELECT department_id FROM DEPARTMENT
4

+-----+
| department_id |
+-----+
| 20            |
| 30            |
| 30            |
| 30            |
| 40            |
| 40            |
| 10            |
| 20            |
| 30            |
| 40            |
+-----+
```

13. Write an SQL query to fetch common records between two tables.

```
SELECT department_id from EMPLOYEE
INTERSECT
SELECT department_id from DEPARTMENT
```

This query fetches common records (department_id values) that exist in both the **EMPLOYEE** table and the **DEPARTMENT** table. **INTERSECT** performs a set operation that returns only the common rows between the results of the preceding **SELECT** statements.

```
1 SELECT department_id FROM EMPLOYEE
2 INTERSECT
3 SELECT department_id FROM DEPARTMENT
4

+-----+
| department_id |
+-----+
| 20            |
| 30            |
| 40            |
+-----+
```

14. Write an SQL query to fetch records that are present in one table but not in another table.

```
SELECT job_id from Employee  
MINUS  
SELECT job_id from JOB
```

The **MINUS** operator is used to return rows from the first query that are not present in the second query. This query will return the job_ids that are present in the **Employee** table but not in the **JOB** table.

```
1 SELECT job_id FROM Employee  
2 MINUS  
3 SELECT job_id FROM JOB  
4  
5  
6  
  
| job_id  
|  
| 667  
| 670  
| 671  
| 671  
| 671  
| 670
```

15. Create a table from the existing table (only structure without data).

```
SELECT * INTO JOB_NEW  
FROM JOB  
WHERE 1 = 0;
```

This query creates a new table called **JOB_NEW** with the same structure as the existing **JOB** table, but without copying any data from the **JOB** table.

The **SELECT * INTO** statement copies the table structure, and the **WHERE 1 = 0** clause is always false, ensuring that no rows are copied, resulting in an empty **JOB_NEW** table that mirrors the structure of **JOB**. This method is commonly used to create a template table for various purposes, such as testing or temporary data storage.

```
MS SQL

1 SELECT * INTO JOB_NEW
2 FROM JOB
3 WHERE 1 = 0;
4
5 SELECT * FROM JOB_NEW
6
7
```

JOB_ID	FUNCTION_JOB

16. Write an SQL query to find the current date.

select getdate() as CurrentDate;

This query is used to fetch the current date and time from the database server. The **GETDATE()** function returns the current system timestamp, which includes both the date and the time. By using **AS CurrentDate**, the result is given an alias, making it easier to reference in the output. This query is particularly useful for applications that need to log the current date and time or display it to the user.

```
MS SQL

1 SELECT getdate() AS CurrentDate;
2
3
4
5
6
7
```

CurrentDate
2024-07-17 05:35:27

Example-3

Create Products Table:

```
Create table Products (  
  product_id INT IDENTITY (1,1) PRIMARY KEY,  
  product_name VARCHAR(255) NOT NULL,  
  brand_id INT NOT NULL,  
  category_id INT NOT NULL,  
  model_year SMALLINT NOT NULL,  
  list_price DECIMAL(10,2) NOT NULL,  
);
```

Create Product_Audits Table:

```
CREATE TABLE product_audits(  
  change_id INT IDENTITY PRIMARY KEY,  
  product_id INT NOT NULL,  
  product_name VARCHAR(255) NOT NULL,  
  brand_id INT NOT NULL,  
  category_id INT NOT NULL,  
  model_year SMALLINT NOT NULL,  
  list_price DEC(10,2) NOT NULL,  
  updated_at DATETIME NOT NULL,  
  operation CHAR(3) NOT NULL,  
  CHECK(operation = 'INS' or operation='DEL')  
);
```

Trigger Creation:

```
CREATE TRIGGER trg_product_audit  
  ON Products  
  AFTER INSERT, DELETE  
  AS  
  BEGIN  
  SET NOCOUNT ON;  
  INSERT INTO product_audits(  
    product_id,  
    product_name,  
    brand_id,  
    category_id,  
    model_year,  
    list_price,
```

```

        updated_at,
        operation)
SELECT
    i.product_id,
    product_name,
    brand_id,
    category_id,
    model_year,
    i.list_price,
    GETDATE(),
    'INS'
FROM
    inserted i

UNION ALL
SELECT
    d.product_id,
    product_name,
    brand_id,
    category_id,
    model_year,
    d.list_price,
    GETDATE(),
    'DEL'
FROM
    deleted d;
END

```

This trigger is defined on the Products table and fires after INSERT and DELETE operations.

- **SET NOCOUNT ON:** It prevents the message indicating the number of rows affected by a T-SQL statement from being displayed.
- **INSERT INTO product_audits:** It inserts a new record into the product_audits table whenever a row is inserted into or deleted from the Products table.
- **INSERT Operation:** It takes the data from the inserted pseudo-table (which holds the new rows) and adds it to the product_audits table with 'INS' as the operation type.
- **DELETE Operation:** It takes the data from the deleted pseudo-table (which holds the old rows) and adds it to the product_audits table with 'DEL' as the operation type.

Insert Into Products:

```

INSERT Into products(
    product_name,
    brand_id,

```

```
category_id,  
model_year,  
list_price  
)  
VALUES(  
  'Test product',  
  1,  
  1,  
  2018,  
  599  
);
```

This query inserts a new row into the **Products** table with the specified values. This will trigger the **trg_product_audit** trigger and an entry will be added to the **product_audits** table with operation 'INS'.

SELECT * from product_audits

After inserting the product into the Product Table, the product_audits table looks like that:

```
1 SELECT * FROM product_audits
```

change_id	product_id	product_name	brand_id	category_id	model_year	list_price	updated_at	operation
1	1	Test product	1	1	2018	599.00	2024-07-17 0...	INS

SELECT * from products

```
1 SELECT * FROM products
```

product_id	product_name	brand_id	category_id	model_year	list_price
1	Test product	1	1	2018	599.00

DELETE from Products where product_id=1

This query deletes the row with **product_id** 1 from the **Products** table. This will trigger the **trg_product_audit** trigger and an entry will be added to the **product_audits** table with operation 'DEL'.

SELECT * from product_audits

After deleting the product from Products table, the product_audits table looks like that:

```
1 SELECT * FROM product_audits
```

change_id	product_id	product_name	brand_id	category_id	model_year	list_price	updated_at	operation
1	1	Test product	1	1	2018	599.00	2024-07-17 0...	INS
2	1	Test product	1	1	2018	599.00	2024-07-17 0...	DEL

Show Triggers:

```
SELECT
    name,
    is_instead_of_trigger
FROM
    sys.triggers
WHERE
    type = 'TR';
```

This query retrieves the names and types of all triggers in the current database that are of type 'TR' (DML triggers). The **is_instead_of_trigger** column indicates if the trigger is an INSTEAD OF trigger or an AFTER trigger.

```
1 SELECT
2     name,
3     is_instead_of_trigger
4 FROM
5     sys.triggers
6 WHERE
7     type = 'TR';
8
9
```

name	is_instead_of_trigger
trg_product_audit	0