SQL QUESTIONS & ANSWERS

Basic SQL Concepts:

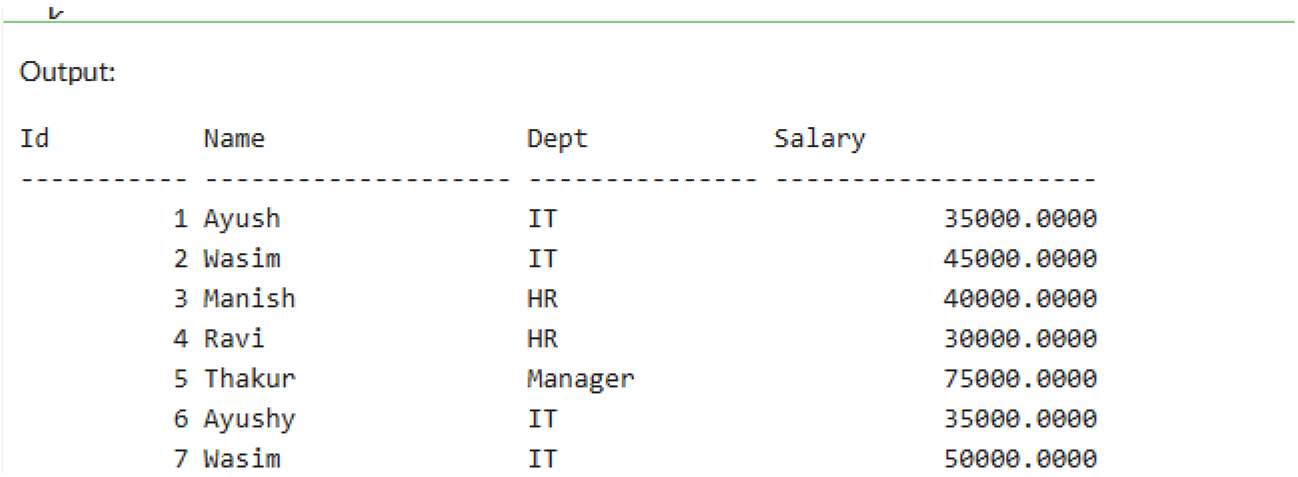
1. What is SQL, and why is it important in data analytics?

SQL (Structured Query Language) is a standardized programming language designed for managing and manipulating relational databases. It is used to perform tasks such as querying, inserting, updating, and deleting data, as well as defining and controlling database structures.

SQL plays a crucial role in data analytics because it enables analysts to work efficiently with large datasets stored in relational databases.

2. Explain the difference between `INNER JOIN`, `LEFT JOIN`, `RIGHT JOIN`, and `FULL OUTER JOIN`.

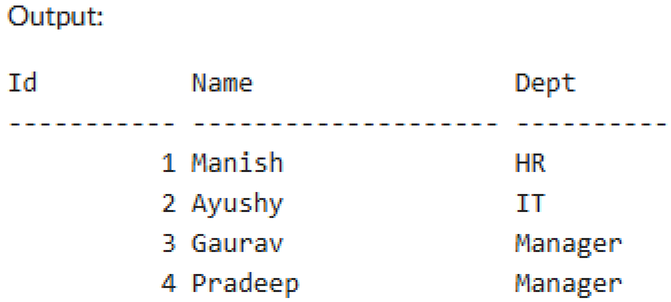
```
Create table Employee(Id int identity(1,1), Name varchar(20),

Dept varchar(15) Check (Dept in ("IT","HR","Manager")), Salary Money);


Insert into Employee Values("Ayush","IT",35000),("Wasim","IT",45000),

("Manish","HR",40000),("Ravi","HR",30000),("Thakur","Manager",75000),

("Ayushy","IT",35000),("Wasim","IT",50000);


Create table Fraud(Id int identity(1,1),Name Varchar(20),Dept char(10));


Insert into Fraud values("Manish","HR"),("Ayushy","IT"),

("Gaurav","Manager"),("Pradeep","Manager");


Select * From Employee;
```



Output:

```
Id          Name                 Dept            Salary
----------- -------------------- --------------- ---------------------
          1 Ayush                IT                         35000.0000
          2 Wasim                IT                         45000.0000
          3 Manish               HR                         40000.0000
          4 Ravi                 HR                         30000.0000
          5 Thakur               Manager                    75000.0000
          6 Ayushy               IT                         35000.0000
          7 Wasim                IT                         50000.0000
```

Select * From Fraud



Output:

```
Id          Name                 Dept
----------- -------------------- ----------
          1 Manish               HR
          2 Ayushy               IT
          3 Gaurav               Manager
          4 Pradeep              Manager
```

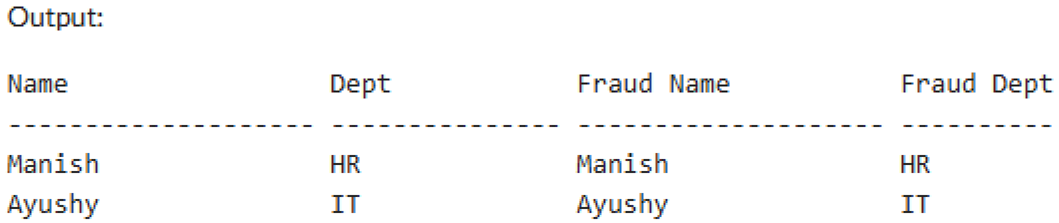
1. INNER JOIN

- **Definition**: Returns only the rows that have matching values in both tables.
- **Key Point**: If there’s no match between the two tables, the row is excluded from the result.
- **Use Case**: When you only need rows with matching data in both tables.

Select E.Name,E.Dept,F.Name [Fraud Name],F.Dept [Fraud Dept] From Employee E

Join Fraud F

On E.Name = F.Name And E.Dept = F.Dept



Output:

```
Name                 Dept            Fraud Name           Fraud Dept
-------------------- --------------- -------------------- ----------
Manish               HR              Manish               HR
Ayushy               IT              Ayushy               IT
```

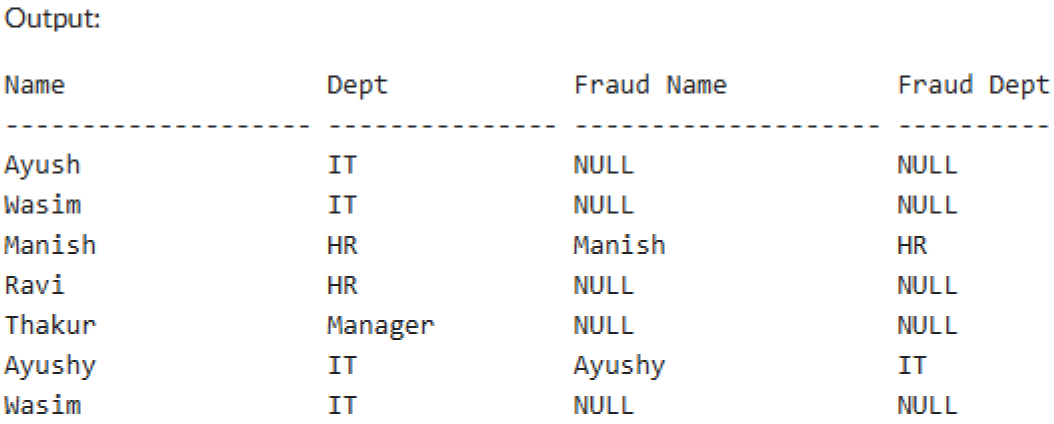
2. LEFT JOIN (or LEFT OUTER JOIN)

- **Definition**: Returns all rows from the left table, along with the matching rows from the right table. If no match exists, NULL values are returned for the right table's columns.
- **Key Point**: Always includes all rows from the left table, even if there's no match in the right table.
- **Use Case**: When you want all rows from the left table, regardless of matches in the right table.

Select E.Name,E.Dept,F.Name [Fraud Name],F.Dept [Fraud Dept] From Employee E

Left Join Fraud F

On E.Name = F.Name And E.Dept = F.Dept



```
Output:

Name                 Dept            Fraud Name           Fraud Dept
-------------------- --------------- -------------------- ----------
Ayush                IT              NULL                 NULL
Wasim                IT              NULL                 NULL
Manish               HR              Manish               HR
Ravi                 HR              NULL                 NULL
Thakur               Manager         NULL                 NULL
Ayushy               IT              Ayushy               IT
Wasim                IT              NULL                 NULL
```

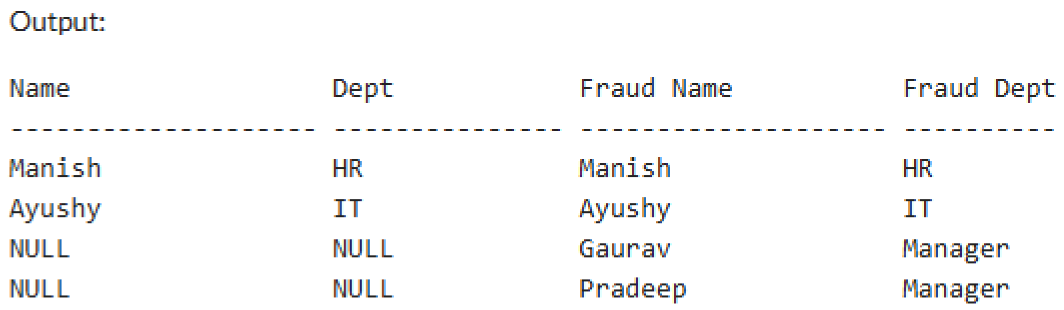
3. RIGHT JOIN (or RIGHT OUTER JOIN)

- **Definition**: Returns all rows from the right table, along with the matching rows from the left table. If no match exists, NULL values are returned for the left table's columns.
- **Key Point**: Always includes all rows from the right table, even if there's no match in the left table.
- **Use Case**: When you want all rows from the right table, regardless of matches in the left table.

Select E.Name,E.Dept,F.Name [Fraud Name],F.Dept [Fraud Dept] From Employee E

Right Join Fraud F

On E.Name = F.Name And E.Dept = F.Dept



```
Output:

Name                 Dept            Fraud Name           Fraud Dept
-------------------- --------------- -------------------- ----------
Manish               HR              Manish               HR
Ayushy               IT              Ayushy               IT
NULL                 NULL            Gaurav               Manager
NULL                 NULL            Pradeep              Manager
```

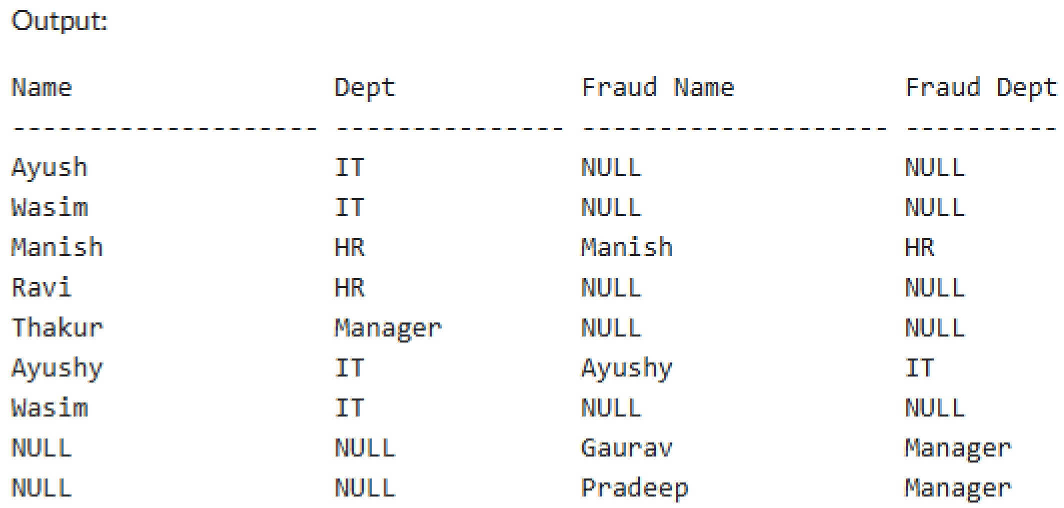
4. FULL OUTER JOIN

- **Definition**: Combines the results of LEFT JOIN and RIGHT JOIN. Returns all rows from both tables, with NULL in columns where no match exists.
- **Key Point**: Includes unmatched rows from both tables.
- **Use Case**: When you want all data, regardless of matches between the two tables.

Select E.Name,E.Dept,F.Name [Fraud Name],F.Dept [Fraud Dept] From Employee E

Full Join Fraud F

On E.Name = F.Name And E.Dept = F.Dept



```
Output:

Name                 Dept            Fraud Name           Fraud Dept
-------------------- --------------- -------------------- ----------
Ayush                IT              NULL                 NULL
Wasim                IT              NULL                 NULL
Manish               HR              Manish               HR
Ravi                 HR              NULL                 NULL
Thakur               Manager         NULL                 NULL
Ayushy               IT              Ayushy               IT
Wasim                IT              NULL                 NULL
NULL                 NULL            Gaurav               Manager
NULL                 NULL            Pradeep              Manager
```

3. What is the difference between `WHERE` and `HAVING` clauses?

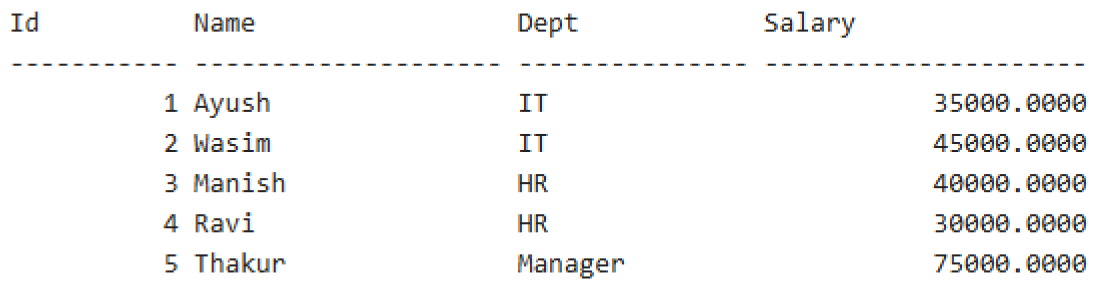
Create table Employee(Id int identity(1,1), Name varchar(20),

Dept varchar(15) Check (Dept in ("IT","HR","Manager")), Salary Money);

Insert into Employee Values("Ayush","IT",35000),("Wasim","IT",45000),

("Manish","HR",40000),("Ravi","HR",30000),("Thakur","Manager",75000);

Select * From Employee;

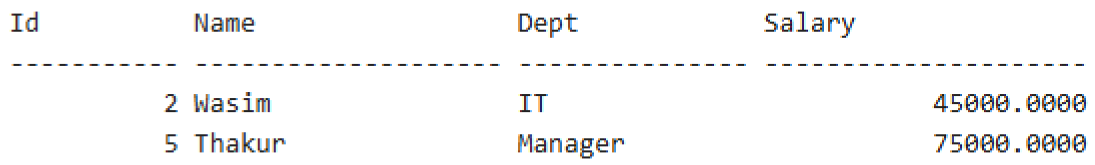


```
Id          Name                 Dept            Salary
----------- -------------------- --------------- ---------------------
          1 Ayush                IT                         35000.0000
          2 Wasim                IT                         45000.0000
          3 Manish               HR                         40000.0000
          4 Ravi                 HR                         30000.0000
          5 Thakur               Manager                    75000.0000
```

Where Clauses

- **Purpose**: Filters rows *before* grouping or aggregation happens.
- **Applies To**: Individual rows in a table.
- **Usage**: Used to specify conditions for raw data (columns without aggregate functions).
- **Example**:

Select * From Employee Where Salary > = 45000



```
Id          Name                 Dept            Salary
----------- -------------------- --------------- ---------------------
          2 Wasim                IT                         45000.0000
          5 Thakur               Manager                    75000.0000
```

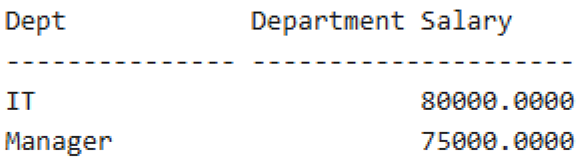
Having Clauses

- **Purpose**: Filters groups *after* aggregation (e.g., using GROUP BY).
- **Applies To**: Aggregated data (e.g., results of COUNT, SUM, AVG).
- **Usage**: Used to specify conditions on aggregate functions or grouped data.
- **Example**:

Select Dept, Sum(Salary) [Department Salary] From Employee

Group By Dept

Having Sum(Salary) > = 75000



```
Dept            Department Salary
--------------- ---------------------
IT                         80000.0000
Manager                    75000.0000
```

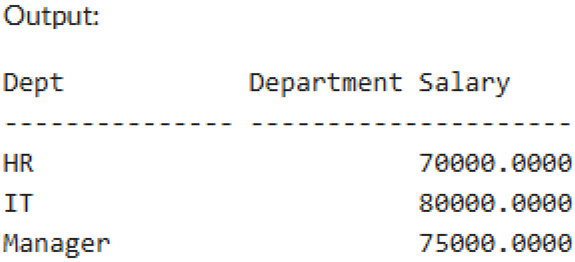
4. How do you use `GROUP BY` and `HAVING` in a query?

Group By

- Groups rows with the same values in specified columns into aggregated groups.
- Example:

Select Dept, Sum(Salary) [Department Salary] From Employee

Group By Dept



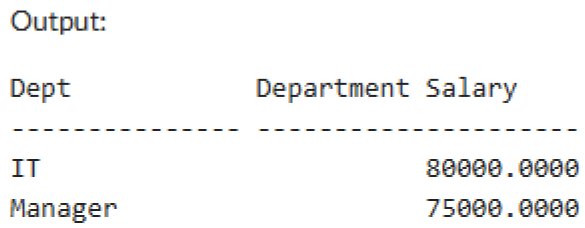
Output:

```
Dept            Department Salary
--------------- ---------------------
HR                         70000.0000
IT                         80000.0000
Manager                    75000.0000
```

Having

- Filters these aggregated groups based on conditions (can use aggregate functions like COUNT, SUM, AVG, etc.).
- Example:

  Select Dept, Sum(Salary) [Department Salary] From Employee
  Group By Dept
  Having Sum(Salary) > = 75000



```
Output:

Dept            Department Salary
--------------- ---------------------
IT                         80000.0000
Manager                    75000.0000
```

5. Write a query to find duplicate records in a table.

Create table Employee(Id int identity(1,1), Name varchar(20),

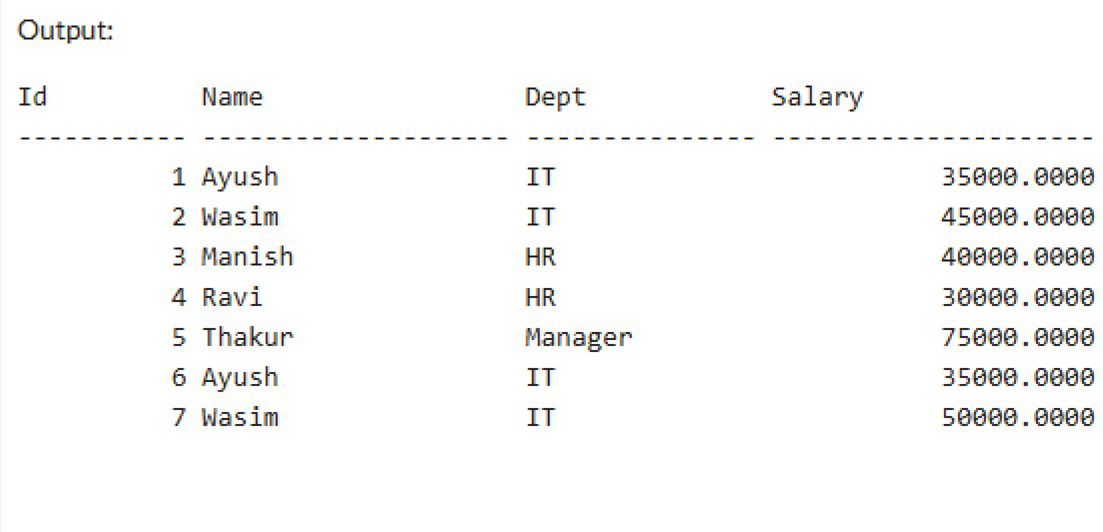
Dept varchar(15) Check (Dept in ("IT","HR","Manager")), Salary Money);

Insert into Employee Values("Ayush","IT",35000),("Wasim","IT",45000),

("Manish","HR",40000),("Ravi","HR",30000),("Thakur","Manager",75000),

("Ayush","IT",35000),("Wasim","IT",50000);

Select * From Employee;



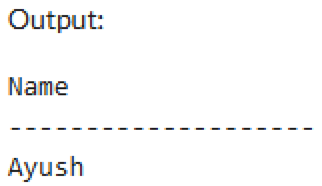
```
Output:

Id          Name                 Dept            Salary
----------- -------------------- --------------- ---------------------
          1 Ayush                IT                         35000.0000
          2 Wasim                IT                         45000.0000
          3 Manish               HR                         40000.0000
          4 Ravi                 HR                         30000.0000
          5 Thakur               Manager                    75000.0000
          6 Ayush                IT                         35000.0000
          7 Wasim                IT                         50000.0000
```

Only Id 6 is Duplicate. The salary of Id 7 is different from Id 2.

- **Using Group by and Havning**

```
Select Name From Employee
Group By Name,Dept,Salary
```

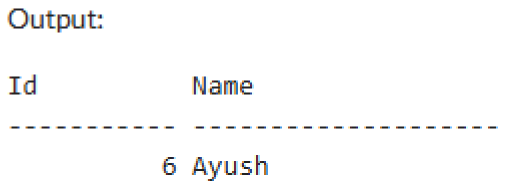
```
        Having count(Id) > 1
```



```
Output:

Name
--------------------
Ayush
```

- **Using CTE**

```
With CTE As (
Select *,
Row_number()over(partition by Name,Dept,Salary order by Id) R
From Employee
)
 Select Id, Name From CTE Where R > 1
```

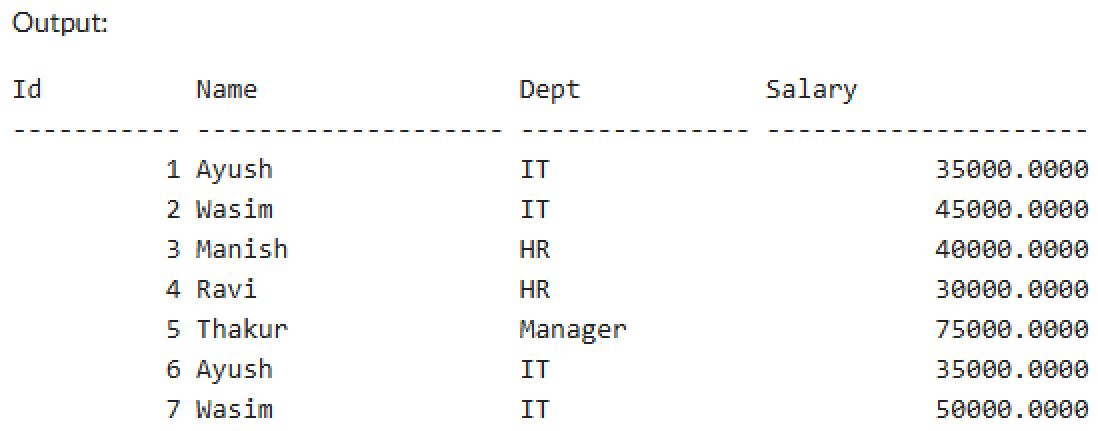


```
Output:

Id          Name
----------- --------------------
          6 Ayush
```

6. How do you retrieve unique values from a table using SQL?

```
Create table Employee(Id int identity(1,1), Name varchar(20),
Dept varchar(15) Check (Dept in ("IT","HR","Manager")), Salary Money);

Insert into Employee Values("Ayush","IT",35000),("Wasim","IT",45000),
("Manish","HR",40000),("Ravi","HR",30000),("Thakur","Manager",75000),
("Ayush","IT",35000),("Wasim","IT",50000);
```

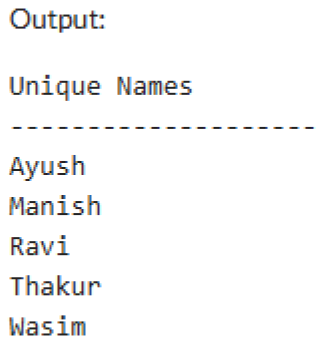
Select * From Employee



```
Output:

Id          Name                 Dept            Salary
----------- -------------------- --------------- ---------------------
          1 Ayush                IT                         35000.0000
          2 Wasim                IT                         45000.0000
          3 Manish               HR                         40000.0000
          4 Ravi                 HR                         30000.0000
          5 Thakur               Manager                    75000.0000
          6 Ayush                IT                         35000.0000
          7 Wasim                IT                         50000.0000
```

I want unique names from the Name column

Select distinct(Name) [Unique Names] From Employee



```
Output:

Unique Names
--------------------
Ayush
Manish
Ravi
Thakur
Wasim
```

7. Explain the use of aggregate functions like `COUNT()`, `SUM()`, `AVG()`, `MIN()`, and `MAX()`.

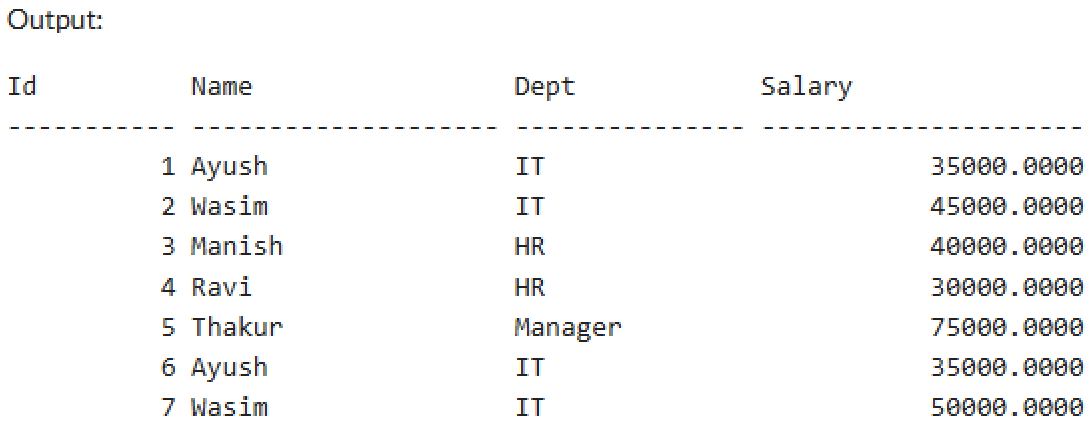
Create table Employee(Id int identity(1,1), Name varchar(20),

Dept varchar(15) Check (Dept in ("IT","HR","Manager")), Salary Money);

Insert into Employee Values("Ayush","IT",35000),("Wasim","IT",45000),

("Manish","HR",40000),("Ravi","HR",30000),("Thakur","Manager",75000), ("Ayush","IT",35000),("Wasim","IT",50000);

Select * From Employee



```
Output:

Id          Name                 Dept            Salary
----------- -------------------- --------------- ---------------------
          1 Ayush                IT                        35000.0000
          2 Wasim                IT                        45000.0000
          3 Manish               HR                        40000.0000
          4 Ravi                 HR                        30000.0000
          5 Thakur               Manager                   75000.0000
          6 Ayush                IT                        35000.0000
          7 Wasim                IT                        50000.0000
```

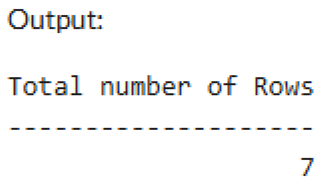
Aggregate Functions in SQL

Aggregate functions are used in SQL to perform calculations on a set of values and return a single summarized result. These functions are commonly used in combination with the GROUP BY clause to analyze grouped data, but they can also work without grouping.

1. COUNT()

- **Purpose**: Counts the number of rows or non-NULL values in a column.
- **Syntax**:

Select Count(*) [Total number of Rows] From Employee

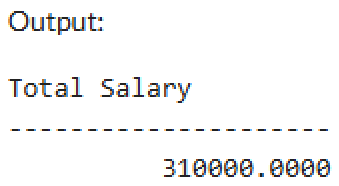


```
Output:

Total number of Rows
--------------------
                   7
```

2. SUM()

- **Purpose**: Calculates the total (sum) of a numeric column.
- **Syntax**:

Select Sum(Salary) [Total Salary] From Employee



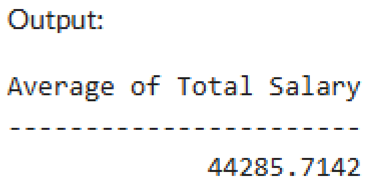
Output:

```
Total Salary
---------------------
          310000.0000
```

3. AVG()

- **Purpose**: Calculates the average of a numeric column.
- **Syntax**:

Select AVG(Salary) [Average of Total Salary] From Employee



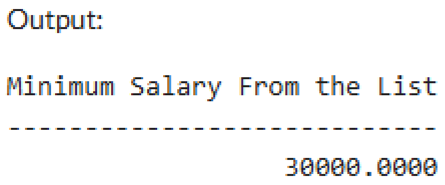
Output:

```
Average of Total Salary
-----------------------
             44285.7142
```

4. MIN()

- **Purpose**: Returns the smallest (minimum) value in a column.
- **Syntax**:

Select Min(Salary)[Minimum Salary From the List] From Employee



Output:

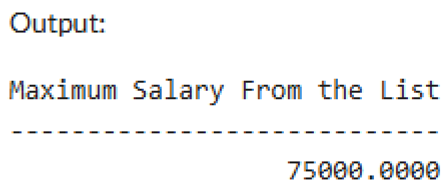
```
Minimum Salary From the List
----------------------------
                  30000.0000
```

5. MAX()

- **Purpose**: Returns the largest (maximum) value in a column.

- **Syntax**:

Select Max(Salary)[Maximum Salary From the List] From Employee



Output:

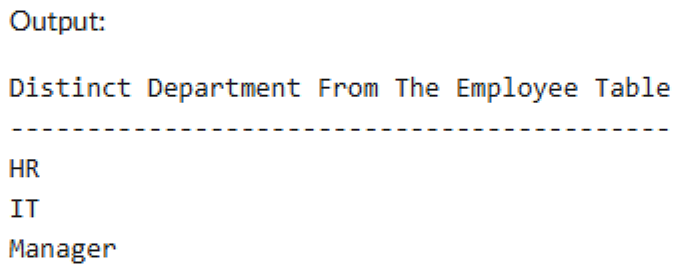
```
Maximum Salary From the List
----------------------------
                  75000.0000
```

8. What is the purpose of a `DISTINCT` keyword in SQL?

The DISTINCT keyword is used in SQL to remove duplicate rows from the result set of a query, ensuring that only unique values are returned for the specified columns. It is often used when you want to retrieve distinct data points from a table.

Example:

Select Distinct(Dept)[Distinct Department From The Employee Table] From Employee



Output:

```
Distinct Department From The Employee Table
-------------------------------------------
HR
IT
Manager
```

Intermediate SQL:

1. Write a query to find the second-highest salary from an employee table.

Create table Employee(Id int identity(1,1), Name varchar(20),

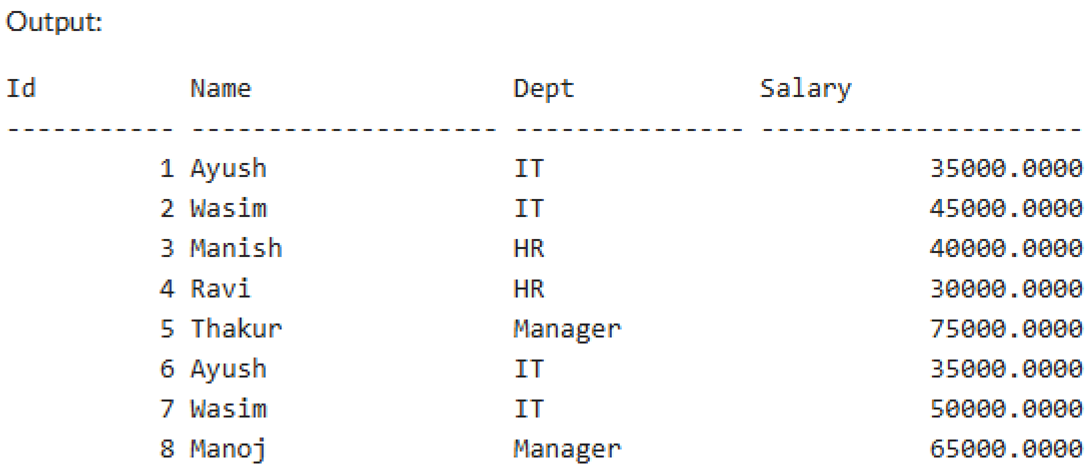
Dept varchar(15) Check (Dept in ("IT","HR","Manager")), Salary Money);

Insert into Employee Values("Ayush","IT",35000),("Wasim","IT",45000),

("Manish","HR",40000),("Ravi","HR",30000),("Thakur","Manager",75000),

("Ayush","IT",35000),("Wasim","IT",50000),("Manoj","Manager",65000);

Select * from Employee



Output:

```
Id          Name                 Dept           Salary
----------- -------------------- -------------- ---------------------
          1 Ayush                IT                        35000.0000
          2 Wasim                IT                        45000.0000
          3 Manish               HR                        40000.0000
          4 Ravi                 HR                        30000.0000
          5 Thakur               Manager                   75000.0000
          6 Ayush                IT                        35000.0000
          7 Wasim                IT                        50000.0000
          8 Manoj                Manager                   65000.0000
```

Second Highest Salary is 65000

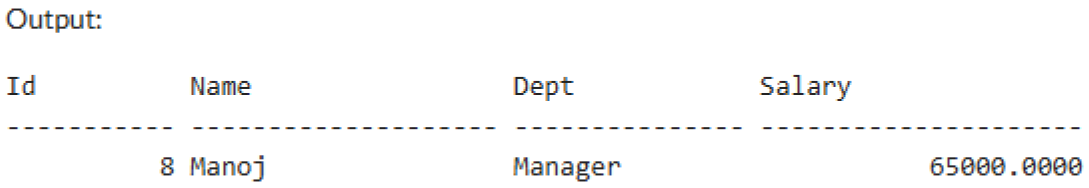
With CTE as (

Select *,rank()over(order by Salary desc) R From Employee

)

Select Id,Name,Dept,Salary From CTE

Where R = 2



Output:

```
Id          Name                 Dept           Salary
----------- -------------------- -------------- ---------------------
          8 Manoj                Manager                   65000.0000
```

2. What are subqueries and how do you use them?

Subqueries in SQL

A **subquery** is a query nested inside another query. It is used to perform operations that require a result set to be passed to the outer query. Subqueries are enclosed in parentheses and can be placed in various parts of an SQL statement such as the SELECT, FROM, WHERE, or HAVING clauses.

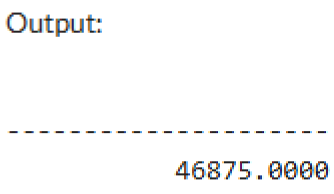
Types of Subqueries

1. **Single-row subqueries**: Return one row with one column.
2. **Multi-row subqueries**: Return multiple rows with a single column.
3. **Multi-column subqueries**: Return multiple rows and multiple columns.
4. **Correlated subqueries**: Refer to columns in the outer query and are evaluated once for each row processed by the outer query.

Example:

Average salary is

Select avg(Salary)From Employee

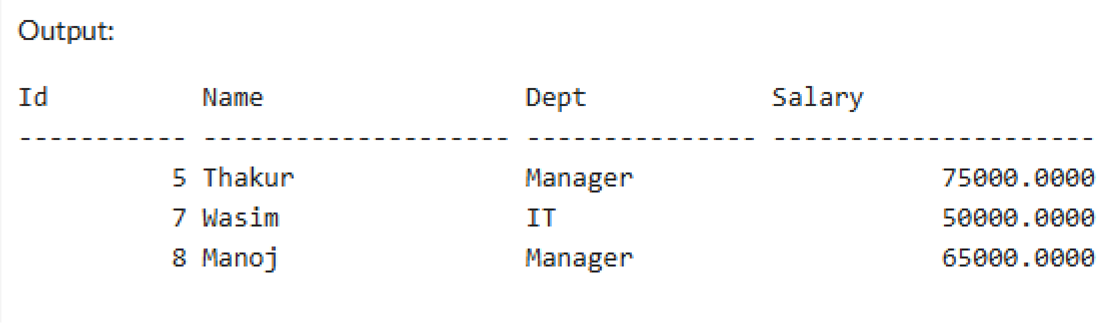


```
Output:


---------------------
           46875.0000
```

I want all the records from the Employee table whose salary is highest than the average salary.

Select * From Employee Where Salary >=

(Select avg(Salary) From Employee)



```
Output:

Id          Name                 Dept            Salary
----------- -------------------- --------------- ---------------------
          5 Thakur               Manager                    75000.0000
          7 Wasim                IT                         50000.0000
          8 Manoj                Manager                    65000.0000
```

3. What is a Common Table Expression (CTE)? Give an example of when to use it.

A **Common Table Expression (CTE)** is a temporary result set that you can reference within a SELECT, INSERT, UPDATE, or DELETE statement. It is defined using the WITH keyword and exists only for the duration of the query.

CTEs are helpful for:

1. Simplifying complex queries by breaking them into smaller, logical parts.

2. Improving readability and maintainability.
3. Using recursive queries, such as traversing hierarchical data.

Syntax of a CTE

```
WITH CTE_Name AS (
    -- Define the CTE query
    SELECT columns
    FROM table
    WHERE condition
)
-- Use the CTE in a query
SELECT *
FROM CTE_Name;
```

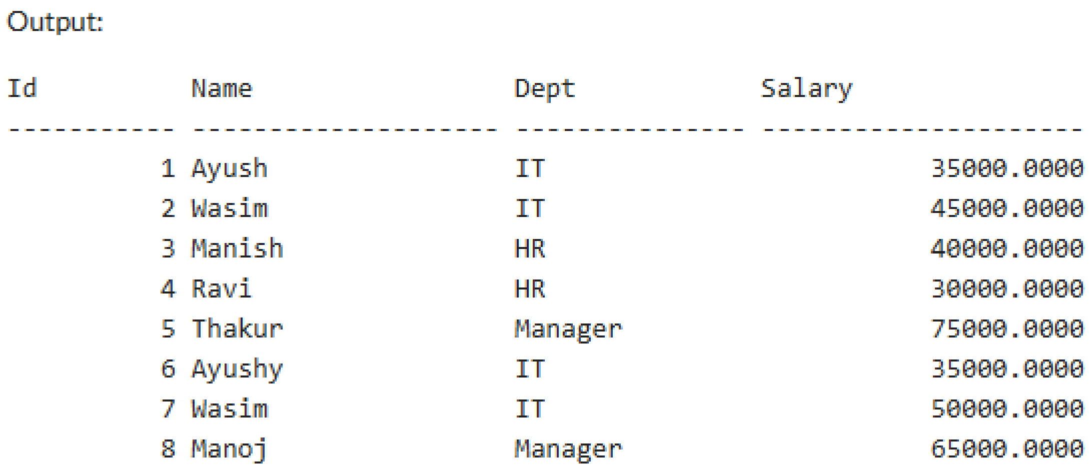
Example: Find the Second Lowest Salary

Using a CTE to find the second lowest salary ensures the query is clean and readable.

Table:

```
Create table Employee(Id int identity(1,1), Name varchar(20),
Dept varchar(15) Check (Dept in ("IT","HR","Manager")), Salary Money);

Insert into Employee Values("Ayush","IT",35000),("Wasim","IT",45000),
("Manish","HR",40000),("Ravi","HR",30000),("Thakur","Manager",75000),
("Ayushy","IT",35000),("Wasim","IT",50000),("Manoj","Manager",65000);
```

Select * From Employee



```
Output:

Id          Name                 Dept            Salary
----------- -------------------- --------------- ---------------------
          1 Ayush                IT                         35000.0000
          2 Wasim                IT                         45000.0000
          3 Manish               HR                         40000.0000
          4 Ravi                 HR                         30000.0000
          5 Thakur               Manager                    75000.0000
          6 Ayushy               IT                         35000.0000
          7 Wasim                IT                         50000.0000
          8 Manoj                Manager                    65000.0000
```

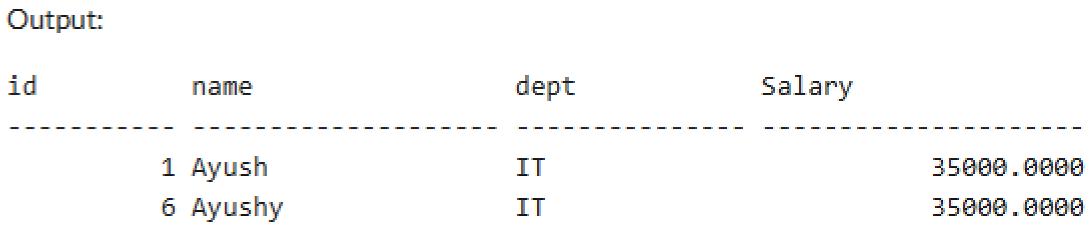
Find the Second Lowest Salary

With CTE as (

Select *,dense_rank()over(order by Salary) R From Employee

)

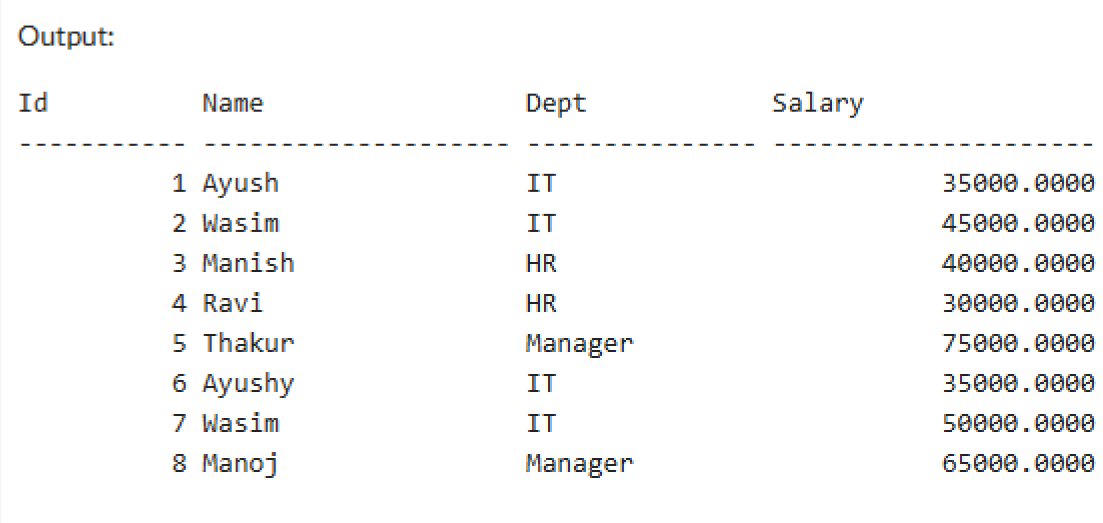
Select id,name,dept,Salary from CTE where R = 2



```
Output:

id          name                 dept            Salary
----------- -------------------- --------------- ---------------------
          1 Ayush                IT                         35000.0000
          6 Ayushy               IT                         35000.0000
```

4. Explain window functions like `ROW_NUMBER()`, `RANK()`, and `DENSE_RANK()`.

Table: Employee

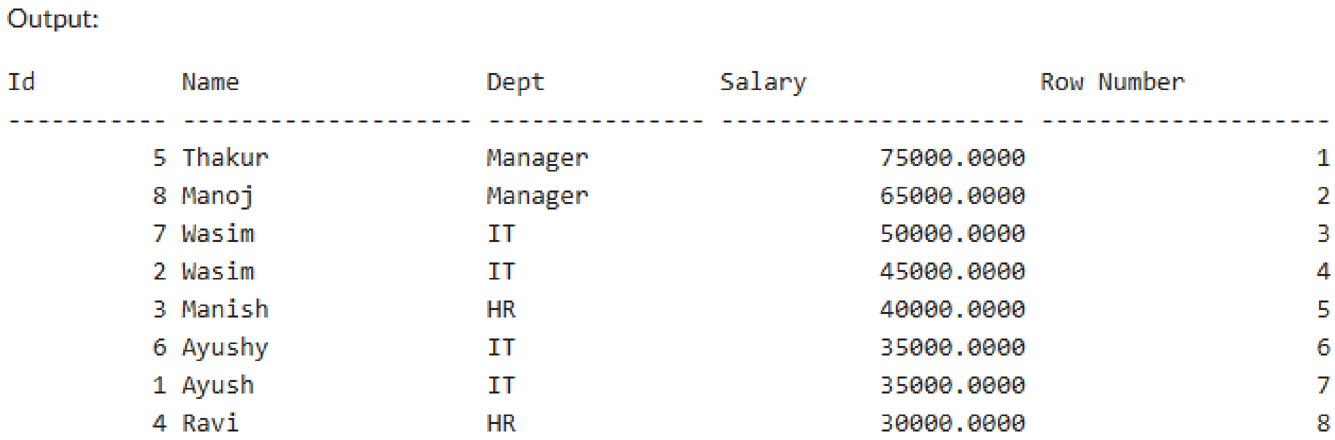


```
Output:

Id          Name                 Dept            Salary
----------- -------------------- --------------- ---------------------
          1 Ayush                IT                         35000.0000
          2 Wasim                IT                         45000.0000
          3 Manish               HR                         40000.0000
          4 Ravi                 HR                         30000.0000
          5 Thakur               Manager                    75000.0000
          6 Ayushy               IT                         35000.0000
          7 Wasim                IT                         50000.0000
          8 Manoj                Manager                    65000.0000
```

1. ROW_NUMBER()

- Assigns a **unique sequential number** to each row within a partition of the result set.
- No ties: Each row gets a unique number, even if values are identical.
- Example:

  Select *,Row_number()over(Order by Salary Desc) [Row Number] From Employee

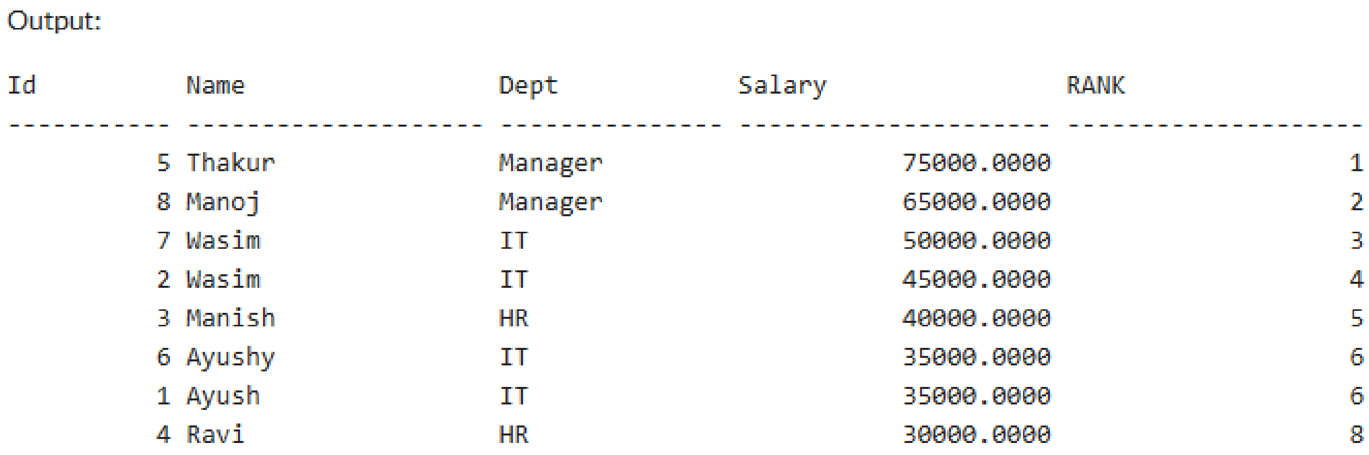


```
Output:

Id          Name                 Dept            Salary                Row Number
----------- -------------------- --------------- --------------------- --------------------
          5 Thakur               Manager                    75000.0000                    1
          8 Manoj                Manager                    65000.0000                    2
          7 Wasim                IT                         50000.0000                    3
          2 Wasim                IT                         45000.0000                    4
          3 Manish               HR                         40000.0000                    5
          6 Ayushy               IT                         35000.0000                    6
          1 Ayush                IT                         35000.0000                    7
          4 Ravi                 HR                         30000.0000                    8
```

2. RANK()

- Assigns a **rank to rows** within a partition of the result set based on the ORDER BY clause.
- **Ties**: Rows with the same values get the same rank, but the next rank skips numbers.
- Example:

Select *,RANK()over(Order by Salary Desc) [RANK] From Employee

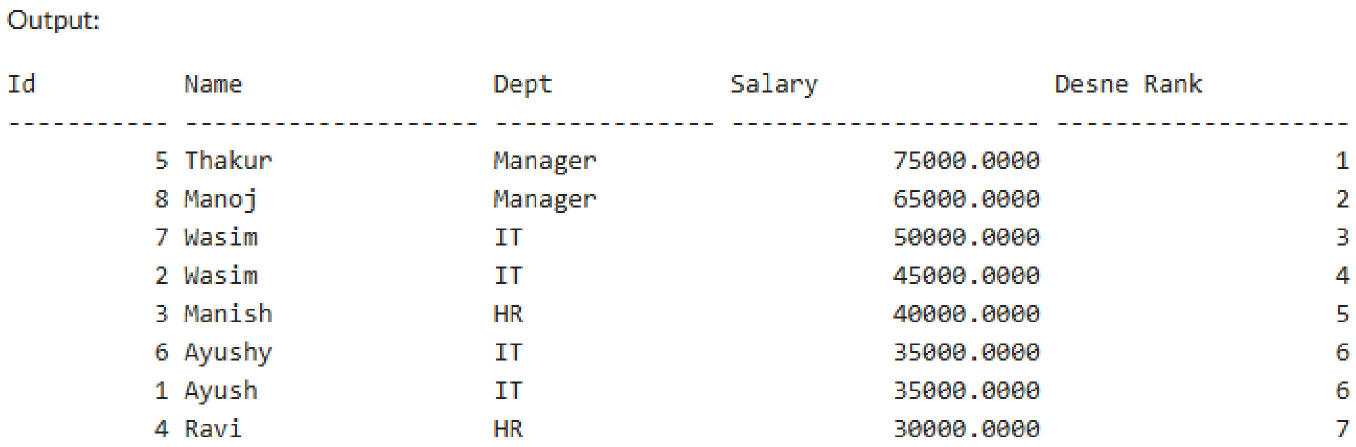


```
Output:

Id          Name                 Dept           Salary               RANK
----------- -------------------- -------------- -------------------- --------------------
          5 Thakur               Manager                  75000.0000                    1
          8 Manoj                Manager                  65000.0000                    2
          7 Wasim                IT                       50000.0000                    3
          2 Wasim                IT                       45000.0000                    4
          3 Manish               HR                       40000.0000                    5
          6 Ayushy               IT                       35000.0000                    6
          1 Ayush                IT                       35000.0000                    6
          4 Ravi                 HR                       30000.0000                    8
```

3. DENSE_RANK()

- Similar to RANK(), but **does not skip ranks** after ties.
- Example:

Select *,Dense_rank()over(Order by Salary Desc) [Desne Rank] From Employee



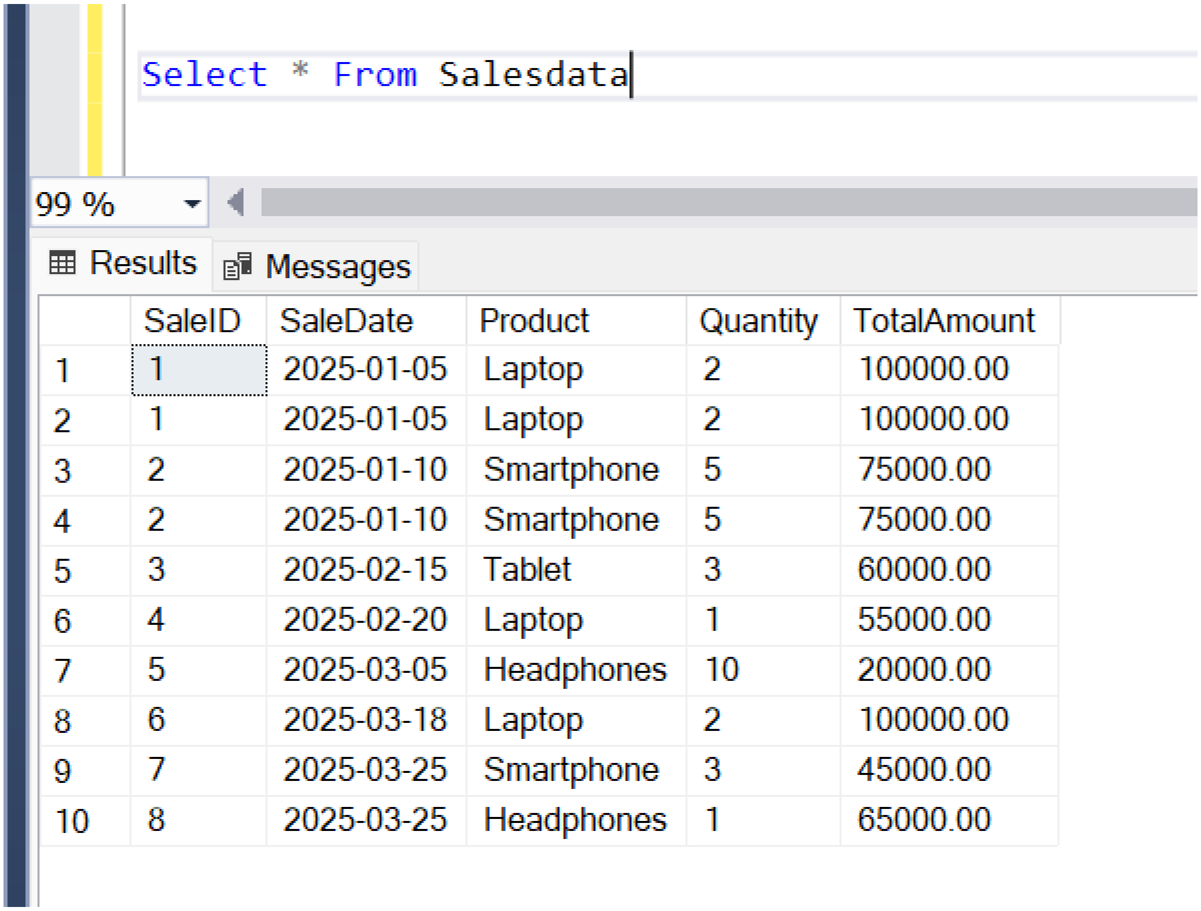
```
Output:

Id          Name                 Dept           Salary               Desne Rank
----------- -------------------- -------------- -------------------- --------------------
          5 Thakur               Manager                  75000.0000                    1
          8 Manoj                Manager                  65000.0000                    2
          7 Wasim                IT                       50000.0000                    3
          2 Wasim                IT                       45000.0000                    4
          3 Manish               HR                       40000.0000                    5
          6 Ayushy               IT                       35000.0000                    6
          1 Ayush                IT                       35000.0000                    6
          4 Ravi                 HR                       30000.0000                    7
```

Use Cases

1. **ROW_NUMBER()**: When you need unique numbering for rows, such as removing duplicates or paginating results.
2. **RANK()**: When you need ranked data and care about skipped ranks for ties, such as in sports rankings.
3. **DENSE_RANK()**: When you need ranked data but cannot afford skipped ranks, such as ranking employees by salary tiers.

5. How do you combine results of two queries using `UNION` and `UNION ALL`?

(Practiced in SQL Server)



```
Select * From Salesdata
```

	SaleID	SaleDate	Product	Quantity	TotalAmount
1	1	2025-01-05	Laptop	2	100000.00
2	1	2025-01-05	Laptop	2	100000.00
3	2	2025-01-10	Smartphone	5	75000.00
4	2	2025-01-10	Smartphone	5	75000.00
5	3	2025-02-15	Tablet	3	60000.00
6	4	2025-02-20	Laptop	1	55000.00
7	5	2025-03-05	Headphones	10	20000.00
8	6	2025-03-18	Laptop	2	100000.00
9	7	2025-03-25	Smartphone	3	45000.00
10	8	2025-03-25	Headphones	1	65000.00

SalesID 1 and 2 have duplicate entries.

Difference Between UNION and UNION ALL

Both UNION and UNION ALL are used to combine the results of two or more SELECT queries into a single result set, but they behave differently in terms of handling duplicate rows.

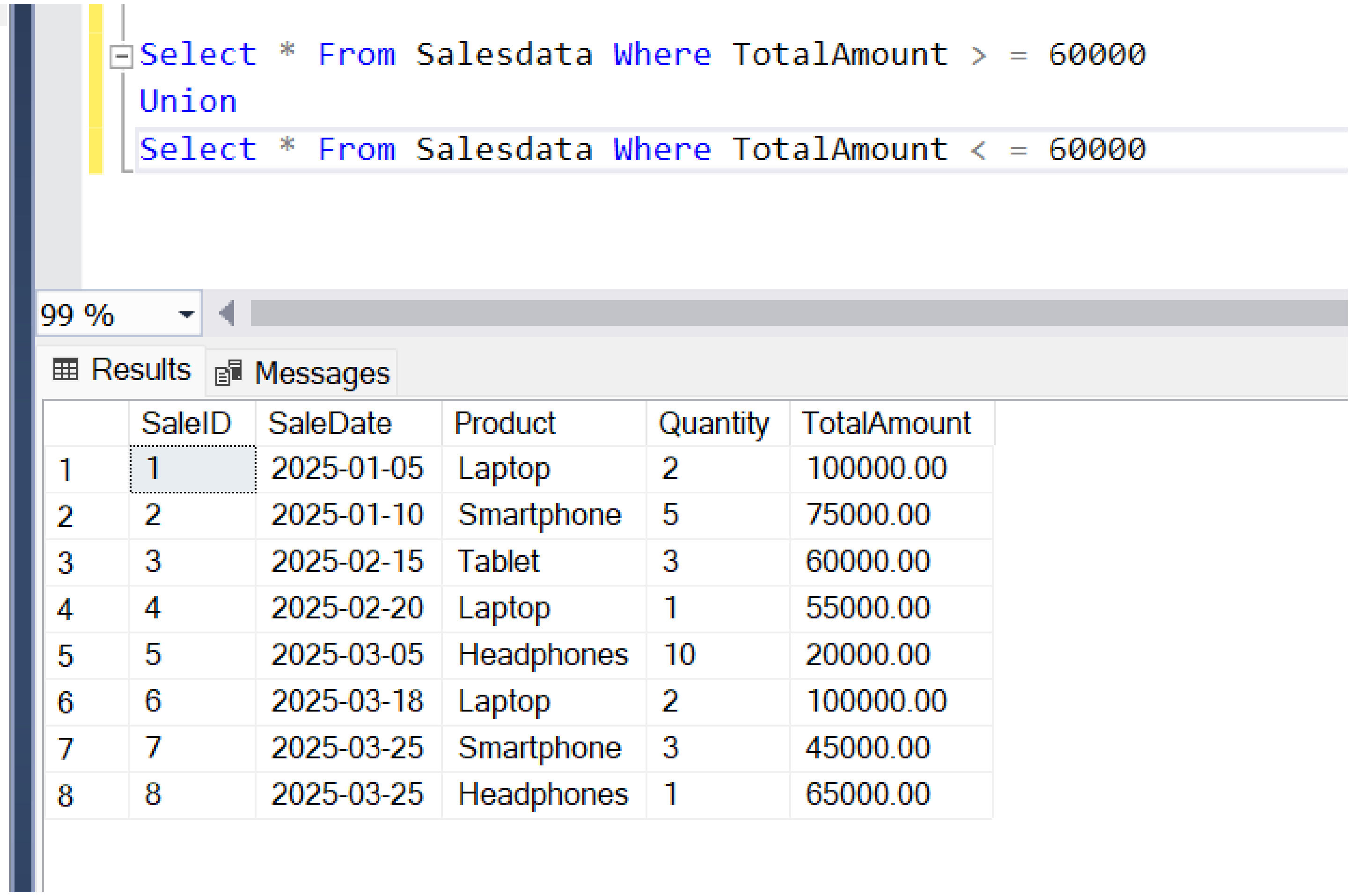
---

1. UNION:

- **Removes Duplicates**:
  UNION combines the results of two or more SELECT statements but **removes duplicate rows** from the final result set.

- **Sorting**:
  Behind the scenes, SQL Server performs a **sort operation** to eliminate duplicates, which can make UNION slower than UNION ALL, especially with large datasets.

- **Use Case**:
  Use UNION when you want to merge results from multiple queries and ensure that each row in the result set is unique.



```
Select * From Salesdata Where TotalAmount > = 60000
Union
Select * From Salesdata Where TotalAmount < = 60000
```

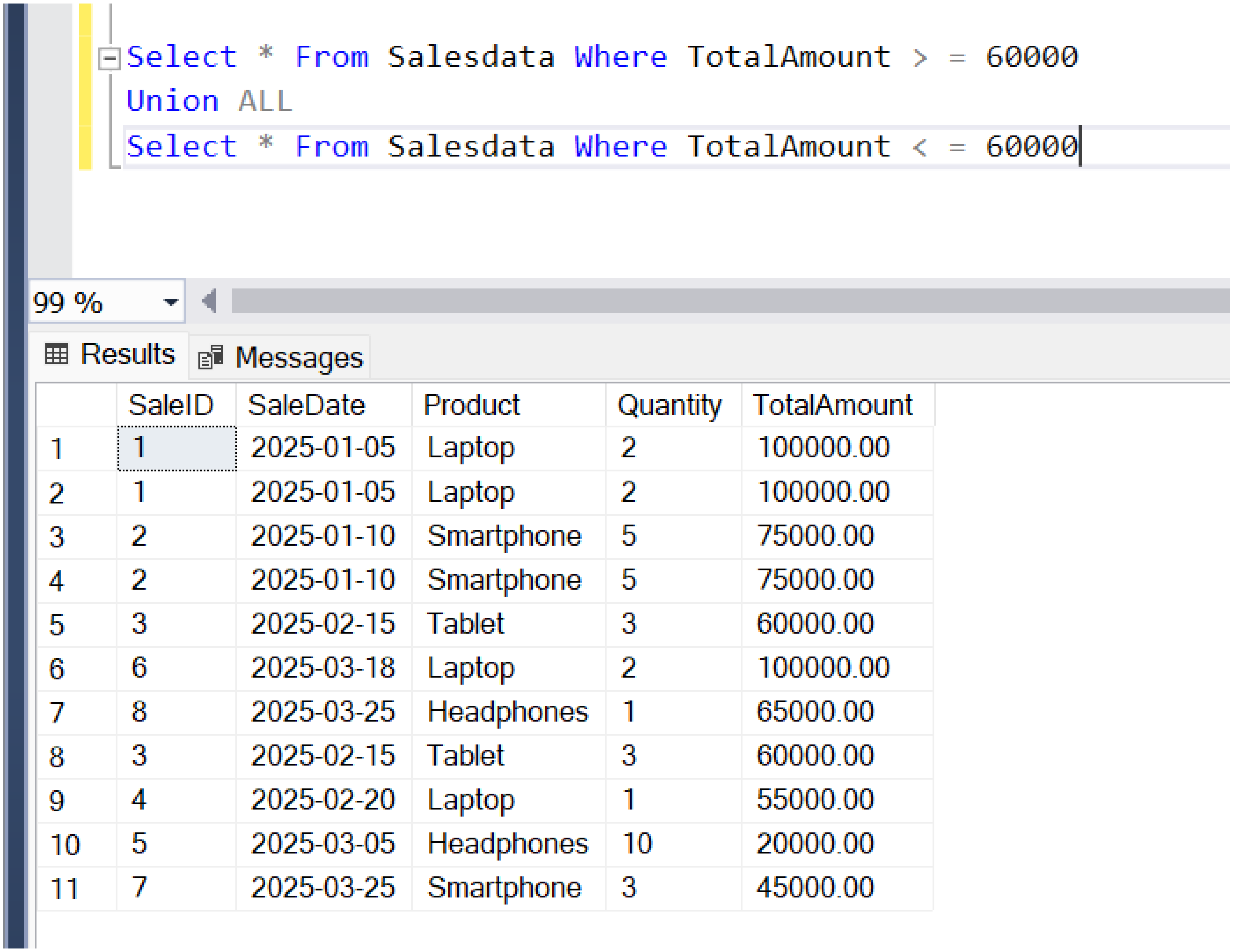
	SaleID	SaleDate	Product	Quantity	TotalAmount
1	1	2025-01-05	Laptop	2	100000.00
2	2	2025-01-10	Smartphone	5	75000.00
3	3	2025-02-15	Tablet	3	60000.00
4	4	2025-02-20	Laptop	1	55000.00
5	5	2025-03-05	Headphones	10	20000.00
6	6	2025-03-18	Laptop	2	100000.00
7	7	2025-03-25	Smartphone	3	45000.00
8	8	2025-03-25	Headphones	1	65000.00

2. UNION ALL:

- **Includes Duplicates**:
  UNION ALL combines the results of two or more SELECT statements but **includes all rows, even duplicates**.

- **Faster Performance**:
  Since UNION ALL does not require the removal of duplicates, it is generally faster than UNION for large datasets.

- **Use Case**:
  Use UNION ALL when you want to include all results from multiple queries, even if there are duplicates, and you need better performance.



```
Select * From Salesdata Where TotalAmount > = 60000
Union ALL
Select * From Salesdata Where TotalAmount < = 60000
```

	SaleID	SaleDate	Product	Quantity	TotalAmount
1	1	2025-01-05	Laptop	2	100000.00
2	1	2025-01-05	Laptop	2	100000.00
3	2	2025-01-10	Smartphone	5	75000.00
4	2	2025-01-10	Smartphone	5	75000.00
5	3	2025-02-15	Tablet	3	60000.00
6	6	2025-03-18	Laptop	2	100000.00
7	8	2025-03-25	Headphones	1	65000.00
8	3	2025-02-15	Tablet	3	60000.00
9	4	2025-02-20	Laptop	1	55000.00
10	5	2025-03-05	Headphones	10	20000.00
11	7	2025-03-25	Smartphone	3	45000.00

6. What are indexes in SQL, and how do they improve query performance?

Types of Indexes in SQL

1. **Clustered Index:**
    - **The data in the table is physically arranged on disk according to the clustered index.**
    - **A table can have only one clustered index because data rows can only be sorted one way.**
    - **By default, the primary key of a table creates a clustered index.**
2. **Non-Clustered Index:**

   - **A separate structure from the data table that holds the indexed columns and pointers to the actual data.**

   - **A table can have multiple non-clustered indexes.**

   - **Non-clustered indexes are typically used for columns that are frequently searched or queried.**

3. **Unique Index:**

   - **Ensures that the values in the indexed columns are unique, such as for columns with unique constraints like the primary key or unique key.**

4. **Composite Index:**

   - **An index that involves multiple columns. It's useful when queries filter or sort based on multiple columns.**

5. **Full-text Index:**

   - **Used for text searching in large text columns, such as VARCHAR or TEXT. It allows for faster searching of words or phrases within the text.**

6. **Spatial Index:**

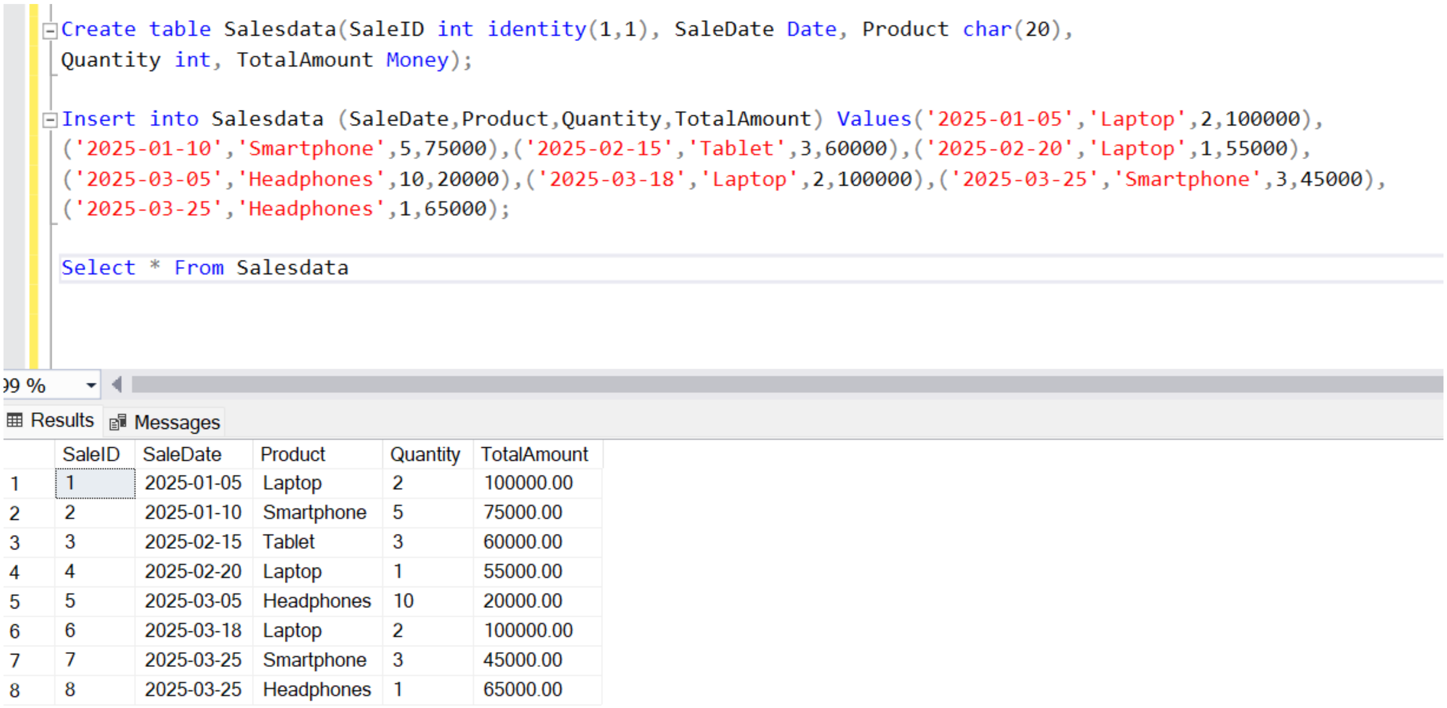
   - **Used for spatial data types (e.g., geometry, geography) to perform fast location-based queries.**

Creating an Index

Here's an example of creating a non-clustered index in SQL Server:

- **Create index in_tam on Salesdata (TotalAmount Desc)**

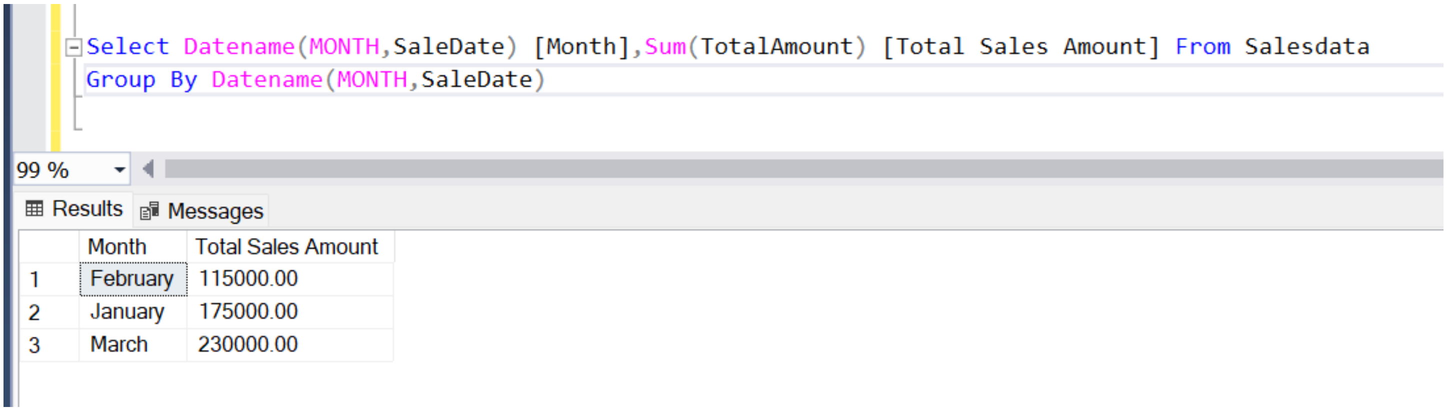
7. Write a query to calculate the total sales for each month using `GROUP BY`. (Practiced in SQL Server)



```sql
Create table Salesdata(SaleID int identity(1,1), SaleDate Date, Product char(20),
Quantity int, TotalAmount Money);

Insert into Salesdata (SaleDate,Product,Quantity,TotalAmount) Values('2025-01-05','Laptop',2,100000),
('2025-01-10','Smartphone',5,75000),('2025-02-15','Tablet',3,60000),('2025-02-20','Laptop',1,55000),
('2025-03-05','Headphones',10,20000),('2025-03-18','Laptop',2,100000),('2025-03-25','Smartphone',3,45000),
('2025-03-25','Headphones',1,65000);

Select * From Salesdata
```

	SaleID	SaleDate	Product	Quantity	TotalAmount
1	1	2025-01-05	Laptop	2	100000.00
2	2	2025-01-10	Smartphone	5	75000.00
3	3	2025-02-15	Tablet	3	60000.00
4	4	2025-02-20	Laptop	1	55000.00
5	5	2025-03-05	Headphones	10	20000.00
6	6	2025-03-18	Laptop	2	100000.00
7	7	2025-03-25	Smartphone	3	45000.00
8	8	2025-03-25	Headphones	1	65000.00

Answer:

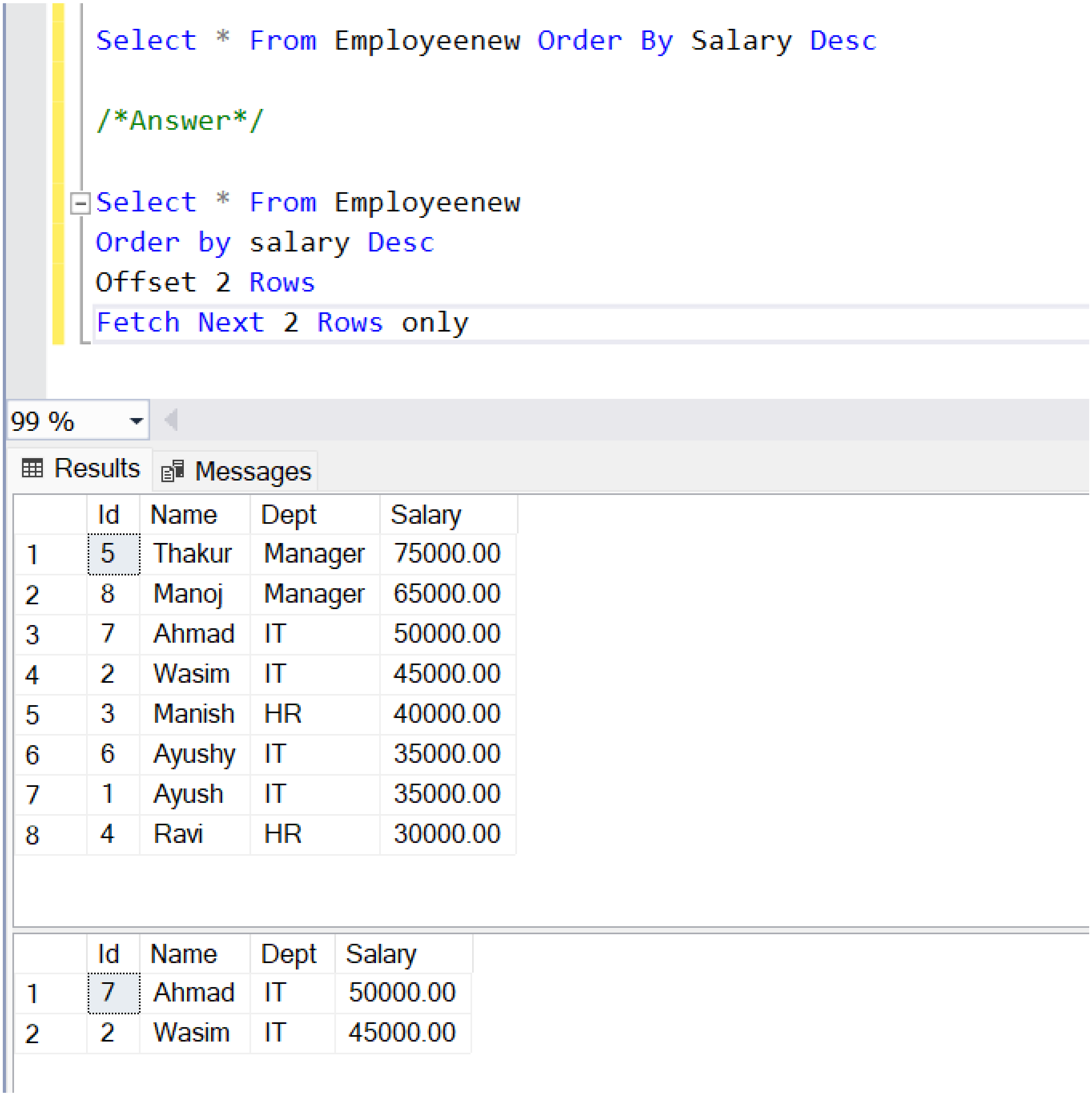


```sql
Select Datename(MONTH,SaleDate) [Month],Sum(TotalAmount) [Total Sales Amount] From Salesdata
Group By Datename(MONTH,SaleDate)
```

	Month	Total Sales Amount
1	February	115000.00
2	January	175000.00
3	March	230000.00

Advanced SQL:

1. How can you retrieve all employees sorted by their salary in descending order, and then fetch the 3rd and 4th highest-paid employees using SQL? (Practiced in SQL Server)



```sql
Select * From Employeenew Order By Salary Desc

/*Answer*/

Select * From Employeenew
Order by salary Desc
Offset 2 Rows
Fetch Next 2 Rows only
```

	Id	Name	Dept	Salary
1	5	Thakur	Manager	75000.00
2	8	Manoj	Manager	65000.00
3	7	Ahmad	IT	50000.00
4	2	Wasim	IT	45000.00
5	3	Manish	HR	40000.00
6	6	Ayushy	IT	35000.00
7	1	Ayush	IT	35000.00
8	4	Ravi	HR	30000.00

	Id	Name	Dept	Salary
1	7	Ahmad	IT	50000.00
2	2	Wasim	IT	45000.00

2. What are views in SQL, and when would you use them? (Practiced in SQL Server)

What are Views?

A **view** in SQL is a virtual table that is based on the result of a SELECT query. It does not store data itself but retrieves data dynamically from the underlying base tables whenever it is queried. Views help simplify complex queries, provide data abstraction, and enhance security by exposing only specific data to users.

Why Use Views?

- **Simplify Queries:** Encapsulate complex SELECT statements for reuse.

- **Security:** Restrict access to sensitive data by exposing only specific columns or rows.
- **Data Abstraction:** Hide the underlying database schema complexity.
- **Reusability:** Create a central query logic that can be reused without rewriting.
- **Logical Independence:** Adjust the view definition without altering dependent queries.

Example:

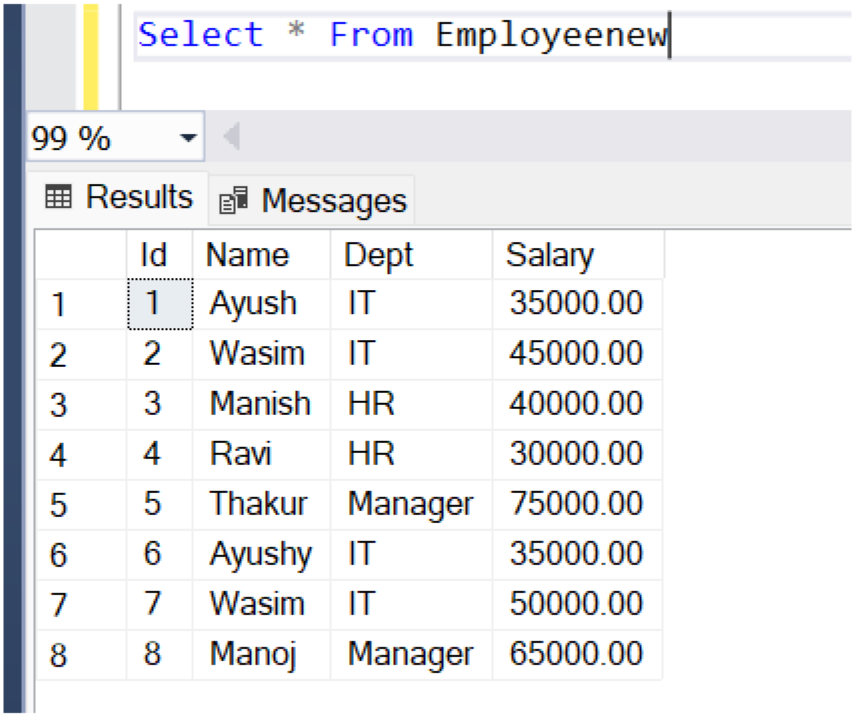
Create table Employeenew(Id int identity(1,1), Name varchar(20),

Dept varchar(15) Check (Dept in ('IT','HR','Manager')), Salary Money);

Insert into Employeenew (Name,Dept,salary) Values('Ayush','IT',35000),('Wasim','IT',45000),

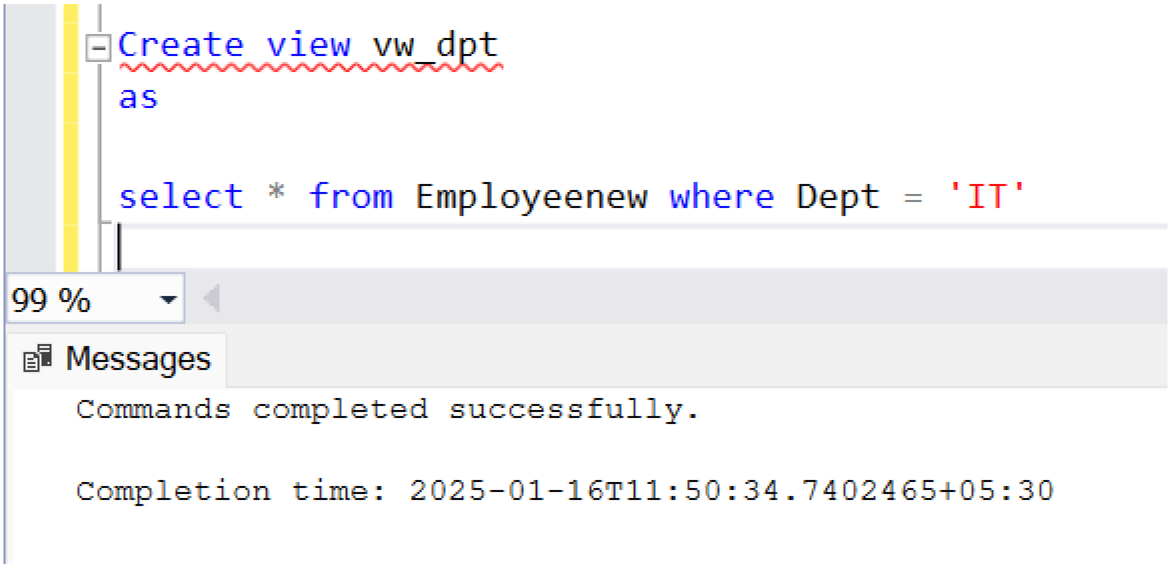
('Manish','HR',40000),('Ravi','HR',30000),('Thakur','Manager',75000),

('Ayushy','IT',35000),('Wasim','IT',50000),('Manoj','Manager',65000);



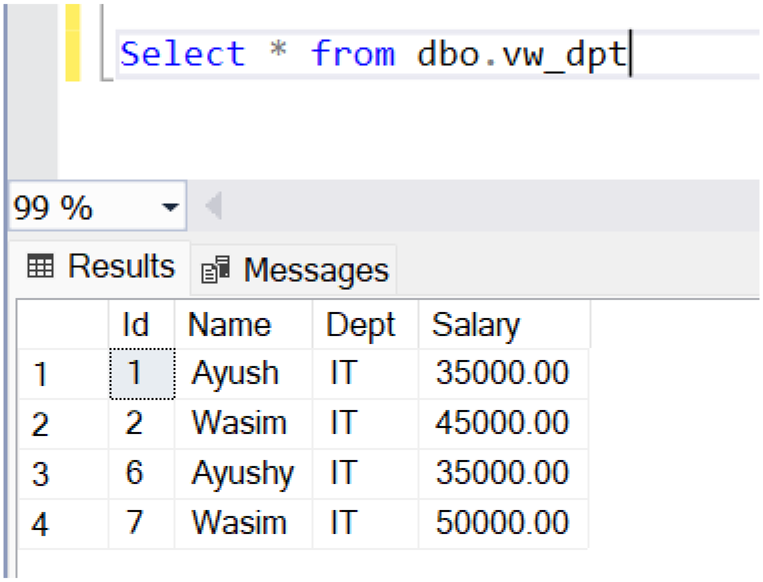
```
Select * From Employeenew
```

	Id	Name	Dept	Salary
1	1	Ayush	IT	35000.00
2	2	Wasim	IT	45000.00
3	3	Manish	HR	40000.00
4	4	Ravi	HR	30000.00
5	5	Thakur	Manager	75000.00
6	6	Ayushy	IT	35000.00
7	7	Wasim	IT	50000.00
8	8	Manoj	Manager	65000.00



```sql
Create view vw_dpt
as

select * from Employeenew where Dept = 'IT'
```

```
Commands completed successfully.

Completion time: 2025-01-16T11:50:34.7402465+05:30
```



```sql
Select * from dbo.vw_dpt
```

	Id	Name	Dept	Salary
1	1	Ayush	IT	35000.00
2	2	Wasim	IT	45000.00
3	6	Ayushy	IT	35000.00
4	7	Wasim	IT	50000.00

In this case, the view vw_dpt simplifies the task of retrieving data for employees in the "IT" department without rewriting the WHERE condition repeatedly.

3. What is the difference between a stored procedure and a function in SQL?

Feature	**Stored Procedure**	**Function**
Return Type	No return value (can return result sets via SELECT)	Returns a single value or table
Usage in SQL	Executed via EXEC or CALL	Used directly in queries (e.g., SELECT)
Side Effects	Can modify data, perform transactions, etc.	Cannot modify data or perform side effects
Transaction Control	Can manage transactions (BEGIN, COMMIT, ROLLBACK)	Cannot manage transactions

Error Handling	Supports TRY...CATCH for error handling	Limited error handling, cannot catch exceptions
Performance	Can be optimized for complex operations	Scalar functions may degrade performance in large queries
Parameters	Input, output, and input-output parameters	Only input parameters

4. Explain the difference between `TRUNCATE`, `DELETE`, and `DROP` commands.

1. DELETE Command

The DELETE command is used to remove specific rows from a table based on a condition. It can be used with a WHERE clause to delete specific records, or if the WHERE clause is omitted, it will delete all rows from the table.

- **Behavior**:
  - Row-by-row deletion: Deletes records one at a time.
  - Can be rolled back: Since DELETE is a logged operation, it can be rolled back within a transaction.
  - Can be used with WHERE to delete specific rows.
  - Triggers are fired: If there are any triggers on the table, they will be activated when DELETE is used.
- **Example**:

```
Delete From Employee
```

```
 Delete From Employee Where Dept = "IT"
```

- **When to use**:
- When you want to delete specific rows from a table.
- When you need to control the deletion process (e.g., filtering with WHERE).

2. TRUNCATE Command

The TRUNCATE command is used to delete **all rows** from a table, but it does so **more efficiently** than DELETE. It is generally faster because it does not log individual row deletions, and it does not fire any triggers.

- **Behavior**:
  - Deletes all rows: It removes all rows from the table, not just specific ones.

    - Cannot be rolled back in all databases: TRUNCATE is often a **non-logged** operation, meaning you can't easily roll it back (though in some systems like SQL Server, you can rollback if it's part of a transaction).
    - Resets identity: If the table has an IDENTITY column (auto-incrementing column), TRUNCATE will reset the counter.
    - Faster than DELETE: Because it does not log individual row deletions and does not fire triggers, it is generally much faster for large tables.
- **Example**:

Truncate table Employee

When to use:

- When you need to delete all rows from a table quickly.
- When you don't need to worry about transaction logs or triggers.

3. DROP Command

The DROP command is used to remove an entire table (or database, view, index, etc.) from the database permanently. It completely deletes the table structure and all its data. Once a table is dropped, it cannot be recovered unless there's a backup.

- **Behavior**:
    - Deletes the table structure: Removes both the data and the definition of the table (i.e., the schema).
    - Cannot be rolled back: Unlike DELETE and TRUNCATE, DROP is typically a permanent operation (though some database systems allow recovery through backups).
    - No triggers or constraints: Since the table itself is removed, triggers, constraints, indexes, and all associated objects are also dropped.
- **Example**:

Drop table Employee

- **When to use**:
    - When you want to completely remove a table and its structure from the database.
    - When you no longer need the table and want to free up resources.

5. What are windowing functions, and how are they used in analytics?

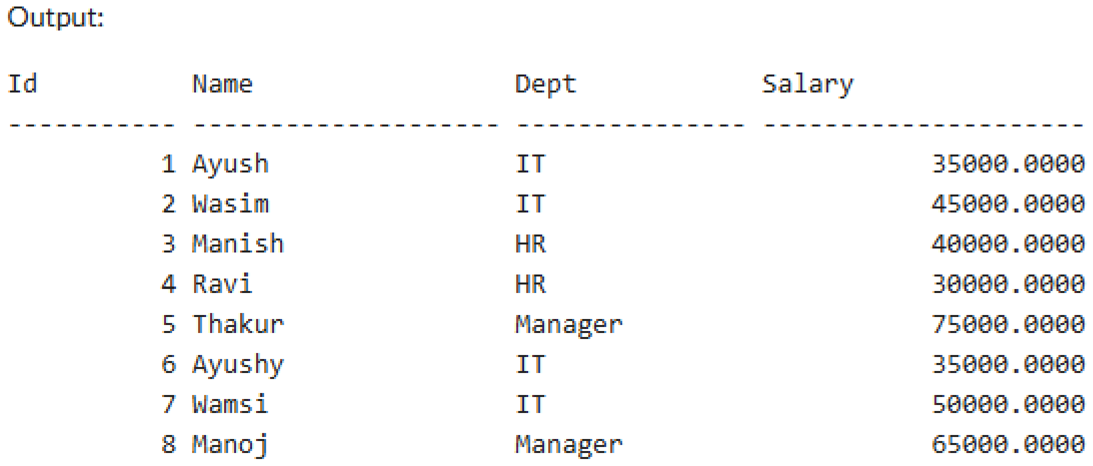
Window functions (also known as **analytic functions**) are a set of functions in SQL that allow you to perform calculations across a set of table rows related to the current row. Unlike regular aggregate functions (like SUM(), COUNT(), etc.), which return a single result for a group of rows, window functions return a value for each row in the result set while still considering other rows in the table for calculations.

Key Characteristics of Window Functions:

1. **Operate over a "window" of rows**: A window function operates on a subset of rows that are related to the current row, defined by a PARTITION BY clause.
2. **Does not collapse rows**: Unlike aggregate functions that group rows into a single result, window functions keep the original row structure intact.
3. **Allow partitioning and ordering**: You can specify partitions (subgroups) and ordering within those partitions, which allows for more advanced analytics.

6. How do you use `PARTITION BY` and `ORDER BY` in window functions?

Table: Employee



Output:

```
Id          Name                 Dept            Salary
----------- -------------------- --------------- ---------------------
          1 Ayush                IT                         35000.0000
          2 Wasim                IT                         45000.0000
          3 Manish               HR                         40000.0000
          4 Ravi                 HR                         30000.0000
          5 Thakur               Manager                    75000.0000
          6 Ayushy               IT                         35000.0000
          7 Wamsi                IT                         50000.0000
          8 Manoj                Manager                    65000.0000
```

Find the Names with the top salary in each Department.

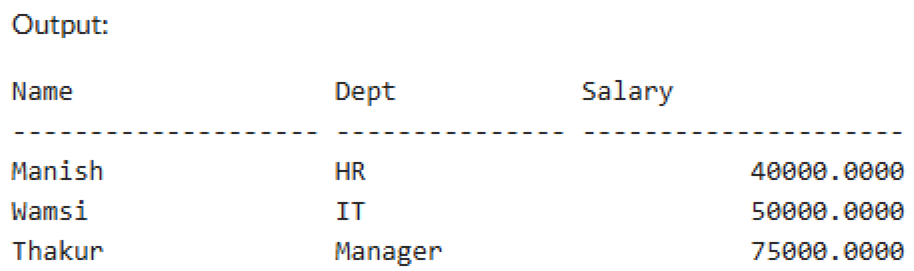
Select Name,Dept,Salary From (

Select *,

row_number()over(partition by dept order by Salary desc) R From Employee

)Sub

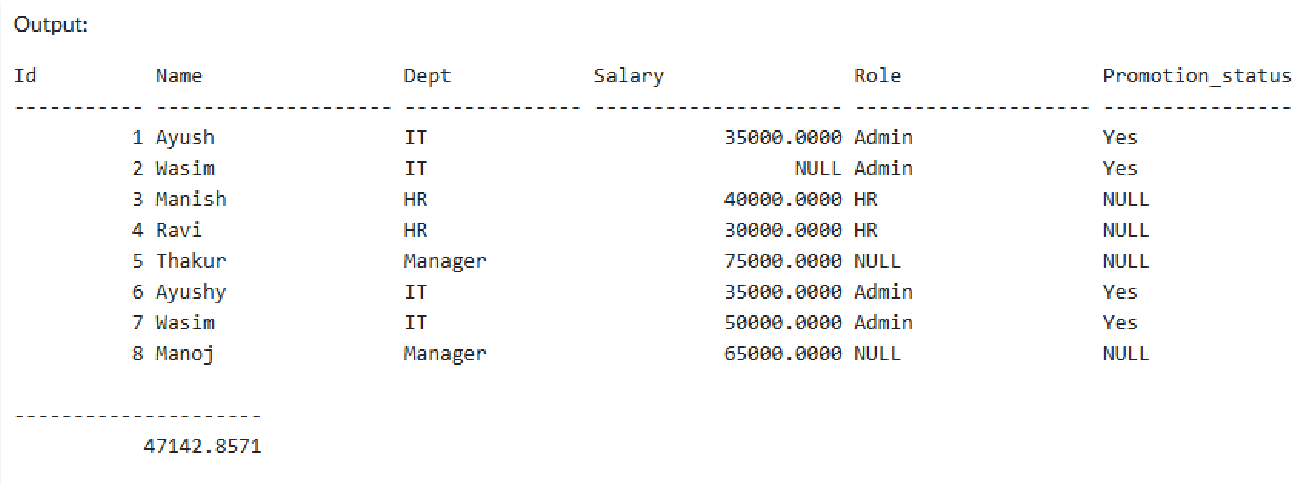
Where R = 1



```
Output:

Name                 Dept            Salary
-------------------- --------------- ---------------------
Manish               HR                         40000.0000
Wamsi                IT                         50000.0000
Thakur               Manager                    75000.0000
```

7. How do you handle NULL values in SQL, and what functions help with that (e.g., `COALESCE`, `ISNULL`)?

Table: Employee



```
Output:

Id          Name                 Dept            Salary                Role                 Promotion_status
----------- -------------------- --------------- --------------------- -------------------- ----------------
          1 Ayush                IT                         35000.0000 Admin                Yes
          2 Wasim                IT                               NULL Admin                Yes
          3 Manish               HR                         40000.0000 HR                   NULL
          4 Ravi                 HR                         30000.0000 HR                   NULL
          5 Thakur               Manager                    75000.0000 NULL                 NULL
          6 Ayushy               IT                         35000.0000 Admin                Yes
          7 Wasim                IT                         50000.0000 Admin                Yes
          8 Manoj                Manager                    65000.0000 NULL                 NULL

---------------------
           47142.8571
```

Handling NULL values is a common task in SQL. NULL represents missing, undefined, or unknown values in a database, and it requires special handling in queries because it behaves differently from other data types. Below are the main ways to handle NULL values and functions that help with that.

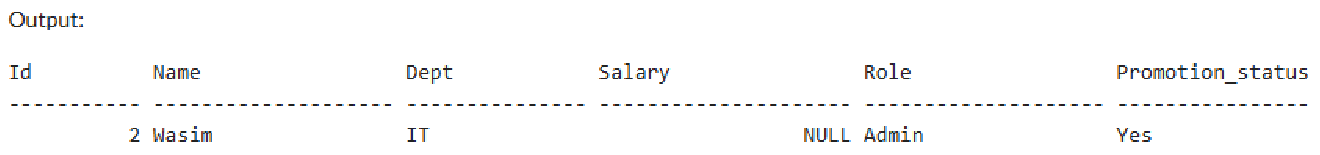
1. Checking for NULL Values

To check if a value is NULL, you can use the following:

- **IS NULL**: Checks if a value is NULL.
- **IS NOT NULL**: Checks if a value is **not** NULL.

Examples:

Select * From Employee Where Salary is Null



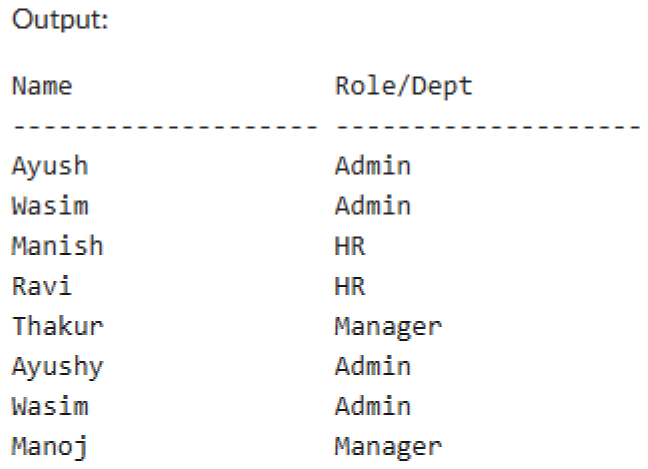
```
Output:

Id         Name                Dept           Salary               Role                Promotion_status
---------- ------------------- -------------- -------------------- ------------------- ----------------
         2 Wasim               IT                             NULL Admin               Yes
```

2. Replacing NULL with a Default Value

- **COALESCE()**: Returns the first non-NULL value in a list of expressions. If all expressions are NULL, it returns NULL.
- **ISNULL()**: Specifically for SQL Server, it replaces NULL with a specified value.

COALESCE() Example:

Select Name,coalesce(Role,Dept) [Role/Dept] From Employee



```
Output:

Name                 Role/Dept
-------------------- --------------------
Ayush                Admin
Wasim                Admin
Manish               HR
Ravi                 HR
Thakur               Manager
Ayushy               Admin
Wasim                Admin
Manoj                Manager
```

Best Practices for Handling NULL:

1. **Use IS NULL or IS NOT NULL** to check for NULL values in queries.
2. **Use COALESCE ()** when you want to replace NULL with a specific value.
3. **Avoid NULL where possible** by setting default values or using NOT NULL constraints if appropriate.