SQL, DB Engine, SQL Services, Ranking Function, Transact SQL (Day 5)

⇒ SQL server has two things:
1- SQL server versions :every year Microsoft add tools or updates.
2- SQL server Editions: some advantages but it is used by paying money (Enterprise(large scale..etc.), Standard developer, BI editions, Express, Azure (for cloud)).

When setting up **SQL Server**, you create an **instance** that includes two main components: **services** and **applications**.

1. Services:

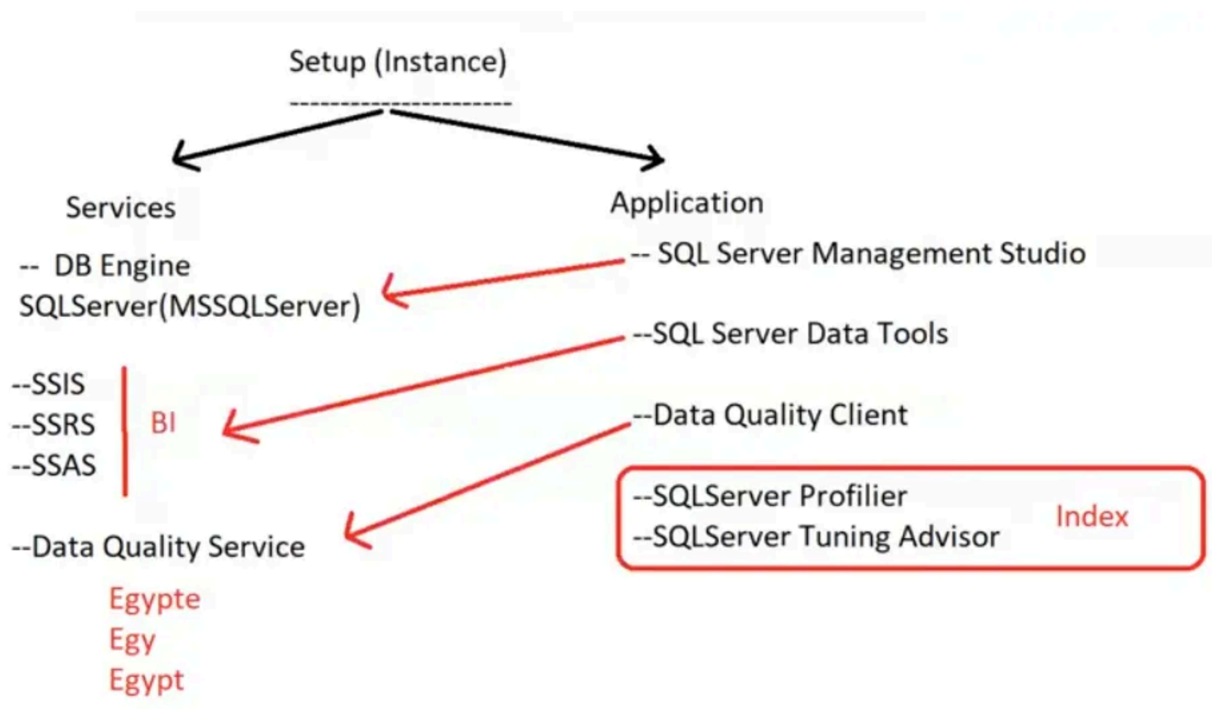
These are background services that run SQL Server's core functions and features, depending on your use case (general database management, business intelligence, etc.).

- **a. Database Engine** (SQLServer/MSSQLSERVER):
    - The core **SQL Server service** that handles **data storage**, **queries**, **transactions**, **security**, and more.
    - This is the main service you'll use for standard database operations.
- **b. Data Quality Services (DQS)**:
    - This service helps ensure that your data is **accurate** and **consistent** by providing **data cleansing** and **matching** tools.
- Services for **Business Intelligence (BI)**:
    - **c. SQL Server Integration Services (SSIS)**:
        - A platform for **data integration** and **workflow automation**.
        - SSIS is used for **ETL processes** (Extract, Transform, Load), helping move and transform data from different sources into SQL Server.
    - **d. SQL Server Reporting Services (SSRS)**:
        - A platform for creating, managing, and delivering **reports**.
        - Allows users to design reports with interactive elements, graphs, and drill-down functionality, which can be viewed on different devices.
    - **e. SQL Server Analysis Services (SSAS)**:
        - Provides **online analytical processing (OLAP)** and **data mining** functionalities.
        - This service allows for building **cubes** for multi-dimensional analysis, making it a powerful tool for advanced data analytics and business intelligence tasks.

2. Applications:

These are client applications or tools that help users interact with SQL Server services. They provide an interface to manage databases, design BI solutions, monitor performance, and more.

- **a. SQL Server Management Studio (SSMS)**:
  - The most commonly used application for connecting to the **Database Engine** and managing databases.
  - Provides a **graphical user interface (GUI)** to perform queries, manage security, configure databases, and perform administrative tasks.
- **b. SQL Server Data Tools (SSDT)**:
  - A development environment used to create **BI solutions** like SSIS packages, SSRS reports, and SSAS cubes.
  - It integrates with **Visual Studio**, allowing developers to design and deploy BI applications.
- **c. Data Quality Client**:
  - This application is used to connect to **Data Quality Services (DQS)** and provides an interface to perform **data cleansing** and **data matching**.
- **d. SQL Server Profiler**:
  - A performance monitoring tool that tracks and logs events happening on the **Database Engine**.
  - It's useful for **diagnosing performance issues**, **debugging queries**, and **monitoring activity** on the server.
- **e. SQL Server Tuning Advisor**:
  - An analysis tool that helps **optimize database performance** by analyzing workloads and suggesting **indexes** or other improvements to increase query efficiency.



⇒ Here are two types of instances you can install:

1. Default Instance:

- This is the first **SQL Server instance** you typically install, and it doesn't require a custom name.
- **Name**: The default instance is named **MSSQLServer**.
- **Connection**: You can connect to this instance using:
  - `.` (dot)
  - `local`
  - `pc-name` (your computer's name)
  - `IP address of the current machine`

  Example connection strings:
  - `localhost`
  - `192.168.1.1`
  - `PC-Name`

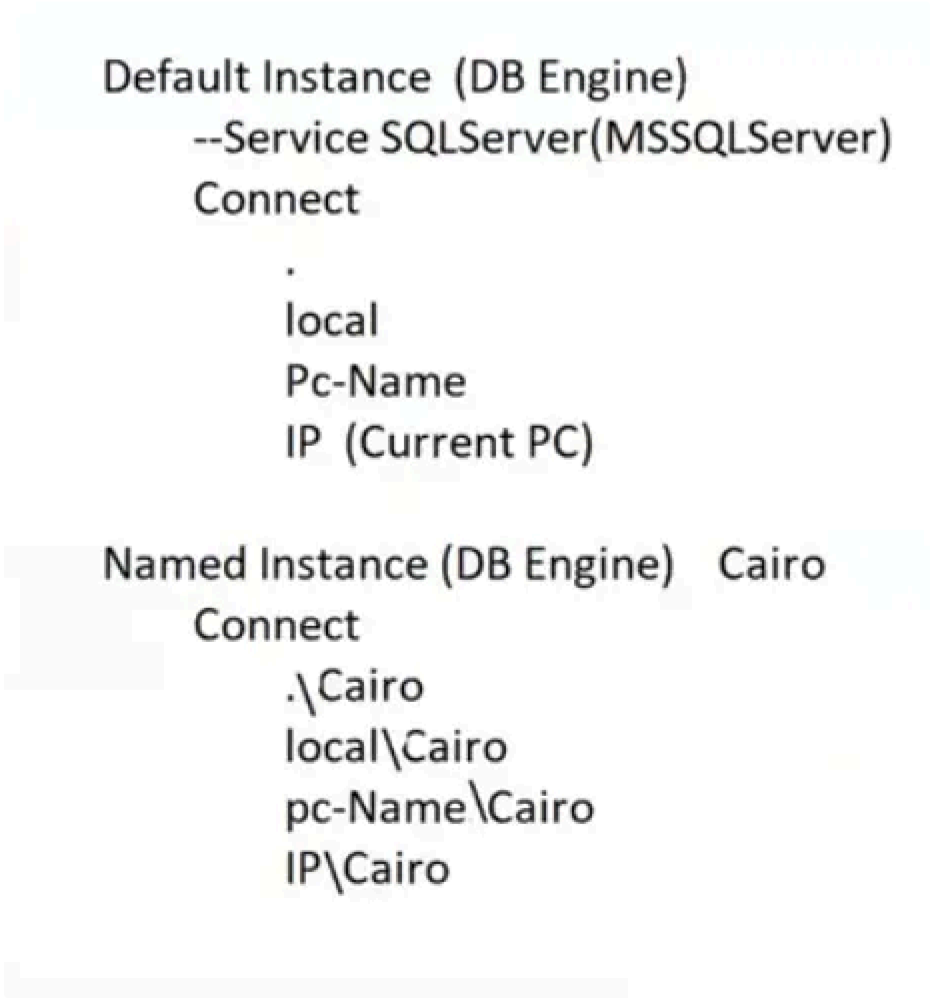
2. Named Instance:

- When you install additional instances of SQL Server in the future, they are **named instances**.
- You can assign a custom name to each new instance, such as **Amr**.
- **Connection**: To connect to a named instance, you need to specify the instance name along with the server. The format for connecting is:
  - `.\\InstanceName` (e.g., `.\\Amr`)
  - `local\\InstanceName` (e.g., `local\\Amr`)
  - `PC-Name\\InstanceName` (e.g., `PC-Name\\Amr`)
  - `IP\\InstanceName` (e.g., `192.168.1.1\\Amr`)

  Example connection strings:
  - `192.168.1.1\\Amr`
  - `PC-Name\\Amr`

Key Points:

- **Multiple Instances**: You can have multiple instances of SQL Server on the same machine. Each instance can contain a combination of services (Database Engine, SSRS, SSIS, etc.).
- **Default vs. Named Instances**: The **default instance** takes the computer's name, while **named instances** are given specific names that you choose during installation.



```
Default Instance  (DB Engine)
        --Service SQLServer(MSSQLServer)
        Connect
                .
                local
                Pc-Name
                IP  (Current PC)

Named Instance (DB Engine)    Cairo
        Connect
                .\Cairo
                local\Cairo
                pc-Name\Cairo
                IP\Cairo
```

In SQL Server, **authorization** and **authentication** play essential roles in securing access to databases.

1. Authentication: Verifying the identity of the user or system trying to access the database.

There are two types of authentication in SQL Server:

a. **Windows Authentication**:

- This mode uses Windows credentials (username and password of a Windows user).
- It is based on the Windows login, meaning if a user is logged into Windows, they can automatically log into SQL Server if they have access.
- **Security Concern**: If someone has access to the Windows machine, they may have access to the database, which can be risky.
- It's generally used for local users who are trusted.

b. **SQL Server Authentication**:

- In this mode, you create a **username** and **password** specifically for SQL Server (for example, `username=Amr`, `password=123`).
- To enable SQL Server Authentication, you must switch the server's authentication mode to **Mixed Mode** (supports both Windows and SQL Server authentication).

Steps to Enable Mixed Mode and Create SQL Server Authentication:

1. **Switch to Mixed Mode**:
    - In **SQL Server Management Studio (SSMS)**, right-click on the SQL Server instance.
    - Select **Properties**.
    - Under the **Security** tab, choose **SQL Server and Windows Authentication Mode**.
    - Click **OK**.
2. **Create a New SQL Server Login**:
    - Open the **Security** folder under the SQL Server instance.
    - Right-click **Logins** and select **New Login**.
    - In the **Login Name**, provide a name (e.g., `Amr`).
    - Choose **SQL Server Authentication** and provide a password (e.g., `123`).
    - Click **OK**.
3. **Access SQL Server Using SQL Server Authentication**:
    - When opening SQL Server, provide the **IP** or **name of the instance** (e.g., `PC-Name\\InstanceName`).
    - Select **SQL Server Authentication** and enter the **username** and **password** created earlier (e.g., `Amr`, `123`).

2. Authorization: Defining what actions the user can perform (permissions).

- After creating a SQL Server login, the user won't automatically have access to databases or tables.

Steps to Grant Authorization:

1. **Grant Database Access**:
    - On the machine that hosts the database, expand the database.
    - Go to **Security** → **Users** → right-click and select **New User**.
    - Add the login you created earlier and map it to the database.
2. **Grant Table or Object Permissions**:
    - After creating a user in the database, they still need permissions to interact with tables, views, or procedures.
    - To do this, navigate to the **Security** folder within the database.
    - Right-click on **Users**, select the user, and grant appropriate permissions (e.g., `SELECT`, `INSERT`, `UPDATE`, etc.).

Key Points:

- **Windows Authentication** is ideal for local, secure environments where users are trusted.
- **SQL Server Authentication** is preferred when sharing databases across networks, as it allows users to access SQL Server from external machines.
- In **Mixed Mode**, you can manage both Windows and SQL Server logins, creating more flexibility.
- **Authorization** (permissions) ensures users can only access specific data or perform certain actions after logging in.

Queries:

1. **Select the top 3 employees from the `Employees` table**:

```
SELECT TOP(3) * FROM Employees;
```

2. **Select the top 3 employees with a salary greater than or equal to 1000**:

```
SELECT TOP(3) * FROM Employees WHERE Salary >= 1000;
```

3. **Select the first names (assuming `Fname` as first name column) of the top 3 employees:**

```sql
SELECT TOP(3) EmployeeName FROM Employees;
```

4. **Select the top 3 salaries from employees, ordered by salary in descending order:**

```sql
SELECT TOP(3) Salary FROM Employees ORDER BY Salary DESC;
```

5. **Select the top 4 rows with ties, based on salary:**

```sql
SELECT TOP(4) WITH TIES * FROM Employees ORDER BY Salary;
```

   This query will select the top 4 rows and will include any additional rows if there is a tie in salary for the 4th value.

6. **Select a random new ID (NEWID):**

```sql
SELECT NEWID(); --GUID (Globally Unique Identifier)
```

7. **Select all columns and assign a random `NEWID` to each row:**

```sql
SELECT *, NEWID() AS NewID FROM Employees;
```

8. **Select the top 3 random employees using `NEWID()`:**

```sql
SELECT TOP(3) * FROM Employees ORDER BY NEWID();
```

9. **Select employees where the full name matches `'Amr Abdo'` (avoiding the error):**
   - Since SQL doesn't allow referencing an alias in the `WHERE` clause, you have two solutions.

   Solution 1: Using concatenation in the `WHERE` clause:

```
SELECT Fname + ' ' + Lname AS Fullname FROM Employees WHERE Fname + ' ' +
Lname = 'Amr Abdo'; => Error ------------------------------------------
-------------------- SELECT Fname + ' ' + Lname AS Fullname FROM Employee
s ORDER BY Fname; => True
```

> This means that to avoid errors, especially when using aliases in WHERE, you either need to compute values within the WHERE clause or use a subquery as shown in the above example.

   Solution 2: Using a subquery:

```
SELECT * FROM ( SELECT Fname + ' ' + Lname AS Fullname FROM Employees ) A
S NewTable WHERE Fullname = 'Amr Abdo';
```

Rule for SQL Query Execution Order:

When SQL executes a query, the following order is followed:

1. `FROM`
2. `JOIN`
3. `ON`
4. `WHERE`
5. `GROUP BY`
6. `HAVING` (aggregate functions)
7. `SELECT`
8. `ORDER BY`
9. `TOP` (if applicable)

Database Objects in SQL Server:

1. **Tables**:
   - Store data in rows and columns.
2. **Views**:
   - Virtual tables based on the result set of a SQL query, often used to simplify complex queries.
3. **Functions**:
   - Return a single value (scalar function) or a table (table-valued function). Used to encapsulate reusable logic.
4. **Stored Procedures**:
   - Precompiled collections of one or more SQL statements that can be executed as a unit. Useful for code reuse and better performance.
5. **Rules**:
   - Used to enforce domain integrity by specifying which data values are allowed in a column.

Default Path to Access Database Objects:

The full path to access any database object in SQL Server follows this structure:

```
[ServerName].[DatabaseName].[SchemaName].[ObjectName]
```

- **ServerName**: The name of the SQL Server instance.
- **DatabaseName**: The name of the database in the instance.
- **SchemaName**: The schema (usually `dbo` by default) that groups database objects.
- **ObjectName**: The actual object being referenced (table, view, etc.).

Query Examples:

1. **With full path:**

```sql
SELECT * FROM [DESKTOP-DLB8G1R\\SQL2022].CompanyDB.dbo.Employees;
```

In this query:

- `[DESKTOP-DLB8G1R\\SQL2022]` is the server instance.
- `Company_SD` is the database.
- `dbo` is the schema.
- `Employee` is the table.

1. **Without server name (when connected to the same server):**

```sql
SELECT * FROM CompanyDB.dbo.Employees;
```

This query omits the server name since you're already connected to the SQL Server instance.

By specifying the schema (`dbo`), it helps SQL Server locate the object more efficiently, though it's optional if there's only one schema in use.

Copying Tables in SQL Server Using DDL:

1. Copy Table Structure and Data:

You can create a new table and copy both the structure and the data from an existing table using the `SELECT INTO` statement.

Example:

```sql
SELECT * INTO table2 FROM Employees;
```

- This creates `table2` with the same structure as `Employee` and copies all the data from the `Employee` table into `table2`.

2. Copy Table Structure Only (No Data):

To copy only the structure (i.e., column definitions) but **no data**, you can use a `SELECT INTO` statement with a condition that is never true, or script the table creation manually.

Example:

```
SELECT * INTO table3 FROM Employees WHERE Sex = 'wme fkwej'; -- Condition tha
t will never match
```

- This creates `table3` with the same structure as `Employee`, but no rows will be inserted because the condition is false.

Alternatively, you can use SQL Server Management Studio (SSMS):

- Right-click the table (e.g., `Employee`) in the Object Explorer.
- Choose **Script Table as** > **CREATE To** > **New Query Editor Window**.
- This will generate the `CREATE TABLE` script without data.

3. Copy Table Data Only:

If the table already exists, and you just want to copy the data from another table into it, you can use the `INSERT INTO` statement.

Example:

```
INSERT INTO table2 SELECT * FROM Employees;
```

- This copies the data from `Employee` into `table2`. Note that `table2` must already exist, and the structure must be compatible with `Employee`.

Using `HAVING` Without `GROUP BY`

In SQL, the `HAVING` clause is typically used to filter groups of rows returned by a `GROUP BY` statement. However, it can also be used without `GROUP BY` when the selected attribute is an aggregate function. This allows you to filter results based on aggregate values directly.

Examples:

1. **Using `HAVING` with an Aggregate Function**:
   - In this example, we use the `SUM` and `COUNT` aggregate functions without a `GROUP BY` clause:

   ```
   SELECT SUM(Salary) AS TotalSalary FROM Employees HAVING COUNT(SSN) < 100;
   ```

   - This query calculates the total salary of all employees but only returns the result if the count of `SSN` (which represents the total number of employees) is less than 100.
2. **Another Example**:
   - Similarly, you can use the `HAVING` clause to impose conditions on aggregate values:

   ```
   SELECT SUM(Salary) AS TotalSalary FROM Employees HAVING COUNT(SSN) > 100;
   ```

   - This query computes the total salary of all employees but only returns the result if the total number of employees (as counted by `SSN`) is greater than 100.

Ranking Functions in SQL

Ranking functions in SQL are used to assign a rank or number to each row within a partition of a result set. These functions can help analyze data by providing insights based on ordered values.

1. `ROW_NUMBER()`

- **Description**: This function assigns a unique sequential integer to rows within a partition of a result set. It starts at 1 and increments for each row.
- **Usage**: Use `ROW_NUMBER()` when you want to assign a unique number to each row based on a specific order.

```
SELECT *, ROW_NUMBER() OVER (ORDER BY Salary DESC) AS RN FROM Employees;
```

- **Example Output**: If you have salaries like 5000, 4500, 4000, the output would rank them as 1, 2, 3.

2. `DENSE_RANK()`

- **Description**: Similar to `ROW_NUMBER()`, but it assigns the same rank to rows with equal values. The next rank will be the immediate next integer, without gaps.
- **Usage**: Use `DENSE_RANK()` when you want to group identical values and avoid gaps in ranking.

```
SELECT *, DENSE_RANK() OVER (ORDER BY Salary DESC) AS DR FROM Employees;
```

- **Example Output**: For salaries like 5000, 4500, 4500, the output would be ranked as 1, 2, 2.

```
SELECT * FROM (SELECT *, ROW_NUMBER() OVER (ORDER BY Salary DESC) AS RN,
DENSE_RANK() OVER (ORDER BY Salary DESC) AS DR FROM Employees) AS NewTable
WHERE RN = 1;
```

3. `NTILE(n)`

- **Description**: This function divides the result set into `n` groups (or buckets) and assigns a group number to each row. The last group may contain fewer rows if the total number of rows isn't evenly divisible by `n`.
- **Usage**: Use `NTILE(n)` when you want to segment your data into a specified number of groups.

```
SELECT *, NTILE(4) OVER (ORDER BY Salary DESC) AS G FROM Employees; ---------
------------------------------------------------ SELECT * FROM (SELECT *, NT
ILE(4) OVER (ORDER BY Salary DESC) AS G FROM Employees) AS NewTable WHERE G =
1;
```

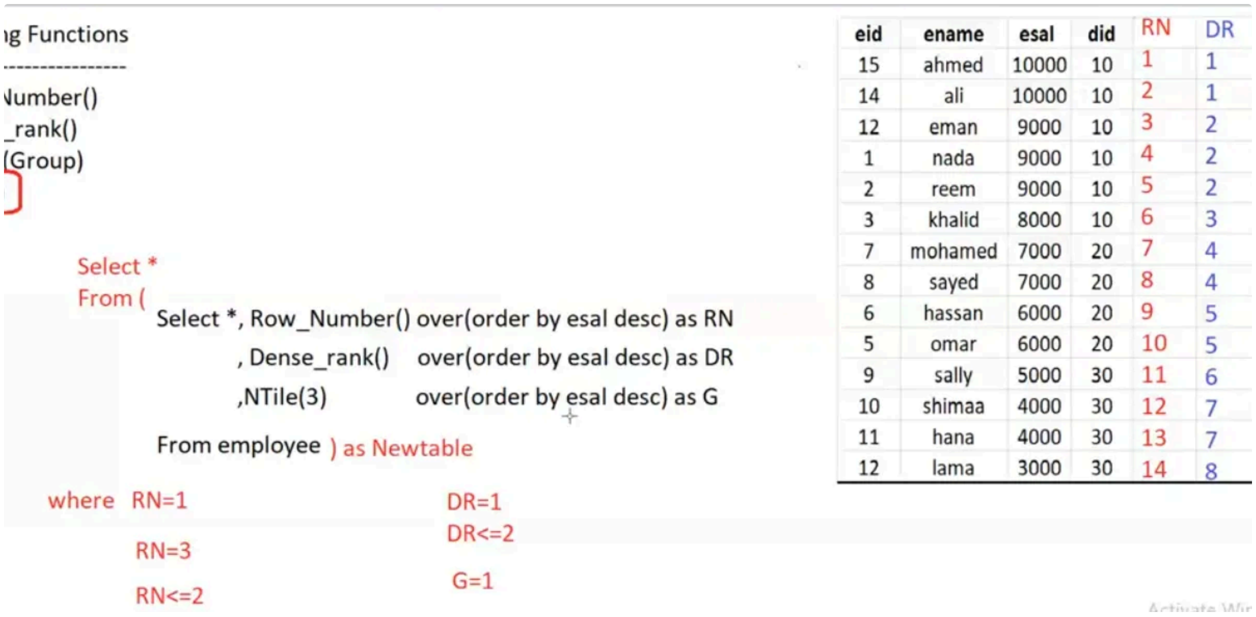
- **Example Output**: For 13 rows divided into 4 groups, you might get groups with sizes 4, 3, 3, and 3.

4. `RANK()`

- **Description**: Similar to `DENSE_RANK()`, but it leaves gaps in the ranking when there are ties. The next rank after a tie will skip numbers.
- **Usage**: Use `RANK()` when you need to acknowledge ties but want to maintain the ranking order.

```sql
SELECT *, RANK() OVER (ORDER BY Salary DESC) AS G FROM Employees;
```

- **Example Output**: For salaries like 5000, 4500, 4500, 4000, the output would be ranked as 1, 2, 2, 4.



g Functions
----
umber()
_rank()
(Group)

```
Select *
From (
      Select *, Row_Number() over(order by esal desc) as RN
                , Dense_rank()  over(order by esal desc) as DR
                ,NTile(3)       over(order by esal desc) as G
      From employee ) as Newtable
where  RN=1                    DR=1
                               DR<=2
       RN=3
                               G=1
       RN<=2
```

eid	ename	esal	did	RN	DR
15	ahmed	10000	10	1	1
14	ali	10000	10	2	1
12	eman	9000	10	3	2
1	nada	9000	10	4	2
2	reem	9000	10	5	2
3	khalid	8000	10	6	3
7	mohamed	7000	20	7	4
8	sayed	7000	20	8	4
6	hassan	6000	20	9	5
5	omar	6000	20	10	5
9	sally	5000	30	11	6
10	shimaa	4000	30	12	7
11	hana	4000	30	13	7
12	lama	3000	30	14	8

1. Select the Third Minimum Salary:

You can use either the `ROW_NUMBER()` function or a subquery approach to achieve this.

- **Using a Subquery**:

```sql
SELECT MIN(Salary) FROM ( SELECT TOP(3) Salary FROM Employees ORDER BY Salary
DESC ) AS Table_One;
```

This query selects the third highest salary using a subquery that retrieves the top 3 salaries.

- Using `ROW_NUMBER()`:

```
SELECT * FROM ( SELECT *, ROW_NUMBER() OVER (ORDER BY Salary DESC) AS RN FROM
Employees ) AS newtable WHERE RN = 3;
```

This query uses the `ROW_NUMBER()` function to assign a rank to each salary and selects the third highest.

2. Select All Rows with Rank Less Than or Equal to 4:

This query uses the `DENSE_RANK()` function to return all rows with ranks 1 to 4, considering ties.

- Using `DENSE_RANK()` with Rank <= 4:

```
SELECT * FROM ( SELECT *, DENSE_RANK() OVER (ORDER BY Salary DESC) AS DR FROM
Employee ) AS newtable WHERE DR <= 4;
```

This query selects rows where the rank (based on salary) is less than or equal to 4, and rows with the same salary will have the same rank.

3. Select Rows with Rank Exactly Equal to 4:

This query returns only the rows with rank 4 (no rows below or above rank 4).

- Using `DENSE_RANK()` with Rank = 4:

```
SELECT * FROM ( SELECT *, DENSE_RANK() OVER (ORDER BY Salary DESC) AS DR FROM
Employee ) AS newtable WHERE DR = 4;
```

This query selects only the rows where the rank equals 4.

4. Divide Employees into 4 Groups and Get Minimum Salary from Each Group:

The `NTILE()` function is used here to divide the rows into four groups and select the minimum salary from each group.

- **Using `NTILE(4)` to Divide into Groups and Find Minimum Salary**:

```
SELECT MIN(Salary), G FROM ( SELECT *, NTILE(4) OVER (ORDER BY Salary DESC) A
S G FROM Employee ) AS newtable GROUP BY G;
```

This query divides the employees into 4 groups based on their salary and selects the minimum salary from each group.

5. Select All Rows in Group 1:

Using the `NTILE()` function to group rows, you can select all rows in a specific group, in this case, group 1.

- **Using `NTILE(4)` to Select All Rows in Group 1**:

```
SELECT * FROM ( SELECT *, NTILE(4) OVER (ORDER BY Salary DESC) AS G FROM Empl
oyee ) AS newtable WHERE G = 1;
```

This query selects all employees who belong to group 1 after dividing the employees into 4 groups based on salary.

These queries demonstrate how to use SQL ranking functions (`ROW_NUMBER()`, `DENSE_RANK()`, `NTILE()`) effectively in practical scenarios for the **CompanyDB** database.

Examples Using Partition in Ranking Functions:

1. **Assign Row Numbers Within Each Department:**

   ```
   SELECT *, ROW_NUMBER() OVER (PARTITION BY DepartmentID ORDER BY Salary DE
   SC) AS RN FROM Employees;
   ```

   - This query divides the `Employees` table into groups based on `DepartmentID` (department number).
   - For each group, it assigns a row number starting from 1, ordered by `Salary` in descending order.

2. **Assign Dense Ranks Within Each Department:**

   ```
   SELECT *, DENSE_RANK() OVER (PARTITION BY DepartmentID ORDER BY Salary DE
   SC) AS DR FROM Employees;
   ```

   - Similar to the above, but uses `DENSE_RANK()` to assign ranks based on salary.
   - Employees with the same salary will have the same rank.

3. **Select Maximum Salary for Each Department (Using Row Number):**

   ```
   SELECT * FROM ( SELECT *, ROW_NUMBER() OVER (PARTITION BY DepartmentID OR
   DER BY Salary DESC) AS RN FROM Employees ) AS NewTable WHERE RN = 1;
   ```

   - This query selects the top employee (with the highest salary) in each department (`DepartmentID`) by filtering `ROW_NUMBER()` where `RN = 1`.

4. **Select Maximum Salary for Each Department (Using Dense Rank):**

   ```
   SELECT * FROM ( SELECT *, DENSE_RANK() OVER (PARTITION BY DepartmentID OR
   DER BY Salary DESC) AS DR FROM Employees ) AS NewTable WHERE DR = 1;
   ```

   - This query selects all employees who have the highest salary in each department (`DepartmentID`), allowing for repetition if multiple employees have the same salary.

5. **Select Top 2 Salaries in Each Department (Using Dense Rank):**

```
SELECT * FROM ( SELECT *, DENSE_RANK() OVER (PARTITION BY DepartmentID OR
DER BY Salary DESC) AS DR FROM Employees ) AS NewTable WHERE DR <= 2;
```

   - This query selects all employees who have the top 2 salaries in each department, including those with ties (i.e., if multiple employees have the same salary).

6. The query retrieves all employees who fall into the **first group** (quartile) when the employees are divided into **four equal groups** based on their `DepartmentID`, ordered in **descending order**.

```
SELECT * FROM ( SELECT *, NTILE(4) OVER (ORDER BY DepartmentID DESC) AS G
FROM Employees ) AS NewTable WHERE G = 1;
```

In SQL Server, data types are categorized as follows:

1. Numeric Data Types:

- **bit**: Stores `0` or `1` (true or false).
- **tinyint**: 1 byte (range: `0 to 255`).
- **smallint**: 2 bytes (range: `32,768 to 32,767` for signed, `0 to 65,535` for unsigned).
- **int**: 4 bytes.
- **bigint**: 8 bytes.

2. Decimal Data Types:

- **smallmoney**: 4 bytes (precision up to four decimal places).
- **money**: 8 bytes (precision up to four decimal places).
- **real**: Floating point with precision up to seven decimal places.
- **float**: Floating point with precision up to fifteen decimal places.

- **decimal** or **dec(p, s)**: Fixed precision and scale. For example, `decimal(5,2)` allows up to five digits, with two digits after the decimal point (e.g., `33.44`, `23.3` are valid, but `3333.3` and `33.444` are invalid).

3. Character Data Types:

- **char(n)**: Fixed-length characters, reserves `n` bytes (e.g., `char(10)` always reserves 10 bytes).
- **varchar(n)**: Variable-length characters, reserves space based on the input.
- **nchar(n)**: Fixed-length Unicode characters (used for storing multilingual data).
- **nvarchar(n)**: Variable-length Unicode characters.
- **nvarchar(max)**: Variable-length Unicode characters, allows storage up to 2 GB.

4. Datetime Data Types:

- **date**: Stores only the date in `mm/dd/yyyy` format.
- **time**: Stores time as `hh:mm:ss` or `hh:mm:ss.sssssss`.
- **smalldatetime**: Stores `mm/dd/yyyy hh:mm:00`.
- **datetime**: Stores `mm/dd/yyyy hh:mm:ss.fff`.
- **datetime2**: Stores `mm/dd/yyyy hh:mm:ss.fffffff`.
- **datetimeoffset**: Includes time zone offset, `yyyy-mm-dd hh:mm:ss.fffffff ±hh:mm`.

5. Binary Data Types:

- **binary**: Stores fixed-length binary data (e.g., binary(10)).
- **image**: Stores images as binary data in the database.

6. Other Data Types:

- **xml**: Stores XML data.
- **uniqueidentifier**: Stores globally unique identifiers (GUIDs).
- **sql_variant**: Stores values of different data types in a single column.

These categories help in choosing the correct data types based on your storage needs, precision requirements, and performance considerations.

Here are two commonly used functions in SQL: **CASE** and **IIF**:

1. CASE Statement:

The `CASE` statement is used for conditional logic in SQL. It allows you to create different outcomes based on specified conditions. You can use it in `SELECT`, `UPDATE`, and other SQL statements.

Example 1: Using `CASE` in a `SELECT` query

This query categorizes employee salaries into "low", "medium", and "high" based on the salary value:

```
SELECT EmployeeName, Salary, CASE WHEN Salary <= 800 THEN 'low' WHEN Salary <
= 1500 THEN 'medium' WHEN Salary > 1500 THEN 'high' ELSE 'No value' END AS Cr
iteria FROM Employees;
```

Example 2: Using `CASE` in an `UPDATE` query

This query updates the salary of employees based on their current salary. The salary is increased by a certain percentage depending on its value:

```
UPDATE Employees SET Salary = CASE WHEN Salary <= 800 THEN Salary * 1.2 WHEN
Salary <= 1500 THEN Salary * 1.3 WHEN Salary > 1500 THEN Salary * 1.4 ELSE 0
END;
```

2. IIF() Function:

The `IIF()` function is a simplified version of `CASE`, often used for evaluating a single condition. It works similarly to the `IF` statement in programming languages.

Example: Using `IIF()` in a `SELECT` query

This query checks whether the employee's salary is greater than or equal to 1600 and categorizes it as "high" or "low":

```
SELECT EmployeeName, IIF(Salary >= 1600, 'high', 'low') AS SalaryCategory FRO
M Employees;
```

- `CASE` is more flexible and can handle multiple conditions.
- `IIF` is simpler but can only handle one condition and two outcomes (like an `IF-ELSE` in programming).

In SQL Server, you can use the `CONVERT` and `CAST` functions to change data types, including converting dates to strings with specific formats.

1. Casting Date to String:

You can cast a `DATETIME` value to a `VARCHAR` string to change its representation.

This converts the current date and time (`GETDATE()`) to a `VARCHAR(20)` string.

```
SELECT CONVERT(VARCHAR(20), GETDATE()); -- Converts current date/time to stri
ng
```

The `CAST()` function can also be used for casting the current date to a string, similar to `CONVERT()`.

```
SELECT CAST(GETDATE() AS VARCHAR(20)); -- Alternative way to cast date to str
ing
```

2. Date Formatting with `CONVERT()`:

The `CONVERT()` function allows you to specify a date format by providing a format code as the third parameter. Each code represents a different date format.

Example of Different Date Formats:

This will return the current date in the format `YYYY.MM.DD`.

```
SELECT CONVERT(VARCHAR(20), GETDATE(), 102); -- Format: YYYY.MM.DD
```

This will return the date in the format `DD/MM/YYYY`.

```
SELECT CONVERT(VARCHAR(20), GETDATE(), 103); -- Format: DD/MM/YYYY
```

This will return the date in the format `DD.MM.YYYY`.

```
SELECT CONVERT(VARCHAR(20), GETDATE(), 104); -- Format: DD.MM.YYYY
```

This will return the date in the format `DD-MM-YYYY`.

```
SELECT CONVERT(VARCHAR(20), GETDATE(), 105); -- Format: DD-MM-YYYY
```

```
SELECT FORMAT(GETDATE(), 'G') AS DefaultFormat, FORMAT(GETDATE(), 'd') AS
ShortDate, FORMAT(GETDATE(), 'D') AS LongDate, FORMAT(GETDATE(), 'F') AS
FullDateTime, FORMAT(GETDATE(), 't') AS ShortTime, FORMAT(GETDATE(), 'T') AS
LongTime, FORMAT(GETDATE(), 'yyyy') AS Year, FORMAT(GETDATE(), 'MMMM') AS
Month, FORMAT(GETDATE(), 'dd') AS Day, FORMAT(GETDATE(), 'dd/MM/yyyy') AS
CustomDate, FORMAT(GETDATE(), 'yyyy-MM-ddTHH:mm:ss') AS ISOFormat;
FORMAT(GETDATE(), 'MM/dd/yyyy') AS MonthDayYear, FORMAT(GETDATE(), 'dd-MM-
yyyy') AS DayMonthYear, FORMAT(GETDATE(), 'HH:mm:ss') AS Time24Hour,
FORMAT(GETDATE(), 'dddd') AS DayOfWeek, FORMAT(GETDATE(), 'MMMM d, yyyy') AS
OrdinalDate, FORMAT(GETDATE(), 'C', 'en-US') AS CurrencyFormat,
FORMAT(0.1234, 'P') AS PercentageFormat, FORMAT(12345.6789, 'N2') AS
FixedPoint, FORMAT(GETDATE(), 'yyyy-MM-ddTHH:mm:ss.fff') AS ISODateTime;
```