What is a View in SQL?

A **view** in SQL is a virtual table that is based on the result of a query. It does not store the data physically, but displays it dynamically based on a SELECT query from one or more tables.

Types of Views in SQL

1. **Simple View**
   - **Definition**: A simple view is based on a single table and does not contain any aggregate functions, joins, or complex subqueries.
   - **Syntax**:

```
CREATE VIEW simple_view AS
SELECT column1, column2
FROM table_name;
```

2. **Materialized View**
   - **Definition**: A materialized view stores the query result physically, unlike a regular view, which is virtual. It improves performance by storing the result of complex queries.
   - **Syntax** (for Oracle, as an example):

```
CREATE MATERIALIZED VIEW mat_view AS
SELECT column1, COUNT(column2)
FROM table_name
GROUP BY column1;
```

3. **Inline View (Subquery View)**
   - **Definition**: An inline view is a subquery in the FROM clause, treated as a table or view for the query.
   - **Syntax**:

```
SELECT subquery.column1
FROM (SELECT column1 FROM table_name) AS subquery;
```

   - **Advantages**:
     - Provides flexibility for complex query operations without creating permanent views.
     - Can break down complex queries into smaller, manageable subqueries.
   - **Disadvantages**:
     - Temporary in nature; cannot be reused in other queries.
     - Can reduce readability and increase complexity.

What is an INDEX in SQL?

An **index** in SQL is a **database object** that improves the speed of data retrieval operations on a table at the cost of additional space and increased maintenance time during data modifications (INSERT, UPDATE, DELETE). It is essentially a data structure that allows for faster searching, sorting, and filtering of data in a database.

Important Types of Indexes in SQL

Here are the most important types of indexes that are commonly asked in interviews:

1. **Unique Index**
2. **Composite (Concatenated) Index**
3. **Clustered Index**
4. **Non-Clustered Index**

1. Unique Index

- **Definition**: A **unique index** ensures that all values in the indexed column are unique. It is automatically created when you define a **UNIQUE** constraint on a column or set of columns.

```
CREATE UNIQUE INDEX idx_employee_name ON Employee (emp_name);
```

Slower performance on write operations (INSERT, UPDATE, DELETE) due to index maintenance.

2. Composite (Concatenated) Index

- **Definition**: A **composite index** is an index on two or more columns in a table. It is useful when you often query the table using multiple columns.
- **Syntax**:

```
CREATE INDEX idx_employee_name_salary ON Employee (emp_name, emp_salary);
```

Slower performance on write operations (INSERT, UPDATE, DELETE) due to index maintenance.

Less efficient if only part of the index is used.

3. Clustered Index

- **Definition**: A **clustered index** determines the physical order of data rows in the table. Each table can have only **one clustered index**. The primary key is usually the clustered index unless specified otherwise.
- **Faster data retrieval** for range queries (e.g., WHERE emp_id BETWEEN 100 AND 200).

Slower performance on write operations (INSERT, UPDATE, DELETE) due to index maintenance.

4. Non-Clustered Index

- **Definition**: A **non-clustered index** creates a separate object that points to the data rows in the table. It does not alter the physical order of rows. You can have multiple non-clustered indexes on a table.
- **Syntax**:

```
CREATE INDEX idx_emp_name ON Employee (emp_name);
```

Slower performance on write operations (INSERT, UPDATE, DELETE) due to index maintenance.

COMMON TABLE EXPRESSIONS (CTE)

In SQL, a Common Table Expression (CTE) is an essential tool for simplifying complex queries and making them more readable. By defining temporary result sets that can be referenced multiple times, a CTE in SQL allows developers to break down complicated logic into manageable parts.

Syntax

```
WITH cte_name AS (
SELECT query
)
SELECT *
FROM cte_name;
```

CTE vs Subqueries

Feature	CTE	Subquery
Reusability	Can be referenced multiple times.	Typically used once.
Readability	Improves readability for complex queries.	Can become difficult to read when nested.
Performance	Optimized for multiple references.	May be less efficient for repeated operations.

EX: WAQTD employee names along with their dnames

```
WITH EmployeeDepartment AS (
   SELECT e.empno, e.ename, d.dname
   FROM Emp e ,Dept d
   WHERE e.deptno = d.deptno
)
SELECT * FROM EmployeeDepartment;
```

The SQL CASE Expression

The CASE expression goes through conditions and returns a value when the first condition is met (like an if-then-else statement). So, once a condition is true, it will stop reading and return the result. If no conditions are true, it returns the value in the ELSE clause.

If there is no ELSE part and no conditions are true, it returns NULL.

SYNTAX:

```
CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    WHEN conditionN THEN resultN
    ELSE result
END;
```

```
SELECT EMPNO, ENAME,
CASE
WHEN SAL > 4000 AND SAL < 6000 THEN 'EARNING GOOD'
WHEN SAL > 2000 AND SAL < 3000 THEN 'NORMAL'
WHEN SAL > 500 AND SAL < 1500 THEN 'POOR'
ELSE 'BEGGER'
END AS EARNINGSALARY
FROM EMP;
```

WINDOWS FUNCTIONS

A window function in SQL performs a calculation across a set of table rows related to the current row within a specified window. Unlike regular aggregate functions, window functions allow you to retain individual rows while performing the calculation.

The OVER clause is key to defining this window. It partitions the data into different sets (using the **PARTITION BY** clause) and orders them (using the **ORDER BY** clause). These windows enable functions like **SUM (), AVG (), ROW_NUMBER (), RANK (),** and **DENSE_RANK ()** to be applied in a sophisticated manner.

Syntax

```
SELECT column_name1,
window_function(column_name2)
OVER ([PARTITION BY column_name1] [ORDER BY column_name3]) AS
new_column
FROM table_name;
```

Windows functions can be categorized into 2 types:

1. **Aggregate window functions**
2. **Ranking window functions**

1. Aggregate Window Function

Aggregate window functions calculate aggregates over a window of rows while retaining individual rows. These include SUM (), AVG (), COUNT (), MAX (), and MIN ().

Example: **AVG ()** Function

Query to calculate the average salary within each department:

```
SELECT Name, Age, Department, Salary,
 AVG(Salary) OVER (PARTITION BY Department) AS Avg_Salary
 FROM employee;
```

2. Ranking Window Functions

These functions provide rankings of rows within a partition based on specific criteria. Common ranking functions include:

1. RANK () Function

The RANK () function assigns ranks to rows within a partition, with the same rank given to rows with identical values. If two rows share the same rank, the next rank is skipped.

Example:

```
SELECT Name, Department, Salary,
       RANK () OVER (PARTITION BY Department ORDER BY Salary DESC) AS emp_rank
FROM employee;
```

2. DENSE_RANK () Function

It assigns rank to each row within partition. Just like rank function first row is assigned rank 1 and rows having same value have same rank. The difference between RANK () and DENSE_RANK () is that in DENSE_RANK (), for the next rank after two same rank, consecutive integer is used, no rank is skipped.

Example:

```
SELECT Name, Department, Salary,
       DENSE_RANK () OVER (PARTITION BY Department ORDER BY Salary DESC) AS emp_dense_rank
FROM employee;
```

3. ROW_NUMBER () Function

ROW_NUMBER () gives e-ach row a unique number. It numbers rows from one- to the total rows. The rows are put into groups based on their values. Each group is called a partition. In e-ach partition, rows get numbers one afte-r another. No two rows have the same- number in a partition.

Example:

```
SELECT Name, Department, Salary,
       ROW_NUMBER () OVER (PARTITION BY Department ORDER BY Salary DESC) AS emp_row_no
FROM employee;
```