REGULAR EXPRESSION

In Microsoft SQL Server, regular expressions aren't natively supported, but you can perform basic pattern matching using the LIKE operator with wildcard characters.

1. Wildcard Characters in LIKE

% (Percent Sign): Matches zero or more characters. Example: LIKE 'A%'

_ (Underscore): Matches exactly one character. LIKE '_b%'

2. Basic Pattern Matching Scenarios

Beginning of String: Use % after the string to match. Example: LIKE 'Sa%' finds names that start with "Sa" (e.g., "Sam", "Sarah").

End of String: Use % before the string to match. Example: LIKE '%on' finds names ending in "on" (e.g., "Norton", "Merton").

Specific Character Position: Use _ to define exact positions. Example: LIKE '__a%' finds strings with "a" as the third character (e.g., "Sarah", "Shawn").

3. Combining Patterns

Multiple Wildcards: Combine % and _ for more complex patterns. Example: LIKE '%a_e%' matches any string containing "a", followed by any character, then "e" (e.g., "Sable", "Cameo").

4. Using Square Brackets [ ] for Character Sets

Character Sets: Match any single character within brackets. Example: LIKE '[ABC]%' matches strings starting with "A", "B", or "C".

Character Ranges: Specify a range within brackets. Example: LIKE '[a-z]%' matches strings starting with any lowercase letter.

Negation: Use [^] inside brackets to exclude characters. Example: LIKE '[^A-C]%' matches strings that do not start with "A", "B", or "C".

SUB QUERY

In Microsoft SQL Server, a **subquery** is a query nested inside another SQL query. Subqueries are often used to retrieve data that will be used in the main query, allowing for complex data retrieval and filtering.

1. Types of Subqueries

Single-Value (Scalar) Subquery: Returns a single value (one row, one column).

SELECT name FROM employees WHERE salary > (SELECT AVG(salary) FROM employees);

Multi-Row Subquery: Returns multiple rows (one column) and is often used with IN, ANY, ALL.

SELECT name FROM employees WHERE department_id IN (SELECT id FROM departments WHERE location = 'NY');

Multi-Column Subquery: Returns multiple columns (one or more rows) and is typically used in the WHERE clause with IN or EXISTS.

SELECT name FROM employees WHERE (department_id, job_id) IN (SELECT department_id, job_id FROM jobs WHERE title = 'Manager');

2. Common Uses of Subqueries

In the WHERE Clause: Filters data based on a condition involving another query.

SELECT name FROM employees WHERE salary = (SELECT MAX(salary) FROM employees);

In the FROM Clause (Inline View): Treats a subquery as a temporary table for the main query.

SELECT AVG(salary) FROM (SELECT salary FROM employees WHERE department_id = 10) AS dept_salary;

In the SELECT Clause: Calculates a value for each row in the main query.

SELECT name, (SELECT COUNT(*) FROM orders WHERE orders.employee_id = employees.id) AS order_count FROM employees;

3. Correlated Subquery

A **correlated subquery** refers to columns from the outer query, running once per row of the outer query.

Find employees with salaries above their department's average.

SELECT name FROM employees e WHERE salary > (SELECT AVG(salary) FROM employees WHERE department_id = e.department_id);

4. Key Points

- **Execution**: Subqueries are typically executed before the main query.
- **Nesting**: Subqueries can be nested multiple levels deep, though this may affect performance.
- **Performance**: For complex queries, consider using JOIN or EXISTS to optimize.

COMMON TABLE EXPRESSION

In Microsoft SQL Server, a **Common Table Expression (CTE)** is a temporary result set defined within the execution of a SELECT, INSERT, UPDATE, or DELETE statement. CTEs improve readability and reusability, especially in complex queries involving recursion or multiple joins.

1. Defining a CTE

A CTE is defined using the WITH keyword followed by the CTE name and the AS keyword, with a query in parentheses.

```
WITH SalesCTE AS (
   SELECT employee_id, SUM(sales) AS total_sales  FROM sales  GROUP BY employee_id
)
SELECT employee_id, total_sales FROM SalesCTE WHERE total_sales > 10000;
```

2. Advantages of Using CTEs

Readability: Simplifies complex queries by breaking them into manageable parts.

Reusability: Allows for the reuse of CTEs in subsequent parts of the main query.

Recursion: Enables recursive operations, especially useful for hierarchical data (e.g., organizational charts).

3. Types of CTEs

Non-Recursive CTE: A standard CTE without self-referencing. It's simply a temporary result set used in the main query.

Recursive CTE: A CTE that references itself, commonly used for hierarchical data processing. It has an **anchor member** (base result) and a **recursive member** (references the CTE itself).

```
WITH EmployeeHierarchy AS (
   SELECT employee_id, manager_id, 1 AS level FROM employees WHERE manager_id IS NULL
   UNION ALL
   SELECT e.employee_id, e.manager_id, eh.level + 1 FROM employees e
   INNER JOIN EmployeeHierarchy eh ON e.manager_id = eh.employee_id
)
SELECT * FROM EmployeeHierarchy;
```

4. CTE Scope and Usage

- CTEs exist only for the duration of the query they are defined in.
- Multiple CTEs can be defined in a single query by separating them with commas.
- CTEs cannot be directly indexed or used outside of their defining query but can reference other CTEs within the same WITH clause.

STAR AND SNOW FLAKE SCHEMA

Star Schema

Structure: In a star schema, a central **fact table** is surrounded by **dimension tables**. The fact table contains quantitative data (e.g., sales amounts), while dimension tables contain descriptive information (e.g., product, customer, time).

Design: The schema resembles a star shape, with each dimension table directly connected to the fact table.

Simplified Joins: Dimension tables connect only to the fact table, making joins simple and improving query performance.

Denormalized Data: Dimension tables are often denormalized (no sub-divisions), which reduces the number of joins but increases redundancy.

Use Cases: Suitable for simple, high-performance read-heavy analytical queries where speed is critical.
Example:
Fact Table: sales (order_id, product_id, customer_id, sales_amount, date)
Dimension Tables: products, customers, dates, locations

Snowflake Schema

Structure: Similar to a star schema, but with normalized (multi-level) dimension tables, creating a structure resembling a snowflake.

Design: Dimension tables are split into additional tables, reducing redundancy. For example, a "location" dimension may split into "country," "state," and "city" tables.

Normalized Data: Dimension tables are normalized to reduce redundancy, but this requires more joins and can slow down query performance.

Complex Joins: Queries require more joins due to the multi-level tables, which can lead to more complex query designs and slower performance.

Use Cases: Suitable when storage efficiency is prioritized, or when dimensions are large and need to be broken down for efficient management.
Example:
Fact Table: sales (order_id, product_id, customer_id, sales_amount, date)
Dimension Tables: products, customers, dates, locations (normalized into city, state, country)

Comparison

Complexity: Star schema is simpler; snowflake schema is more complex.
Performance: Star schema has faster query performance due to fewer joins; snowflake schema is slower due to multiple joins.
Storage: Star schema requires more storage due to denormalized data; snowflake schema is more storage-efficient due to normalization.

VIEWS

Definition and Structure:

A view is created with the CREATE VIEW statement, followed by a SELECT query defining the view's structure. Views don't store data physically; instead, they fetch data from the underlying tables each time they are queried.

Advantages:

Simplifies Complex Queries: Views allow you to save complex queries, making them reusable with a simple SELECT statement.

Data Security: Views can restrict access to specific columns and rows, allowing users to see only the data they need.

Data Abstraction: Changes in the underlying tables can be hidden from users, providing a layer of abstraction.

Code Reusability: Views provide reusable queries that can be referenced by multiple applications or queries.

Creating a View:

```
CREATE VIEW EmployeeSales AS
SELECT employee_id, SUM(sales_amount) AS total_sales FROM sales GROUP BY employee_id;
```

Modifying a View:

```
ALTER VIEW EmployeeSales AS
SELECT employee_id, SUM(sales_amount) AS total_sales, COUNT(order_id) AS order_count
FROM sales GROUP BY employee_id;
```

Updating Data Through Views:

Views are usually read-only, but in some cases, you can update the underlying data if the view is based on a single table and doesn't include complex joins or aggregations.

Limitations:

Views cannot contain certain elements, such as ORDER BY (unless combined with TOP), and cannot reference temporary tables.

Performance might be slower for complex views due to the underlying query execution each time the view is called.