Self-Study

1. When A Stored Procedure Is Cached and What is Cold Cache?

   In SQL Server, a **stored procedure** is cached when it is first executed. During this initial execution, SQL Server generates an **execution plan** (a roadmap for how the query will be processed) and stores it in the **plan cache**, a part of SQL Server's memory. This cached plan allows subsequent executions of the same stored procedure to skip the costly compilation step, improving performance. The plan remains in the cache until:

   - **Memory Pressure**: SQL Server removes it to free memory for other operations.
   - **Server Restart**: The entire plan cache is cleared.
   - **Explicit Clearing**: Commands like DBCC FREEPROCCACHE or ALTER DATABASE modifications clear the cache.
   - **Parameter Sniffing Issues**: If a stored procedure is recompiled due to changing data patterns or explicit RECOMPILE hints, a new plan is cached.

   A **cold cache** refers to a state where SQL Server's caches—either the **plan cache** (for execution plans) or the **buffer pool** (for data pages)—are empty or have been cleared. This occurs:

   - After a SQL Server restart, as all caches are reset.
   - When the cache is manually cleared (e.g., DBCC DROPCLEANBUFFERS for the buffer pool or DBCC FREEPROCCACHE for the plan cache).
   - When data or plans have not yet been loaded into memory due to low activity or new queries.

In a cold cache scenario, SQL Server must:

- Recompile execution plans for stored procedures or queries, increasing CPU overhead.
- Read data from disk into the buffer pool, increasing I/O latency.

This results in slower performance for the first execution of queries or stored procedures until the cache is "warmed" by loading plans and data into memory. For example, running a stored procedure on a cold cache may take longer initially but speeds up on subsequent runs as the plan and data are cached.

2. What Is Passing By Value/Reference ?

In C#, **passing by value** and **passing by reference** (ref/out) determine how parameters are handled in method calls. Here's a concise comparison with simple SQL and C# examples:

Passing by Value:

A copy of the parameter's value is passed to the method.
Changes to the parameter inside the method do not affect the original variable.
Default behavior for value types (e.g., int, struct) and immutable reference types (e.g., string).

C# Example:

```
void UpdateValue(int x) { x = 20; }
int num = 10;
UpdateValue(num); // num remains 10
Console.WriteLine(num); // Output: 10
```

SQL Example (Stored Procedure with default parameter passing):

```
CREATE PROCEDURE UpdateValue @Value INT

AS
BEGIN
   SET @Value = 20;
END;
DECLARE @Num INT = 10;
EXEC UpdateValue @Num;
PRINT @Num; -- Output: 10 (original unchanged)
```

Passing by Reference (ref):

Passes the memory address of the parameter, so changes in the method affect the original variable.
Requires the variable to be initialized before passing.

C# Example:

```
void UpdateRef(ref int x) { x = 20; }
int num = 10;
UpdateRef(ref num); // num is modified
Console.WriteLine(num); // Output: 20
```

SQL Example (Using OUTPUT parameter, SQL's equivalent to reference passing):

```
CREATE PROCEDURE UpdateRef @Value INT OUTPUT
AS
BEGIN
   SET @Value = 20;
END;
DECLARE @Num INT = 10;
EXEC UpdateRef @Num OUTPUT;
PRINT @Num; -- Output: 20 (original modified)
```

Passing by Reference (out):

Similar to ref, but the parameter does not need to be initialized before passing, and the method must assign a value.

Used when the method must return a value via the parameter.

C# Example:

```
void GetValue(out int x) { x = 20; }
int num;
GetValue(out num); // num is assigned
Console.WriteLine(num); // Output: 20
```

Differences:

- **By *V*alue**: Copies the value; original is unchanged. Used for simple, independent operations.
- **By Reference (ref)**: Passes the variable's address; changes affect the original. Requires initialization. Used when modifying the caller's variable.
- **By Reference (out)**: Like ref, but no initialization needed; method must assign a value. Used to return values via parameters..

3. SP with dynamic Query

Use Cases for Dynamic SQL in Stored Procedures:

Flexible Queries: Build queries with variable tables, columns, or conditions (e.g., dynamic reports or searches).

Dynamic Data Modifications: Insert, update, or delete data in different tables based on input (e.g., logging to various audit tables).

Custom Sorting/Filtering: Allow users to specify sort order or filter criteria dynamically (e.g., sortable grids in applications).

Schema Modifications: Create or alter database objects dynamically (e.g., creating partitioned tables based on date).

Multi-Tenant Applications: Query tenant-specific tables or schemas dynamically based on tenant ID.

Example:

Query different tables based on a parameter (e.g., selecting data from various log tables).

```
CREATE PROCEDURE GetTableData
   @TableName NVARCHAR(128)
AS
BEGIN

   DECLARE @SQL NVARCHAR(MAX);
   SET @SQL = N'SELECT * FROM ' + QUOTENAME(@TableName);
   EXEC sp_executesql @SQL;

END;

-- Usage
EXEC GetTableData @TableName = 'Orders';
EXEC GetTableData @TableName = 'Customers';
```