**---self-study**

**Procedure Cache and Cold Cache in SQL Server**

In SQL Server, **procedure cache** and **cold cache** are important concepts related to how the server manages query execution performance and memory usage. These affect how quickly SQL Server can execute stored procedures, ad hoc queries, and other batches of code.

**Procedure Cache (Plan Cache)**

The **procedure cache**, also known as the **plan cache**, is a portion of SQL Server's memory that stores **execution plans** for queries and stored procedures. When a query is executed, SQL Server checks whether an execution plan already exists in the cache:

* If a cached plan is found (called a **cache hit**), SQL Server reuses it, avoiding the overhead of parsing, compiling, and optimizing the query again.
* If not (a **cache miss**), SQL Server must generate a new plan, which takes more time and resources.

**Key Benefits of Procedure Cache:**

* **Performance**: Reusing execution plans saves CPU and compilation time.
* **Scalability**: Reduces load on the optimizer during high-concurrency scenarios.

**Cached Objects May Include:**

* Stored procedure execution plans
* Ad hoc and prepared SQL statements
* Triggers
* Functions

**Cold Cache**

A **cold cache** refers to the state of SQL Server immediately **after a restart** or when the cache has been **flushed manually** (e.g., using DBCC FREEPROCCACHE or DBCC DROPCLEANBUFFERS). In this state:

* **No execution plans** are in memory.
* SQL Server must compile all queries from scratch.
* Performance may temporarily degrade due to the overhead of plan generation.

This state is often used in **performance testing** to simulate the server’s behavior when it has no prior knowledge or history of queries.

**When Cold Cache Happens:**

* After a **SQL Server restart**
* After **clearing the procedure cache**
* On a **new SQL Server instance**
* During planned **performance testing or benchmarking**

**Cold Cache vs. Warm Cache**

| **Aspect** | **Cold Cache** | **Warm Cache (Procedure Cache Active)** |
| --- | --- | --- |
| Execution Plan | Must be compiled | Reused from cache |
| Performance | Slower initial execution | Faster due to reuse |
| Typical Scenario | After restart or manual flush | Ongoing database operation |
| Use Case | Benchmarking, testing initial load | Normal day-to-day operations |

In SQL Server, **input** and **output parameters** are used in stored procedures and functions to pass and return data.

**Input Parameters**

* These are used to pass data **into** a stored procedure or function.
* The calling code supplies a value when the procedure is invoked.
* They allow the procedure to work with dynamic values instead of hardcoded ones.
* Declared with the @parameter\_name followed by a data type.

**Output Parameters**

* These are used to pass data **back to the caller** from the stored procedure.
* They must be explicitly marked with the OUTPUT keyword.
* The calling code must also declare the corresponding variable and use the OUTPUT keyword when executing the procedure.
* Useful for returning multiple values, especially when a procedure cannot return a result set or you want additional return information.

**General Use**

* Input and output parameters allow for modular, reusable, and flexible stored procedures.
* They help maintain cleaner code and enable better separation of concerns.
* Parameters must match in name, data type, and direction when being passed between the calling code and the procedure.

**Dynamic Query in SQL Server**

A **dynamic query** is a SQL statement that is constructed and executed at runtime as a string. In SQL Server, this is usually done using the EXEC or sp\_executesql command.

**Stored Procedures with Dynamic Query**

Stored procedures can be written to include **dynamic SQL**. This means the SQL command inside the procedure is built as a string based on parameters or logic, and then executed.

This is useful when:

* The structure of the query (e.g., columns, tables, filters) can vary.
* You need to build flexible WHERE, ORDER BY, or JOIN clauses at runtime.
* Queries involve conditional logic that can’t be expressed easily with static SQL.

**Merits of Using Dynamic SQL in Stored Procedures**

1. **Flexibility**
   * Allows construction of queries with dynamic table names, column names, or conditions.
2. **Code Reusability**
   * One stored procedure can handle many query variations.
3. **Simplified Complex Logic**
   * Dynamic logic is easier for highly conditional query generation.
4. **Runtime Decisions**
   * Enables logic that adapts to user inputs or system state during execution.
5. **Support for Optional Parameters**
   * Easier to build optional filter logic based on provided input.

**Demerits of Using Dynamic SQL in Stored Procedures**

1. **SQL Injection Risk**
   * If not carefully handled, dynamic SQL can be vulnerable to SQL injection attacks.
2. **Harder to Debug and Maintain**
   * Dynamic query strings can be more difficult to read, test, and maintain.
3. **Poor Performance Optimization**
   * Execution plans for dynamic SQL are not always cached, leading to slower performance.
4. **Lack of Compile-Time Checking**
   * SQL Server cannot validate the dynamic query structure at compile time, increasing runtime error risk.
5. **Security & Permissions Issues**
   * May require additional permissions or considerations, especially when dealing with object names as parameters.

In SQL Server, **passing parameters** to stored procedures involves defining and supplying values for them when calling the procedure. Parameters can have:

**1. Order of Parameters**

* Parameters are passed **in the order they are defined** in the stored procedure (if using **positional syntax**).
* **Positional passing** requires the values to match the order, data types, and number of parameters exactly.

**2. Name of Parameters**

* You can pass parameters by **name**, using the format @param\_name = value.
* **Named parameters** can be passed **in any order**, which improves readability and reduces errors.

**3. Default Values**

* Parameters in stored procedures can have **default values**.
* If a value is **not passed** during the call, the default is used.
* This makes the parameter **optional**.
* Default values must be constants (not expressions or variables).

**Key Points**

* You can mix named and positional parameters, but named parameters must come **after** positional ones.
* Using default values allows more flexible and reusable procedures.
* Named parameters improve clarity and reduce dependency on parameter order.