SQL, Stored Procedure, Triggers, XML Tables (Day 9)

In SQL Server, when a query is executed, it goes through several steps to ensure efficient processing and accurate results. The execution of any SQL query follows this sequence:

1. Parsing

- The SQL engine first checks for syntax errors.
- If the syntax is correct, it proceeds; otherwise, it returns an error.

2. Optimization

- This step ensures the validity of metadata, like table names, column names, and data types.
- The optimizer evaluates various ways to execute the query and selects the most efficient one.

3. Query Tree (Logical Plan)

- The SQL engine organizes the query components into a hierarchical structure.

- It follows a logical order: `FROM` (table selection), `WHERE` (filter conditions), and finally `SELECT` (columns to be retrieved).

4. Execution Plan

- The execution plan details how data will be accessed, processed, and stored in memory.
- It includes decisions about memory allocation, variable handling, and the use of algorithms, ensuring the query is executed efficiently.

Stored Procedures & Triggers

Stored Procedures

Stored procedures provide a structured, efficient, and secure method for managing data, especially in applications where data access and manipulation are frequent. Here's a detailed explanation of how stored procedures solve common data access issues and improve error handling:

Solving Data Access Problems with Stored Procedures

Problem 1: Redundant Execution and Processing Time

When a user repeatedly executes a query, SQL Server must perform all query processing steps each time—parsing, optimizing, and creating a query execution plan. This can be inefficient for large datasets.

Problem 2: Network Traffic

With lengthy queries, a lot of data has to be sent across the network, increasing traffic and slowing down performance.

Solution: Stored Procedures

A stored procedure stores the query execution plan after the first execution, avoiding repeated parsing and optimization. By simply sending `Getemp 4` across the network, the procedure significantly reduces network load and reuses the precompiled execution plan.

```
CREATE PROCEDURE Getemp @id INT AS BEGIN SELECT * FROM Employees WHERE SSN =
@id; END;
```

Benefits of Stored Procedures:

1. Performance Optimization

- **Reduced Network Traffic**: Instead of sending a long query to the server every time, only the stored procedure name and parameters need to be transmitted, which minimizes data transfer.
- **Execution Plan Reuse**: Stored procedures cache their execution plans in memory, so after the first call, subsequent calls can skip parsing, optimizing, and query tree formation, making execution faster.
- **Batch Execution**: Stored procedures can execute multiple SQL statements in one call, improving performance by reducing the number of round trips to the database.

2. Enhanced Security

- **Permission Control**: Permissions can be granted at the procedure level, restricting access to underlying tables and reducing the risk of unauthorized actions.
- **SQL Injection Prevention**: Stored procedures can limit exposure to SQL injection attacks by separating SQL code from user input.
- **Code Encryption**: Procedures can be encrypted using the `WITH ENCRYPTION` option, making it difficult for unauthorized users to view the source code.

3. Modularity and Reusability

- **Centralized Business Logic**: Stored procedures encapsulate business logic, enabling consistent data processing rules and calculations across different applications.
- **Reusability**: A single stored procedure can be called multiple times from different applications or modules, reducing duplicate code.

4. Improved Maintenance

- **Simplified Updates**: When a stored procedure is updated, changes are applied across all applications using that procedure, making it easy to manage centralized code updates.

- **Debugging Ease**: Procedures isolate database logic, allowing faster debugging and troubleshooting without impacting other application components.

5. Error Handling and Data Validation

- **Custom Error Messages**: Stored procedures can be programmed to detect issues (such as duplicate entries) and return specific error messages, which improves application reliability.
- **Transaction Management**: Procedures allow for complex transaction control (`BEGIN TRANSACTION`, `COMMIT`, and `ROLLBACK`) within the database, ensuring data integrity in multi-step operations.

6. Flexibility and Functionality

- **Dynamic SQL Execution**: Procedures can execute dynamic SQL, meaning SQL statements can be constructed and executed based on input parameters, allowing flexible, adaptable queries.
- **Default and Optional Parameters**: Stored procedures support default values and optional parameters, making them adaptable to various requirements without needing multiple versions.

7. Enhanced Data Abstraction

- **Data Encapsulation**: Procedures abstract complex queries and database structures, allowing applications to interact with the database without needing to understand table relationships or join structures.
- **Simplified API for Applications**: Applications interact with stored procedures rather than complex SQL, making the code easier to read, manage, and understand.

8. Consistent Data Management

- **Enforced Business Rules**: Centralizing business logic in stored procedures ensures data is processed consistently according to business rules, reducing the risk of inconsistent data across applications.
- **Avoiding Redundant Logic**: With all logic in one place, stored procedures prevent duplicating complex queries across different parts of an application.

1. **Reduced Network Load**: The procedure is called using a short command (e.g., `Getemp 4`), minimizing data sent over the network.
2. **Improved Performance**: On the first call, SQL Server compiles and saves the execution plan. On subsequent calls, it skips the parsing and optimization steps, making it faster.
3. **Memory Caching**: The server recognizes if a procedure has already been compiled by storing it in memory, thus speeding up repeated use.
4. **Modular Code**: Procedures can call functions, views, and other stored procedures, allowing for a flexible and modular design.
5. **Recompilation on Server Restart**: Procedures will recompile only after a server restart, ensuring they are always optimized.

Error Handling in Stored Procedures

Stored procedures offer a more controlled environment for handling errors, such as in an insertion operation with unique constraints (e.g., SSN in an employee table).

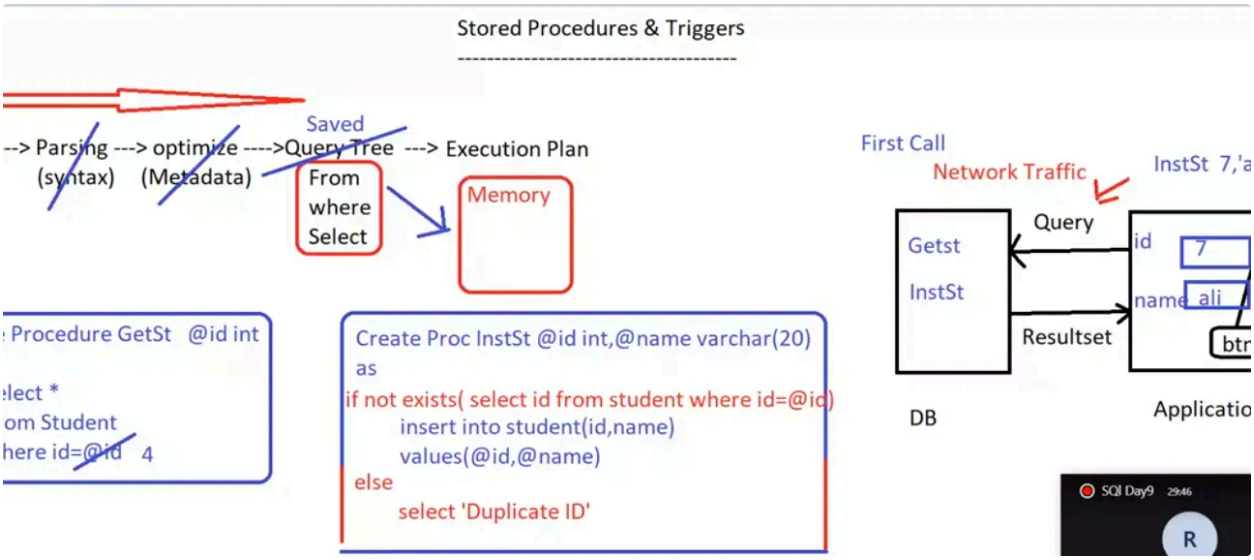
Example: Handling Duplicate Records with Stored Procedure

If a button in an app triggers an `INSERT` statement, such as `INSERT INTO Employee VALUES ('Amr', 'Abod', 7)`, it can throw an error if SSN `7` already exists. By using a stored procedure, you can manage this scenario gracefully with conditional logic.

```
CREATE PROCEDURE insertemp @id INT, @EmployeeName VARCHAR(20) AS BEGIN IF NOT EXISTS (SELECT * FROM Employees WHERE SSN = @id) INSERT INTO Employees (SSN, EmployeeName) VALUES (@id, @EmployeeName); ELSE SELECT 'Duplicated IDs'; END;
```

Benefits:

- **Error Prevention**: The `IF NOT EXISTS` clause checks for duplicates before inserting, avoiding SQL errors.
- **Controlled Responses**: When duplicates are found, a custom message (`'Duplicated IDs'`) is returned instead of a server error.
- **Reusability and Efficiency**: Procedures like this can be reused across the application, simplifying error handling and ensuring consistency.



Types of Stored Procedures

1. **Built-in Procedures**: These are provided by SQL Server for specific database management tasks.
    - **Examples**:
        - `sp_bindrule`: Binds a rule to a column or user-defined data type.
        - `sp_unbindrule`: Unbinds a rule from a column or user-defined data type.
        - `sp_helpconstraint`: Displays information about constraints on a table.
        - `sp_rename`: Renames a database object like a table or a column.
        - `sp_addtype`: Creates a new user-defined data type.

2. **User-defined Procedures**: Custom procedures created by users for specific business logic and data manipulation.
    - **Basic Example**:

```
CREATE PROCEDURE getemps AS SELECT * FROM Employee;
```

        - **Calling the Procedure**:
            - `EXECUTE getemps;` or `getemps;`
    - **Inserting Procedure Results into a Table**:

```
INSERT INTO table4(SSN, Fname) EXECUTE getemps;
```

    - **Dynamic Stored Procedure with Parameters**:

```
CREATE PROCEDURE getempbyGender @Gender VARCHAR(10) AS SELECT * FROM
Employees WHERE Gender = @Gender;
```

        - **Usage**: `getempbyGender 'M'`
    - **Stored Procedure with Conditional Insert**:

```
CREATE PROCEDURE insertemp1 @id INT, @name VARCHAR(20) AS IF NOT EXIS
TS (SELECT * FROM Employees WHERE id = @id AND name = @name) INSERT I
NTO Employees (id, name) VALUES (@id, @name); ELSE SELECT 'duplicated
values';
```

        - **Usage**: `insertemp1 1, 'Amr'`

- Error Handling with `TRY...CATCH`:

```
CREATE PROCEDURE insertemp2 @id INT, @name VARCHAR(20) AS BEGIN TRY I
NSERT INTO Employees (id, name) VALUES (@id, @name); END TRY BEGIN CA
TCH SELECT 'duplicated values'; END CATCH;
```

  - Usage: `insertemp2 2, 'Amr'`

- Procedures with Input Parameters and Defaults:

```
CREATE PROCEDURE getsum1 @x INT = 10, @y INT = 20 AS SELECT @x + @y;
```

  - Usage: `getsum1;` (returns 30 by default) and
  - `getsum1 @x = 13, @y = 16` (returns 29 Calling parameter by Name)
  - `getsum1 3, 11` (return 14 Calling parameter by Position)
  - `getsum1 33` (returns 53)

- Procedures with Output Parameters:

```
ALTER PROCEDURE getname1 @id INT, @name VARCHAR(20) OUTPUT AS SELECT
@name = name FROM omar WHERE id = @id;
```

  - Usage:

```
DECLARE @x VARCHAR(20); EXECUTE getname1 2, @x OUTPUT; SELECT @x;
```

- Procedures with Multiple Output Parameters:

```
ALTER PROCEDURE getname @id INT, @name1 VARCHAR(20) OUTPUT, @name2 VA
RCHAR(20) OUTPUT AS SELECT @name1 = name, @name2 = name FROM omar WHE
RE id = @id;
```

  - Usage:

```
DECLARE @x VARCHAR(20), @y VARCHAR(20); EXECUTE getname 2, @x OUT
PUT, @y OUTPUT; SELECT @x, @y;
```

- Input and Output Parameters Combined:

```
CREATE PROCEDURE getdata @id INT OUTPUT, @name VARCHAR(20) OUTPUT AS
SELECT @id = id, @name = name FROM Employees WHERE id = @id;
```

  - Usage:

```
DECLARE @x INT = 1, @y VARCHAR(20); EXECUTE getdata @x OUTPUT, @y
OUTPUT; SELECT @x, @y;
```

- Dynamic SQL Execution in a Procedure:

```
CREATE PROCEDURE getalldata @col VARCHAR(20), @tab VARCHAR(20) AS EXE
CUTE ('SELECT ' + @col + ' FROM ' + @tab);
```

  - Usage: `getalldata 'id', 'omar';`

- **Encrypting a Stored Procedure**:

```
CREATE PROCEDURE getalldata @col VARCHAR(20), @tab VARCHAR(20) WITH E
NCRYPTION AS EXECUTE ('SELECT ' + @col + ' FROM ' + @tab);
```

  - To prevent others from viewing the code, you can use `WITH ENCRYPTION`. After encrypting, commands like `sp_helptext 'getalldata'` will not reveal the code.

Triggers Overview

- **Triggers** are special types of stored procedures that automatically execute when specific database events occur, like `INSERT`, `UPDATE`, or `DELETE`.
- Unlike stored procedures, **triggers cannot accept parameters and cannot be directly called**; they are tied to specific events on tables or views.
- Triggers are categorized based on their **scope levels**, including table-level (the most common), schema-level, and server-level triggers.

Types of Table-Level Triggers

1. **AFTER Triggers**: Execute after the specified operation (e.g., after `INSERT`, `UPDATE`, or `DELETE`).
2. **INSTEAD OF Triggers**: Execute in place of the specified operation, providing custom logic instead of the default database action.

Examples and Usage of Triggers

1. AFTER INSERT Trigger

```
CREATE TRIGGER message ON Employees AFTER INSERT AS SELECT 'Welcom to Robot C
ompany'
```

- This trigger displays "god bless you" after inserting a new row into the `omar` table.

2. AFTER UPDATE Trigger

```
CREATE TRIGGER message1 ON Employees FOR UPDATE AS SELECT GETDATE()
```

- Executes the `GETDATE()` function, showing the current date and time whenever an update is made to `omar`.

3. INSTEAD OF DELETE Trigger

```
CREATE TRIGGER message2 ON Employees INSTEAD OF DELETE AS SELECT 'you can not
delete data ' + SUSER_NAME()
```

- Prevents deletion from `Employees`, instead displaying a message with the current user name.

4. Read-Only Table Trigger

```
CREATE TRIGGER message3 ON Employees INSTEAD OF INSERT, UPDATE, DELETE AS SEL
ECT 'Not allowed'
```

- Makes `Employees` read-only by blocking any insert, update, or delete operation, displaying "not allowed."

Trigger Functionalities

a. Update-Specific Functionality

```
CREATE TRIGGER message4 ON Employees FOR UPDATE AS IF UPDATE(name) SELECT 'al
lowed'
```

- Executes only if the `Employees` column is updated, showing "allowed" if so. Other updates produce no output.

b. Enable and Disable Triggers

- Use `ALTER TABLE` to **disable** or **enable** triggers without dropping them:

```
ALTER TABLE Employees DISABLE TRIGGER message3
```

c. Using the `INSERTED` and `DELETED` Tables

- SQL Server automatically creates two temporary tables within triggers:
  - `INSERTED`: Holds new data after `INSERT` or `UPDATE`.
  - `DELETED`: Holds old data before `DELETE` or `UPDATE`.

```
CREATE TRIGGER message5 ON Employees FOR UPDATE AS SELECT * FROM inserted --
New row data after update SELECT * FROM deleted -- Previous row data before u
pdate
```

d. Conditional DELETE Trigger

```
CREATE TRIGGER message6 ON Employees FOR DELETE AS IF FORMAT(GETDATE(), 'ddd
d') = 'Monday' BEGIN SELECT 'not allowed' --rollback INSERT INTO Employees SE
LECT * FROM deleted END
```

- Prevents deletion on Mondays, restoring data from `deleted` if a delete operation is attempted.

e. Tracking Changes in a History Table

- This trigger logs each update to a history table for audit purposes:

```
CREATE TABLE history ( _user VARCHAR(20), _date DATE, _oldid INT, _newid INT
) CREATE TRIGGER message7 ON Employees INSTEAD OF UPDATE AS IF UPDATE(id) BEG
IN DECLARE @old INT, @new INT, @date1 DATE = GETDATE(), @nam VARCHAR(20) = SU
SER_NAME() SELECT @old = id FROM deleted SELECT @new = id FROM inserted INSER
T INTO history VALUES(@nam, @date1, @old, @new) END
```

f. Using the `OUTPUT` Keyword in DML Statements

- The `OUTPUT` clause can capture modified rows and display information during runtime:

```
DELETE FROM Employees OUTPUT GETDATE(), deleted.name WHERE id = 7 UPDATE Empl
oyees SET age = 26 OUTPUT GETDATE(), deleted.name WHERE id = 5 INSERT INTO Em
ployees OUTPUT 'Hello new partner' VALUES (6, 'vesta', 21)
```

Triggers are special types of stored procedures in SQL that automatically execute in response to specific events on a table or view, such as `INSERT`, `UPDATE`, or `DELETE` operations.

1. Automatic Execution

- Triggers automatically execute in response to specific database events, ensuring that certain actions are always performed without requiring explicit calls.

2. Data Integrity and Validation

- Triggers can enforce data integrity rules by validating data before it's inserted or updated. For example, they can prevent invalid data entries based on specific conditions.

3. Auditing Changes

- Triggers can log changes made to a table, creating an audit trail for data modifications. This is particularly useful for tracking who made changes and when.

4. Enforcing Business Rules

- Triggers can enforce complex business rules that go beyond standard constraints. For example, they can prevent deletion of certain rows based on related data.

5. Cascading Actions

- Triggers can facilitate cascading actions, such as automatically updating or deleting related records in other tables when changes occur.

6. Preventing Invalid Operations

- Triggers can prevent certain operations, such as deletions or updates, based on specific conditions (e.g., preventing deletion of records if they are referenced by other tables).

7. Synchronous Processing

- Triggers allow for synchronous processing of data changes, ensuring that related actions occur immediately in response to the triggering event.

8. Support for Complex Logic

- Triggers can implement complex logic that may not be easily expressed using constraints alone, allowing for more flexibility in database operations.

9. Enhancing Performance

- By handling certain operations at the database level through triggers, applications may achieve better performance and reduced network traffic since fewer client-server round trips are required.

10. Custom Notifications

- Triggers can be used to send notifications or execute custom actions (such as calling stored procedures or sending alerts) in response to data changes.

11. Consistent Application Logic

- By centralizing business logic within triggers, organizations can ensure consistent enforcement of rules across multiple applications interacting with the same database.

12. Conditional Execution

- Triggers can perform different actions based on the specifics of the data change, allowing for tailored responses to various scenarios.

13. Supporting Historical Data

- Triggers can be used to archive old data or keep historical records in separate tables whenever changes are made, which aids in data analysis and reporting.

14. Dynamic Changes

- Triggers can dynamically modify the behavior of database operations based on the context or state of the data, allowing for adaptable database management.

Using **XML in SQL Server** allows for transferring and transforming data across different database systems, such as from Oracle to SQL Server or vice versa. Below is a guide to using XML features in SQL Server, focusing on converting tables to XML format and XML data to table format.

1. Converting a Table to XML (Using `FOR XML`)

- `FOR XML` allows data to be formatted as XML, useful for data exchange and export.

a. FOR XML RAW

- Generates XML output with each row as a `<row>` element, or a specified name.

```
SELECT * FROM Employees FOR XML RAW -- Basic raw format with default <row> el
ements SELECT * FROM Employees FOR XML RAW('member') -- Uses <member> as the
element name for each row SELECT * FROM Employees FOR XML RAW('member'), ELEM
ENTS -- Each column is displayed as an XML element SELECT * FROM Employees FO
R XML RAW('member'), ELEMENTS, ROOT('Company_members') -- Wraps in root eleme
nt <omar_members> SELECT * FROM Employees FOR XML RAW('member'), ELEMENTS XSI
NIL, ROOT('Company_members') -- Includes nulls as empty elements
```

b. FOR XML AUTO

- Automatically organizes XML elements based on table hierarchy, beneficial in `JOIN` scenarios.

```
SELECT Employee.SSN, Employee.Fname, Departments.Dname FROM Employee INNER JO
IN Departments ON Employee.Dno = Departments.Dnum FOR XML AUTO, ELEMENTS XSIN
IL, ROOT('Company_members')
```

c. FOR XML PATH

- Customizes XML by specifying elements and attributes.

```
SELECT SSN AS "@emp_id", -- `SSN` becomes an attribute in <Employee> tag Fnam
e AS "EmployeeName/FirstName", -- Fname inside <FirstName> within <EmployeeNa
me> Dno AS "EmployeeName/@EmpDno", -- Dno as an attribute in <EmployeeName> L
name AS "EmployeeName/LastName", -- Lname inside <LastName> within <EmployeeN
ame> Address AS "Address" -- Address as a separate tag FROM Employee FOR XML
PATH('Employee') -- Each row is wrapped in <Employee>
```

d. FOR XML explicit

Now, we want to create XML output that has a structure where each employee is an `<Employee>` element containing their details. We can use `FOR XML EXPLICIT` like this:

```
SELECT 1 AS Tag, NULL AS Parent, EmployeeID AS [Employee!1], FirstName AS [Em
ployee!2], LastName AS [Employee!3], Department AS [Employee!4] FROM Employee
s ORDER BY EmployeeID FOR XML EXPLICIT;
```

Explanation of the Query

- **Tag**: This is used to define the level of the hierarchy in the XML output. In this case, all employees are on the same level, so we set it to `1`.
- **Parent**: This specifies the parent element for the current row. Since there's no hierarchy here, it's set to `NULL`.
- **Element Specification**:
  - `Employee!1`, `Employee!2`, etc. are used to define the elements and attributes. The `!` is a separator that indicates the element is part of the `<Employee>` tag.
- **ORDER BY**: This sorts the output by `EmployeeID`.

Resulting XML Output:

```
<Employee> <EmployeeID>1</EmployeeID> <FirstName>John</FirstName> <LastName>D
oe</LastName> <Department>Sales</Department> </Employee> <Employee> <Employee
ID>2</EmployeeID> <FirstName>Jane</FirstName> <LastName>Smith</LastName> <Dep
artment>HR</Department> </Employee> <Employee> <EmployeeID>3</EmployeeID> <Fi
rstName>Mike</FirstName> <LastName>Johnson</LastName> <Department>IT</Departm
ent> </Employee>
```

Note

The `FOR XML EXPLICIT` method is more complex and requires you to manage the structure manually. It is generally recommended to use `FOR XML AUTO` or `FOR XML PATH` for most scenarios due to their simpler syntax and flexibility, unless you need very specific XML structures.

2. Converting XML to Table (Using `OPENXML`)

To convert XML data into table format:

1. **Declare an XML Variable**

```
DECLARE @docs XML = '<Employees> <Employee emp_id="112233"> <EmployeeName
><FirstName>omar</FirstName></EmployeeName> <EmployeeVame EmpDno="10" />
<EmployeeName><LastName>Ali</LastName></EmployeeName> <Address>15 Ali fah
my St.Giza</Address> </Employee> <Employee emp_id="112299"> <EmployeeName
><FirstName>mostafa</FirstName></EmployeeName> <EmployeeVame EmpDno="10"
/> <EmployeeName><LastName>nasser</LastName></EmployeeName> <Address>Assu
it</Address> </Employee> <Employee emp_id="123456"> <EmployeeName><FirstN
ame>aya</FirstName></EmployeeName> <EmployeeVame EmpDno="10" /> <Employee
Name><LastName>Sobhy</LastName></EmployeeName> <Address>38 Abdel Khalik T
harwat St. Downtown.Cairo</Address> </Employee> </Employees>';
```

2. **Create a Document Handle**
   - Initialize a document handle using `sp_xml_preparedocument` for easy parsing.

```
DECLARE @hdocs INT; -- Handle to the XML document EXEC sp_xml_preparedocu
ment @hdocs OUTPUT, @docs;
```

3. **Process the XML Document**
   - Use `OPENXML` with XPath to extract elements and attributes as table columns.

```
SELECT * FROM OPENXML(@hdocs, '//Employee') -- XPath to target Employee n
odes WITH ( Employeeid INT '@emp_id', -- `@emp_id` attribute becomes `Emp
loyeeid` column Lname VARCHAR(20) 'EmployeeName/LastName', -- LastName el
ement becomes `Lname` column Address VARCHAR(20) 'Address', -- Address el
ement employeedep INT 'EmployeeVame/@EmpDno' -- EmpDno attribute from Emp
loyeeVame )
```

4. **Release Document Handle**
   - Clean up resources by removing the XML document from memory.

```
EXEC sp_xml_removedocument @hdocs;
```