DAY 7

SQL, Variables, If, While, functions (Day 7)

Variables in SQL Server

- **Local variables**: Used within stored procedures, functions, or batches to hold temporary data.
- **Global variables**: Predefined by SQL Server and provide information about the system state (e.g., server name, row counts, errors).
- **Dynamic SQL**: Allows execution of SQL code dynamically based on variable input.

1. Local Variables:

Local variables are used to store data temporarily in a session, procedure, or batch. You can define and manipulate local variables within a batch, function, or stored procedure.

- **Batch**: A block of code that is highlighted and executed together.
- **Function**: Variables can be used inside user-defined functions.
- **Stored Procedure**: Variables are commonly used within stored procedures to hold temporary data.

- To define a variable:

```
DECLARE @x INT; -- Declares a variable @x of type INT. The initial value
is NULL.
```

- To assign a value to a variable:
  - Using `SET`:

```
SET @x = 10; -- Assigns the value 10 to the variable @x.
```

  - Using `SELECT`:

```
SELECT @x = 100; --Assigns the value 100 to the variable @x.
```

  - From a query result:

```
SELECT @x = (SELECT AVG(Salary) FROM Employees WHERE EmployeeID = 1);
-- Fetches the average salary of EmployeeID 1 and assigns it to @x.
```

  - During an `UPDATE` statement:

```
UPDATE Employees SET EmployeeName = 'Omar', @x = Salary WHERE Employe
eID = 9; -- Updates the EmployeeName of EmployeeID 9 to 'Omar' and as
signs their Salary to @x.
```

- To print the value of a variable:

```
SELECT @x; -- Displays the value stored in @x.
```

Examples of Local Variables:

1. **Calculate and assign the average salary to a variable**:

```
USE CompanyDB; -- Switch to the CompanyDB database. -- ⇒ 1, 2, and 3 mus
t excute together 1⇒ DECLARE @x INT; -- Declare a variable @x of type IN
T. 2⇒ SELECT @x = AVG(Salary) FROM Employees; --Calculate the average sa
lary of all employees and assign it to @x. 3⇒ SELECT @x; -- Display the
value of @x.
```

2. **Handling cases where a query returns no result**:
   If a query doesn't return a result, the variable retains its previous value:

```
DECLARE @x INT = 100; -- Declare @x with an initial value of 100. SELECT
@x = Salary FROM Employees WHERE EmployeeID = 121312312; -- No Employee w
ith this ID, so the variable remains 100. SELECT @x; -- Output will still
be 100.
```

3. **Handling multiple row results**:
   When a query returns multiple rows, the variable is assigned the last value:

```
DECLARE @x INT; -- Declare @x without an initial value. SELECT @x = Salar
y FROM Employees WHERE EmployeeID > 1; -- If multiple employees are retur
ned, @x will hold the last Salary. SELECT @x; -- Outputs the last salary
from the result set.
```

4. **Using multiple variables**:

```
DECLARE @EmpSalary INT, @EmpName VARCHAR(50); -- Declare two variables: @
x (int) and @y (string). SELECT @EmpSalary = Salary, @EmpName = EmployeeN
ame FROM Employees WHERE EmployeeID = 1; -- Assign the salary and first n
ame of EmployeeID 1 to @x and @y, respectively. SELECT @EmpSalary, @EmpNa
me; -- Display the values of @x and @y.
```

5. Using variables in an `UPDATE` statement:

```
DECLARE @DeptID INT; -- Declare a variable @x. UPDATE Employees SET Emplo
yeeName = 'Gazal', @DeptID = DepartmentID WHERE EmployeeID = 1; -- Update
the EmployeeName of EmployeeID 1 and assign their DepartmentID to @x. SEL
ECT @DeptID; -- Display the value of @x.
```

6. Using table variables:
   Table variables can be used to temporarily store multiple rows in memory:

```
DECLARE @SalaryTable TABLE (Salary INT); -- Declare a table variable @Sal
aryTable with a Salary column of type INT. INSERT INTO @SalaryTable SELEC
T Salary FROM Employees WHERE EmployeeID > 1; -- Insert the salaries of e
mployees with ID > 1 into the table variable. SELECT * FROM @SalaryTable;
-- Select all rows from the table variable @SalaryTable.
```

7. Table variable with multiple columns:

```
DECLARE @Emp_ID_Salary_table TABLE (EmployeeID INT, Salary INT); -- Decla
re a table variable @Emp_ID_Salary_table with EmployeeID and Salary colum
ns. INSERT INTO @Emp_ID_Salary_table SELECT EmployeeID, Salary FROM Emplo
yees WHERE EmployeeID > 1; -- Insert EmployeeID and Salary into the table
variable. SELECT * FROM @Emp_ID_Salary_table; -- Select all rows from the
table variable @Emp_ID_Salary_table.
```

8. Using variables with the `TOP` clause:

```
DECLARE @Top_var INT = 4; -- Declare @Top_var with an initial value of 4.
SELECT TOP (@Top_var) * FROM Employees; -- Select the top 4 rows from the
Employees table.
```

9. **Dynamic SQL Execution**:
   Dynamic SQL allows you to execute SQL queries constructed at runtime:

```
DECLARE @x INT = 4; -- Declare @x with an initial value of 4. EXEC ('SELE
CT TOP (' + CAST(@x AS VARCHAR) + ') * FROM Employees'); -- Executes a qu
ery that selects the top 4 rows from the Employees table.
```

10. **Dynamic SQL with column and table names**:
    This demonstrates how you can use variables to define column and table names dynamically:

```
DECLARE @column VARCHAR(50) = 'Salary', @table VARCHAR(50) = 'Employees';
-- Declare variables for the column and table names. EXEC ('SELECT ' + @c
olumn + ' FROM ' + @table); -- Executes a query selecting the Salary colu
mn from the Employees table
```

The command you're using:

```
EXEC('SELECT * FROM Employees');
```

This is an example of **dynamic SQL** in SQL Server. Dynamic SQL allows you to build and execute SQL statements at runtime as a string.

Explanation:

- `EXEC` (or `EXECUTE`): Executes a dynamically constructed SQL query or stored procedure. The SQL query is passed as a string.
- `'SELECT * FROM Employees'`: This string contains the actual SQL query to be executed, in this case, selecting all rows from the `Employees` table.

Use Cases for Dynamic SQL:

- **Conditional Execution**: You might use dynamic SQL when you need to build queries dynamically based on variable inputs (e.g., table names, column names, or filtering conditions).

- **Flexibility**: It is useful when you don't know the exact query at compile time but need to decide it at runtime.

Security Consideration:

Dynamic SQL can lead to **SQL injection attacks** if not used properly, especially when user inputs are directly embedded in the query string. Always validate and sanitize inputs to avoid security vulnerabilities. Using **parameterized queries** is a better alternative to mitigate this risk.

Example with Dynamic Column Name:

If you want to dynamically specify the column name, here's an example:

```
DECLARE @column VARCHAR(50) = 'EmployeeName'; EXEC('SELECT ' + @column + ' FR
OM Employees');
```

---

2. Global Variables:

Global variables in SQL Server provide system information and cannot be declared or assigned manually. They always start with `@@`.

- **Examples of Global Variables**:

1. **Server name**:

```
SELECT @@SERVERNAME; -- Returns the name of the server.
```

2. **Row count of the last operation**:

```
UPDATE Employees SET Salary = Salary + 1; -- Increases the salary of all
employees by 1. SELECT @@ROWCOUNT; -- Returns the number of rows affected
by the last operation (in this case, the update).
```

3. **SQL Server version**:

```
SELECT @@VERSION; -- Returns the version of SQL Server you're using.
```

4. **Error code of the last operation**:
   If the last operation resulted in an error, the global variable `@@ERROR` returns the error number:

```
UPDATE Employeess -- Intentional typo in the table name to generate an er
ror. SET Salary += 1; SELECT @@ERROR; -- Returns the error code for the l
ast operation (e.g., 208 for an invalid table name).
```

5. **Last identity value inserted**:

```
INSERT INTO Employees (FirstName, LastName, Email, HireDate, JobTitle, Sa
lary) VALUES ('John', 'Doe', 'john.doe@example.com', '2024-01-01', 'Manag
er', 5000); -- Inserts a new employee. SELECT @@IDENTITY; -- Returns the
last inserted identity value (EmployeeID) from the Employees table.
```

Example combining local and global variables:

```
DECLARE @x VARCHAR(50); -- Declare a local variable @x of type VARCHAR. SET @
x = @@VERSION; -- Assign the value of the global variable @@VERSION to @x. SE
LECT @x; -- Display the value of @x (which now holds the SQL Server version).
```

Security Consideration for Dynamic SQL

Dynamic SQL, while flexible, can introduce significant security risks, especially **SQL injection attacks**, if not handled correctly. This occurs when malicious users manipulate the input to execute unintended SQL commands, potentially gaining unauthorized access to or altering the database.

Example of SQL Injection:

Suppose a dynamic SQL query includes unsanitized user input:

```
DECLARE @userID VARCHAR(50) = '1 OR 1=1'; EXEC('SELECT * FROM Employees WHERE
EmployeeID = ' + @userID);
```

This would generate the query:

```
SELECT * FROM Employees WHERE EmployeeID = 1 OR 1=1;
```

Here, the condition `1=1` is always true, so it retrieves all rows in the `Employees` table, bypassing the intent of the query.

Mitigating SQL Injection in Dynamic SQL

1. **Avoid Directly Embedding User Inputs:**
   Never directly concatenate user input into dynamic SQL. This exposes the system to malicious injections.
2. **Use Parameterized Queries:**
   Whenever possible, use parameterized queries to safely inject variables into SQL statements, preventing malicious data from being treated as executable code.

Example of a Safe Parameterized Query:

Instead of constructing the query dynamically with string concatenation, use parameters:

```
DECLARE @userID INT = 1; EXEC sp_executesql N'SELECT * FROM Employees WHERE E
mployeeID = @userID', N'@userID INT', @userID;
```

- `sp_executesql`: A system-stored procedure that executes a SQL statement with parameters. This allows for safely passing values without risking SQL injection.
- **Parameters**: The variable `@userID` is treated as data, not code, ensuring that any user input is not executed as part of the SQL query.

Other Practices to Improve Security:

1. **Input Validation**: Always validate user inputs to ensure they conform to expected formats (e.g., numeric values, valid email addresses).
2. **Least Privilege Principle**: Ensure that the user executing the dynamic SQL has only the minimal required database privileges, reducing the risk of damage in case of an attack.
3. **Escaping Inputs**: If for some reason dynamic SQL must be used, ensure that all user inputs are properly escaped or sanitized.

The query `SELECT * FROM Employees WHERE EmployeeID = 1 OR 1=1;` is a typical example of **SQL injection** vulnerability.

Query Explanation:

- `SELECT * FROM Employees`: This part of the query retrieves all columns from the `Employees` table.
- `WHERE EmployeeID = 1`: This part is supposed to filter the records to return only the row where the `EmployeeID` is 1.
- `OR 1=1`: This condition essentially makes the query unsafe. Here's why:
  - `1=1` is a logical expression that is always **true**.
  - The `OR` operator means that if either of the conditions is true, the row will be selected.
  - So, no matter what the value of `EmployeeID` is, the expression `1=1` will always return `TRUE`. Therefore, the query will **ignore the actual filtering condition** (`EmployeeID = 1`) and will return **all rows** from the `Employees` table.

Why is This a Problem?

- **SQL Injection Vulnerability**: This kind of query can occur if an application allows user input without proper sanitization or validation. If an attacker can inject `1=1` into the query, it essentially bypasses the filtering conditions and returns all rows from the table.
    - For example, in a login form, if the query were:
      The query would bypass authentication entirely and return all users, allowing an attacker unauthorized access to the system.

```
SELECT * FROM Users WHERE Username = 'user' AND Password = 'pass' OR
1=1;
```

Impact:

- **Data Leakage**: An attacker could retrieve confidential information from the entire table when they were only supposed to see specific rows.
- **Security Breach**: If this kind of query is used in authentication systems, an attacker could gain access to sensitive user accounts.

Proper Query:

The query should only return the row where `EmployeeID = 1`:

```
SELECT * FROM Employees WHERE EmployeeID = 1;
```

Prevention:

To avoid such vulnerabilities, you should always:

1. **Use parameterized queries** or **prepared statements**.
2. **Validate and sanitize user input**.
3. Ensure that dynamic SQL queries are constructed securely to prevent unauthorized data access.

Control of Flow Statements in SQL Server

1. IF Statement

The `IF` statement checks a condition and executes a block of code based on whether the condition is `TRUE` or `FALSE`.

Example:

```
USE CompanyDB; DECLARE @NumRows INT; UPDATE Employees SET Salary += 1 WHERE E
mployeeID > 5; SELECT @NumRows = @@ROWCOUNT; IF @NumRows > 0 SELECT 'There ar
e affected rows'; ELSE SELECT 'There are no affected rows';
```

- Here, `@@ROWCOUNT` captures the number of rows affected by the `UPDATE` statement. If rows are updated (`@NumRows > 0`), it outputs "There are affected rows"; otherwise, it outputs "There are no affected rows."

2. BEGIN & END

`BEGIN` and `END` mark a block of SQL code to be executed together, often used in `IF` or `WHILE` statements.

Example:

```
USE CompanyDB; DECLARE @NumRows INT; UPDATE Employees SET Salary += 1 WHERE E
mployeeID > 5; SELECT @NumRows = @@ROWCOUNT; IF @NumRows > 0 BEGIN SELECT 'Th
ere are affected rows'; SELECT 'This is the second message for affected row
s'; END ELSE BEGIN SELECT 'There are no affected rows'; END
```

- The `BEGIN...END` ensures multiple statements are executed as a block inside the `IF` or `ELSE`.

3. IF EXISTS / IF NOT EXISTS

These are used to check whether a certain condition (such as the existence of a record or table) is met before executing a statement.

Example 1: **IF EXISTS**

```
IF EXISTS (SELECT name FROM sys.tables WHERE name = 'Employees') SELECT 'Tabl
e exists'; ELSE CREATE TABLE Employees ( EmployeeID INT, Name VARCHAR(50) );
```

- This checks if the `Employees` table exists. If it does, it prints "Table exists"; otherwise, it creates the table.

Example 2: **IF NOT EXISTS**

```
IF NOT EXISTS (SELECT DepartmentID FROM Employees WHERE DepartmentID = 40) DE
LETE FROM Departments WHERE DepartmentID = 40; -- Safe delete ELSE SELECT 'De
partment 40 has relation';
```

- This deletes the department if no employees are linked to it, otherwise it displays a message.

4. TRY...CATCH (Exception Handling)

This is used to handle errors that occur during SQL execution.

Example:

```
BEGIN TRY DELETE FROM Departments WHERE DepartmentID = 10; -- Safe delete END
TRY BEGIN CATCH SELECT 'Department 10 has a relation'; SELECT ERROR_LINE(), E
RROR_NUMBER(), ERROR_MESSAGE(); END CATCH;
```

- If the `DELETE` fails due to a foreign key constraint, the `CATCH` block is executed, showing the error details.

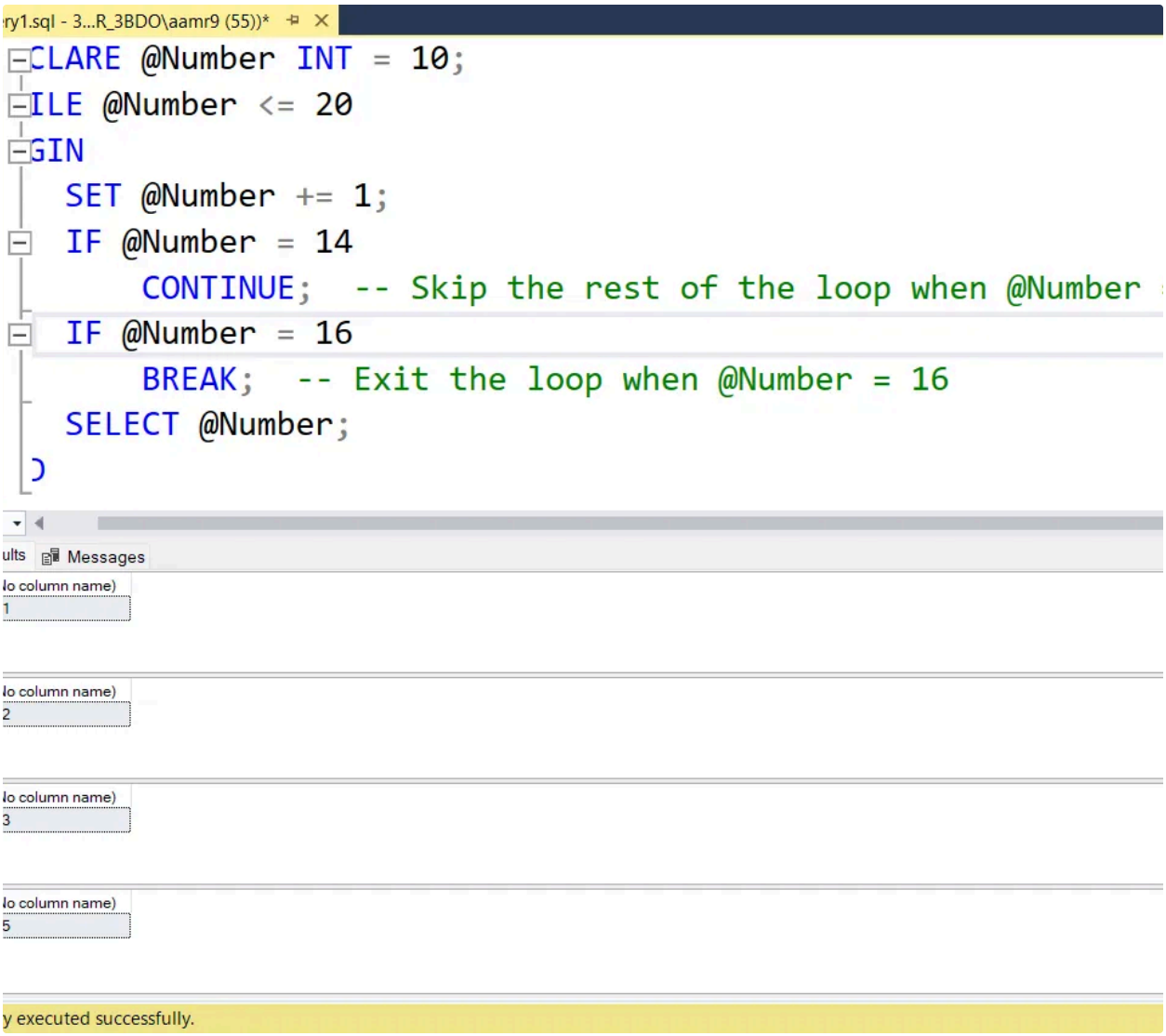
5. WHILE Loop

The `WHILE` loop repeatedly executes a block of code as long as a condition is `TRUE`.

Example:

```
DECLARE @Number INT = 10; WHILE @Number <= 20 BEGIN SET @Number += 1; IF @Num
ber = 14 CONTINUE; -- Skip the rest of the loop when @Number = 14 IF @Number
= 16 BREAK; -- Exit the loop when @Number = 16 SELECT @Number; END
```

- The `WHILE` loop increments `@Number` by 1 in each iteration. `CONTINUE` skips to the next loop cycle if `@Number = 14`, and `BREAK` stops the loop if `@Number = 16`.



6. CASE Statement

`CASE` allows you to apply conditional logic to columns in an `UPDATE` or `SELECT` statement.

Example:

```
UPDATE Employees SET Salary = CASE WHEN Salary <= 800 THEN Salary * 1.2 WHEN
Salary <= 1500 THEN Salary * 1.3 WHEN Salary > 1500 THEN Salary * 1.4 ELSE 0
END;
```

- This example updates employee salaries based on conditions.

7. IIF Function

`IIF` is a shorthand for `IF...ELSE` in a `SELECT` query.

Example:

```
SELECT EmployeeName, IIF(Salary >= 1600, 'High', 'Low') AS SalaryLevel FROM E
mployees;
```

- This assigns 'High' or 'Low' to each employee depending on their salary.

8. CHOOSE Function

`CHOOSE` selects a value from a list based on an index.

Example:

```
SELECT CHOOSE(2, 'Omar', 'Nasr', 'Ali') AS Name; -- Returns 'Nasr'
```

9. WAITFOR

`WAITFOR` delays execution for a specified amount of time.

Example:

```
WAITFOR DELAY '00:00:10'; -- Waits for 10 seconds
```

Batch, Script, and Transaction in SQL Server

- **Batch**: A batch is a collection of SQL statements executed together. Multiple SQL commands can be grouped and run in a single execution.
- **Script**: A script is a collection of SQL queries, but it can't execute conflicting statements together (like `CREATE TABLE` and `DROP TABLE`). You need to use the `GO` keyword to separate them.
- **Transaction**: A transaction ensures that a series of operations either all succeed or all fail. Use `BEGIN TRANSACTION`, `COMMIT`, and `ROLLBACK` to control transaction behavior.

Batch

A **batch** is a collection of SQL statements that are sent to the SQL Server for execution as a single unit. When a batch is executed, all the statements within it are processed together.

- **Example**:

```
CREATE TABLE Employees ( EmployeeID INT PRIMARY KEY, Name VARCHAR(50) ); GO D
ROP TABLE Employees; -- This will execute after the previous batch
```

- The `GO` keyword indicates the end of a batch. The statements before `GO` are executed before any that follow it.

Script

A **script** is a collection of SQL queries that cannot be run together due to their nature (like creating or dropping tables). To separate these types of statements, the `GO` keyword is used.

- **Example**:

```
CREATE RULE SalaryRule AS CHECK (Salary > 0); GO SP_BINDRULE Employees.Salar
y, SalaryRule;
```

- Scripts often include various operations that need to be executed sequentially but in different batches.

Transaction

A **transaction** ensures that a series of SQL statements are executed as a single unit. If any statement fails, the entire transaction can be rolled back, ensuring data integrity.

Types of Transactions

1. **Implicit Transaction**:
   - Every `INSERT`, `UPDATE`, or `DELETE` statement operates as its own transaction. SQL Server automatically wraps each of these operations in a transaction (i.e., it starts a transaction, commits it, or rolls it back).
2. **Explicit Transaction**:
   - You can define the beginning and end of a transaction using `BEGIN TRANSACTION`, followed by `COMMIT` or `ROLLBACK`.

Example of Transactions

1. **Creating Tables and Inserting Data**:

```
USE CompanyDB; CREATE TABLE Parent ( PID INT PRIMARY KEY ); CREATE TABLE Chil
d ( CID INT REFERENCES Parent(PID) ); INSERT INTO Parent VALUES (1); INSERT I
NTO Parent VALUES (2); INSERT INTO Parent VALUES (3); INSERT INTO Parent VALU
ES (4);
```

1. **Using Explicit Transactions**:

```
BEGIN TRANSACTION; INSERT INTO Child VALUES (1); INSERT INTO Child VALUES
(2); INSERT INTO Child VALUES (3); ROLLBACK;
```

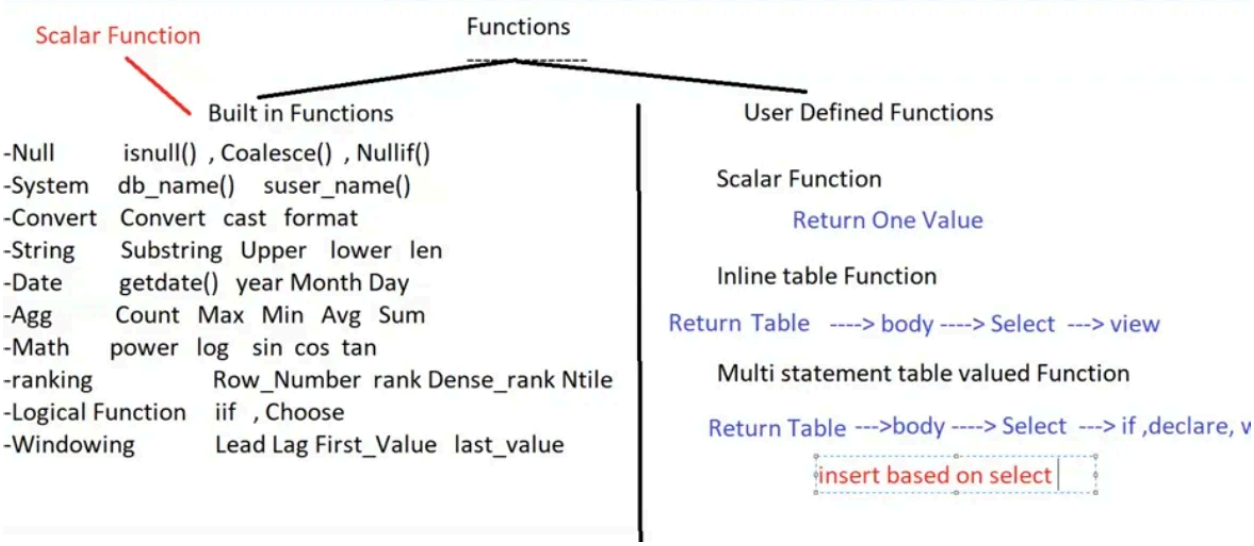
- In this example, since a `ROLLBACK` is executed, none of the inserted values are added to the `Child` table. The changes are reverted.

1. **Handling Errors with TRY...CATCH**:

```
BEGIN TRY BEGIN TRANSACTION; -- Start a transaction INSERT INTO Child VALUES (1); -- First insert INSERT INTO Child VALUES (6); -- Second insert INSERT INTO Child VALUES (2); -- This may cause a primary key violation COMMIT; -- If all inserts are successful, commit the transaction END TRY BEGIN CATCH ROLLBACK; -- If an error occurs, roll back the transaction SELECT ERROR_MESSAGE() AS ErrorMessage; -- Display the error message END CATCH;
```

- In this case, if the insert into the `Child` table fails (due to a primary key violation, for instance), the `CATCH` block executes, rolling back any changes made during the transaction.

SQL Functions: Built in functions



1. NULL Functions

`ISNULL`

Returns a specified replacement value if the expression is NULL.

```
SELECT Fname, ISNULL(PhoneNumber, 'No Phone') AS Phone FROM Employee;
```

`COALESCE`

Returns the first non-NULL value in the list of expressions.

```
SELECT Fname, COALESCE(Email, 'No Email', Lname) AS Contact FROM Employee;
```

`NULLIF`

Returns NULL if the two expressions are equal.

```
SELECT Fname, NULLIF(Salary, 0) AS AdjustedSalary FROM Employee;
```

---

2. System Functions

`DB_NAME()`

Returns the name of the current database.

```
SELECT DB_NAME() AS CurrentDatabase;
```

`SUSER_NAME()`

Returns the login name of the current user.

```
SELECT SUSER_NAME() AS CurrentUser;
```

---

3. Conversion Functions

`CONVERT`

Converts an expression from one data type to another.

```
SELECT Fname, CONVERT(VARCHAR(20), HireDate, 101) AS HireDateFormatted FROM E
mployee;
```

CAST

Similar to CONVERT, but ANSI SQL compliant.

```
SELECT Fname, CAST(Salary AS DECIMAL(10, 2)) AS SalaryDecimal FROM Employee;
```

FORMAT

Formats a value based on a specified format.

```
SELECT Fname, FORMAT(Salary, 'C', 'en-US') AS SalaryFormatted FROM Employee;
-- Formats salary as currency
```

---

4. String Functions

SUBSTRING

Extracts a substring from a string.

```
SELECT Fname, SUBSTRING(Lname, 1, 3) AS LnamePrefix FROM Employee;
```

UPPER

Converts a string to uppercase.

```
SELECT UPPER(Fname) AS UpperFname FROM Employee;
```

`LOWER`

Converts a string to lowercase.

```
SELECT LOWER(Lname) AS LowerLname FROM Employee;
```

`LEN`

Returns the length of a string.

```
SELECT Fname, LEN(Fname) AS FnameLength FROM Employee;
```

---

5. Date Functions

`GETDATE()`

Returns the current date and time.

```
SELECT GETDATE() AS CurrentDateTime;
```

`YEAR`, `MONTH`, `DAY`

Extracts the year, month, and day from a date.

```
SELECT Fname, YEAR(HireDate) AS HireYear, MONTH(HireDate) AS HireMonth, DAY(H
ireDate) AS HireDay FROM Employee;
```

---

6. Aggregate Functions

`COUNT`

Counts the number of rows that match a specified condition.

```sql
SELECT COUNT(*) AS TotalEmployees FROM Employee;
```

`MAX`

Returns the maximum value.

```sql
SELECT MAX(Salary) AS MaxSalary FROM Employee;
```

`MIN`

Returns the minimum value.

```sql
SELECT MIN(Salary) AS MinSalary FROM Employee;
```

`AVG`

Returns the average of a numeric column.

```sql
SELECT AVG(Salary) AS AverageSalary FROM Employee;
```

`SUM`

Returns the total sum of a numeric column.

```sql
SELECT SUM(Salary) AS TotalSalaries FROM Employee;
```

---

7. Mathematical Functions

`POWER`

Raises a number to a specified power.

```
SELECT Fname, POWER(Salary, 2) AS SalarySquared FROM Employee;
```

`LOG`

Returns the logarithm of a number.

```
SELECT Fname, LOG(Salary) AS SalaryLog FROM Employee;
```

`SIN`, `COS`

Returns the sine and cosine of an angle (in radians).

```
SELECT Fname, SIN(Salary) AS SalarySine FROM Employee;
```

---

8. Ranking Functions

`ROW_NUMBER`

Assigns a unique number to each row based on the specified order.

```
SELECT Fname, ROW_NUMBER() OVER (ORDER BY Salary DESC) AS RowNum FROM Employe
e;
```

`DENSE_RANK`

Assigns ranks to rows in a result set with no gaps.

```
SELECT Fname, DENSE_RANK() OVER (ORDER BY Salary DESC) AS DenseRank FROM Empl
oyee;
```

`RANK`

Similar to DENSE_RANK but allows gaps in ranking values.

```
SELECT Fname, RANK() OVER (ORDER BY Salary DESC) AS Rank FROM Employee;
```

---

9. Logical Functions

`IIF`

Returns one of two values based on the evaluation of a Boolean expression.

```
SELECT Fname, IIF(Salary >= 1600, 'High', 'Low') AS SalaryStatus FROM Employe
e;
```

`CHOOSE`

Returns the item at the specified index from a list of values.

```
SELECT CHOOSE(2, 'Finance', 'HR', 'IT', 'Sales') AS Department; -- Returns 'H
R'
```

---

10. Windowing Functions

`LEAD`

Returns the value of a column from a subsequent row.

```
SELECT Fname, LEAD(Salary) OVER (ORDER BY Salary) AS NextSalary FROM Employe
e;
```

`LAG`

Returns the value of a column from a previous row.

```
SELECT Fname, LAG(Salary) OVER (ORDER BY Salary) AS PrevSalary FROM Employee;
```

`FIRST_VALUE`

Returns the first value in an ordered set.

```
SELECT Fname, FIRST_VALUE(Salary) OVER (ORDER BY Salary) AS FirstSalary FROM
Employee;
```

`LAST_VALUE`

Returns the last value in an ordered set.

```
SELECT Fname, LAST_VALUE(Salary) OVER (ORDER BY Salary ROWS BETWEEN UNBOUNDED
PRECEDING AND UNBOUNDED FOLLOWING) AS LastSalary FROM Employee;
```

1. Using LAG and LEAD

- `LAG` retrieves the value of a specified column from the previous row within the result set.
- `LEAD` retrieves the value of a specified column from the next row.

Example: Get Previous and Next Employee Names Based on Salary

```
SELECT Lname, Salary, PrevLname = LAG(Lname) OVER (ORDER BY Salary), NextLnam
e = LEAD(Lname) OVER (ORDER BY Salary) FROM Employee;
```

2. LAG and LEAD with Salary

Example: Get Previous and Next Salaries

```
SELECT Lname, Salary, PrevSalary = LAG(Salary) OVER (ORDER BY Salary), NextSa
lary = LEAD(Salary) OVER (ORDER BY Salary) FROM Employee;
```

3. Using Subquery with LAG and LEAD

Example: Get Previous and Next Names for a Specific Employee

```
SELECT * FROM ( SELECT Lname, Salary, PrevLname = LAG(Lname) OVER (ORDER BY S
alary), NextLname = LEAD(Lname) OVER (ORDER BY Salary) FROM Employee ) AS New
Table WHERE Lname = 'Ahmed';
```

4. Using PARTITION BY with LAG and LEAD

Example: Get Previous and Next Names by Department

```
SELECT Lname, Salary, Dno, PrevLname = LAG(Lname) OVER (PARTITION BY Dno ORDE
R BY Salary), NextLname = LEAD(Lname) OVER (PARTITION BY Dno ORDER BY Salary)
FROM Employee;
```

5. Using FIRST_VALUE and LAST_VALUE

- `FIRST_VALUE` retrieves the first value in an ordered set of values.
- `LAST_VALUE` retrieves the last value in an ordered set of values.

Example: Get First and Last Employee Names Based on Salary

```
SELECT Lname, Salary, Dno, FirstName = FIRST_VALUE(Lname) OVER (ORDER BY Sala
ry), LastName = LAST_VALUE(Lname) OVER (ORDER BY Salary ROWS BETWEEN UNBOUNDE
D PRECEDING AND UNBOUNDED FOLLOWING) FROM Employee;
```

6. Combined Example with LAG, LEAD, FIRST_VALUE, and LAST_VALUE

Example: Combine All Functions for Employee Analysis

```
SELECT Lname, Salary, Dno, PrevLname = LAG(Lname) OVER (ORDER BY Salary), Nex
tLname = LEAD(Lname) OVER (ORDER BY Salary), FirstName = FIRST_VALUE(Lname) O
VER (ORDER BY Salary), LastName = LAST_VALUE(Lname) OVER (ORDER BY Salary ROW
S BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) FROM Employee;
```

2- User defined functions

1. Scalar Function

A scalar function returns a single value. It can be used to encapsulate logic that needs to be reused.

Example: Get Employee First Name by SSN

```
CREATE FUNCTION dbo.getsname(@id INT) RETURNS VARCHAR(20) AS BEGIN DECLARE @n
ame VARCHAR(20); SELECT @name = EmployeeName FROM Employee WHERE SSN = @id; R
ETURN @name; END; GO -- Usage SELECT dbo.getsname(112233) AS EmployeeFirstNam
e;
```

2. Inline Table-Valued Function

An inline table-valued function returns a table and consists of a single SELECT statement. It cannot contain any control-of-flow language like `IF` or `TRY...CATCH`.

Example: Get Employees Working in a Specific Department

```
CREATE FUNCTION dbo.people_work_in_dep(@Dep_id INT) RETURNS TABLE AS RETURN ( SELECT EmployeeName, Salary * 12 AS TotalSalary FROM Employees WHERE DepartmentID = @Dep_id ); GO -- Usage SELECT * FROM dbo.people_work_in_dep(10); SELECT SUM(TotalSalary) FROM dbo.people_work_in_dep(10); SELECT Fname FROM dbo.people_work_in_dep(10);
```

3. Multi-Statement Table-Valued Function

A multi-statement table-valued function can contain multiple statements, including `IF` conditions and loops. It returns a table variable that you define within the function.

Example: Get Employees Based on Format

```
CREATE FUNCTION dbo.getemployees(@format VARCHAR(20)) RETURNS @t TABLE (id INT, ename VARCHAR(20)) AS BEGIN IF @format = 'first' INSERT INTO @t SELECT SSN, Fname FROM Employees; ELSE IF @format = 'last' INSERT INTO @t SELECT SSN, Lname FROM Employees; ELSE IF @format = 'full' INSERT INTO @t SELECT SSN, Fname + ' ' + Lname FROM Employees; RETURN; END; GO -- Usage SELECT * FROM dbo.getemployees('full');
```

Notes on User-Defined Functions

- The type of function you create depends on the return type you need (scalar or table).
- Only `SELECT` statements can be used inside these functions.
- You can find user-defined functions under `Company_DB` -> `Programmability` -> `Functions`.

⇒ To get max name length

```
DECLARE @max_name_length INT; SELECT @max_name_length =
MAX(LEN(EmployeeName)) FROM Employees; SELECT EmployeeName FROM Employees
WHERE LEN(EmployeeName) = @max_name_length; SELECT TOP 1 EmployeeName FROM
Employees ORDER BY LEN(EmployeeName) DESC;
```