# 📝 Complete MS SQL Stored Procedures Tutorial

**Goal:** By the end of this tutorial, you will be able to confidently create, modify, execute, debug, and optimize stored procedures in MS SQL for real-world data engineering scenarios.

---

## Table of Contents

---

## 1. Introduction to Stored Procedures

**Definition:**
A stored procedure (SP) is a **precompiled set of SQL statements stored in the database**, which can be executed as a single call.

**Benefits:**

- Reusability
- Reduced network traffic
- Security (can restrict direct table access)
- Maintainability
- Performance improvements (precompiled execution plan)

**Syntax:**

```
CREATE PROCEDURE procedure_name
AS
BEGIN
    -- SQL statements here
END;
```

---

## 2. Basic Stored Procedures

### 2.1 Creating a Simple Procedure

```
CREATE PROCEDURE GetAllProducts
AS
BEGIN
    SELECT * FROM Production.Product;
END;
```

### 2.2 Executing a Stored Procedure

```
EXEC GetAllProducts;
-- or
EXECUTE GetAllProducts;
```

### 2.3 Dropping a Stored Procedure

```
DROP PROCEDURE IF EXISTS GetAllProducts;
```

---

## 3. Input Parameters & Output Parameters

### 3.1 Input Parameters

```sql
CREATE PROCEDURE GetProductByCategory
    @CategoryID INT
AS
BEGIN
    SELECT ProductID, Name, ListPrice
    FROM Production.Product
    WHERE ProductCategoryID = @CategoryID;
END;
```

Execute with parameter:

```sql
EXEC GetProductByCategory @CategoryID = 3;
```

### 3.2 Output Parameters

```sql
CREATE PROCEDURE GetProductCountByCategory
    @CategoryID INT,
    @ProductCount INT OUTPUT
AS
BEGIN
    SELECT @ProductCount = COUNT(*)
    FROM Production.Product
    WHERE ProductCategoryID = @CategoryID;
END;
```

Execute with OUTPUT:

```sql
DECLARE @Count INT;
EXEC GetProductCountByCategory @CategoryID = 3, @ProductCount = @Count OUTPUT;
SELECT @Count AS ProductCount;
```

---

## 4. Control-of-Flow & Error Handling

### 4.1 IF...ELSE

```
CREATE PROCEDURE CheckProductStock
    @ProductID INT
AS
BEGIN
    DECLARE @Stock INT;
    SELECT @Stock = SafetyStockLevel FROM Production.Product WHERE ProductID =
@ProductID;

    IF @Stock > 50
        PRINT 'Stock is sufficient';
    ELSE
        PRINT 'Stock is low';
END;
```

## 4.2 WHILE Loop

```
CREATE PROCEDURE PrintNumbers
AS
BEGIN
    DECLARE @i INT = 1;
    WHILE @i <= 5
    BEGIN
        PRINT @i;
        SET @i = @i + 1;
    END
END;
```

## 4.3 TRY...CATCH for Error Handling

```
CREATE PROCEDURE SafeDivision
    @a INT,
    @b INT
AS
BEGIN
    BEGIN TRY
        SELECT @a / @b AS Result;
    END TRY
    BEGIN CATCH
        PRINT 'Error: ' + ERROR_MESSAGE();
    END CATCH
END;
```

## 5. Advanced Concepts

### 5.1 Default Parameter Values

```
CREATE PROCEDURE GetProductsByCategory
    @CategoryID INT = NULL
AS
BEGIN
    IF @CategoryID IS NULL
        SELECT * FROM Production.Product;
    ELSE
        SELECT * FROM Production.Product WHERE ProductCategoryID = @CategoryID;
END;
```

### 5.2 Returning Values

```
CREATE PROCEDURE MultiplyNumbers
    @a INT,
    @b INT
AS
BEGIN
    RETURN @a * @b;
END;

DECLARE @Result INT;
EXEC @Result = MultiplyNumbers @a = 5, @b = 4;
SELECT @Result AS Product;
```

### 5.3 Nested Stored Procedures

```
CREATE PROCEDURE OuterProcedure
AS
BEGIN
    PRINT 'Outer SP Start';
    EXEC InnerProcedure;
    PRINT 'Outer SP End';
END;

CREATE PROCEDURE InnerProcedure
AS
BEGIN
    PRINT 'Inner SP Executed';
END;
```

## 6. Dynamic SQL in Stored Procedures

**Use Case:** Table or column names are not known until runtime.

```
CREATE PROCEDURE GetTableData
    @TableName NVARCHAR(128)
AS
BEGIN
    DECLARE @SQL NVARCHAR(MAX);
    SET @SQL = 'SELECT TOP 5 * FROM ' + QUOTENAME(@TableName);
    EXEC sp_executesql @SQL;
END;
```

**Execute:**

```
EXEC GetTableData @TableName = 'Production.Product';
```

⚠ **Security Note:** Always use `QUOTENAME` to prevent SQL injection.

## 7. Stored Procedure Best Practices

1. Always use **schema-qualified names** (e.g., `dbo.ProcedureName`).
2. Use **parameters** instead of concatenating strings for filtering.
3. Keep SPs **small and modular**.
4. Use **SET NOCOUNT ON** to prevent extra messages for better performance.

   ```
   CREATE PROCEDURE SampleProcedure
   AS
   BEGIN
    SET NOCOUNT ON;
    -- SQL statements here
   END;
   ```

5. Include **error handling** using TRY...CATCH.
6. Comment and document SP logic for maintainability.

## 8. Debugging, Performance & Optimization

- **Execution Plan:** Use `SET SHOWPLAN_TEXT ON;` to check the SP execution plan.
- **Profiler & Extended Events:** Monitor SP performance in real time.
- **Index usage:** Ensure indexed columns are used in WHERE and JOIN.
- **Avoid CURSORs:** Use set-based operations where possible.
- **Use WITH RECOMPILE cautiously:** Forces SP recompilation to avoid parameter sniffing issues.

## 9. Real-World Scenarios

### 9.1 Reporting Stored Procedure

```sql
CREATE PROCEDURE GetMonthlySales
    @Year INT,
    @Month INT
AS
BEGIN
    SELECT CustomerID, SUM(TotalDue) AS MonthlySales
    FROM Sales.SalesOrderHeader
    WHERE YEAR(OrderDate) = @Year AND MONTH(OrderDate) = @Month
    GROUP BY CustomerID
    ORDER BY MonthlySales DESC;
END;
```

### 9.2 Data Pipeline Stored Procedure

```sql
CREATE PROCEDURE LoadStagingTable
AS
BEGIN
    BEGIN TRY
        BEGIN TRANSACTION;

        TRUNCATE TABLE Staging.Product;

        INSERT INTO Staging.Product(ProductID, Name, ListPrice)
        SELECT ProductID, Name, ListPrice
        FROM Production.Product;

        COMMIT TRANSACTION;
    END TRY
    BEGIN CATCH
        ROLLBACK TRANSACTION;
        PRINT 'Error: ' + ERROR_MESSAGE();
    END CATCH
END;
```

### 9.3 Automation via SQL Agent

- Schedule stored procedures to run automatically for ETL or reporting.

---

## ☑ Summary

- Stored procedures encapsulate logic in the database for **reusability, security, and performance**.
- Master **parameters, control-of-flow, error handling, and dynamic SQL**.
- Optimize SPs using **indexes, execution plans, and set-based operations**.
- Real-world applications: **reporting, ETL pipelines, automation, and data validation**.