# Complete Guide to Stored Procedures in SSMS

## Table of Contents

1. [What is a Stored Procedure?](#what-is-a-stored-procedure)
2. [Benefits of Stored Procedures](#benefits-of-stored-procedures)
3. [Basic Stored Procedure Syntax](#basic-stored-procedure-syntax)
4. [Simple Stored Procedures](#simple-stored-procedures)
5. [Stored Procedures with Parameters](#stored-procedures-with-parameters)
6. [Input and Output Parameters](#input-and-output-parameters)
7. [Return Values and Status](#return-values-and-status)
8. [Control Flow in Stored Procedures](#control-flow-in-stored-procedures)
9. [Error Handling](#error-handling)
10. [Dynamic SQL in Stored Procedures](#dynamic-sql-in-stored-procedures)
11. [Cursors in Stored Procedures](#cursors-in-stored-procedures)
12. [Advanced Stored Procedures](#advanced-stored-procedures)
13. [Performance Optimization](#performance-optimization)
14. [Security and Permissions](#security-and-permissions)
15. [Managing Stored Procedures in SSMS](#managing-stored-procedures-in-ssms)
16. [Best Practices](#best-practices)

## What is a Stored Procedure?

A **Stored Procedure** is a precompiled collection of SQL statements and optional control-flow statements stored in the database. Stored procedures can accept parameters, return values, and execute complex business logic.

### Key Characteristics:

* **Precompiled**: Execution plan is cached for better performance
* **Parameterized**: Can accept input and output parameters
* **Reusable**: Can be called multiple times from different applications
* **Secure**: Provides controlled access to database objects
* **Centralized**: Business logic is centralized in the database

## Benefits of Stored Procedures

### Performance Benefits

* **Execution Plan Caching**: Plans are compiled and cached
* **Reduced Network Traffic**: Only procedure name and parameters sent
* **Batch Processing**: Multiple statements executed together

### Security Benefits

* **SQL Injection Prevention**: Parameters are strongly typed
* **Controlled Access**: Users can execute procedures without direct table access
* **Permission Management**: Grant execute permissions without table permissions

### Maintenance Benefits

* **Centralized Logic**: Business rules in one location
* **Code Reusability**: Same logic used across multiple applications
* **Easier Updates**: Change logic without updating applications

## Basic Stored Procedure Syntax

-- Create Stored Procedure  
CREATE PROCEDURE procedure\_name  
 @parameter1 datatype = default\_value,  
 @parameter2 datatype = default\_value  
AS  
BEGIN  
 -- SQL statements here  
 SELECT, INSERT, UPDATE, DELETE statements  
 -- Control flow statements  
 IF, WHILE, TRY-CATCH, etc.  
END  
  
-- Execute Stored Procedure  
EXEC procedure\_name @parameter1 = value1, @parameter2 = value2  
-- or  
EXECUTE procedure\_name value1, value2  
  
-- Modify Stored Procedure  
ALTER PROCEDURE procedure\_name  
AS  
BEGIN  
 -- Modified SQL statements  
END  
  
-- Drop Stored Procedure  
DROP PROCEDURE procedure\_name

## Simple Stored Procedures

Let’s start with sample tables:

-- Create sample tables  
CREATE TABLE employees (  
 employee\_id INT IDENTITY(1,1) PRIMARY KEY,  
 first\_name VARCHAR(50),  
 last\_name VARCHAR(50),  
 email VARCHAR(100),  
 phone VARCHAR(20),  
 department VARCHAR(50),  
 position VARCHAR(50),  
 salary DECIMAL(10,2),  
 hire\_date DATE,  
 manager\_id INT,  
 is\_active BIT DEFAULT 1  
);  
  
CREATE TABLE departments (  
 dept\_id INT IDENTITY(1,1) PRIMARY KEY,  
 dept\_name VARCHAR(50),  
 location VARCHAR(50),  
 budget DECIMAL(12,2)  
);  
  
CREATE TABLE audit\_log (  
 log\_id INT IDENTITY(1,1) PRIMARY KEY,  
 table\_name VARCHAR(50),  
 operation VARCHAR(10),  
 user\_name VARCHAR(50),  
 timestamp DATETIME,  
 details VARCHAR(MAX)  
);  
  
-- Insert sample data  
INSERT INTO employees VALUES  
('John', 'Doe', 'john.doe@company.com', '555-1001', 'IT', 'Developer', 75000, '2020-01-15', NULL, 1),  
('Jane', 'Smith', 'jane.smith@company.com', '555-1002', 'IT', 'Senior Developer', 85000, '2019-03-10', 1, 1),  
('Bob', 'Johnson', 'bob.johnson@company.com', '555-1003', 'Sales', 'Sales Rep', 55000, '2021-06-20', NULL, 1),  
('Alice', 'Brown', 'alice.brown@company.com', '555-1004', 'HR', 'HR Manager', 70000, '2018-09-05', NULL, 1);  
  
INSERT INTO departments VALUES  
('Information Technology', 'New York', 500000),  
('Sales', 'Chicago', 300000),  
('Human Resources', 'Los Angeles', 200000),  
('Finance', 'Boston', 250000);

### Basic Select Procedure

-- Simple procedure to get all active employees  
CREATE PROCEDURE sp\_GetAllEmployees  
AS  
BEGIN  
 SELECT   
 employee\_id,  
 first\_name,  
 last\_name,  
 first\_name + ' ' + last\_name AS full\_name,  
 email,  
 department,  
 position,  
 salary,  
 hire\_date  
 FROM employees  
 WHERE is\_active = 1  
 ORDER BY last\_name, first\_name;  
END  
  
-- Execute the procedure  
EXEC sp\_GetAllEmployees;

### Basic Insert Procedure

-- Procedure to add new employee  
CREATE PROCEDURE sp\_AddEmployee  
AS  
BEGIN  
 INSERT INTO employees (first\_name, last\_name, email, department, position, salary, hire\_date)  
 VALUES ('New', 'Employee', 'new.employee@company.com', 'IT', 'Trainee', 45000, GETDATE());  
   
 -- Return the new employee ID  
 SELECT SCOPE\_IDENTITY() AS new\_employee\_id;  
END  
  
-- Execute  
EXEC sp\_AddEmployee;

## Stored Procedures with Parameters

### Input Parameters

-- Procedure with input parameters  
CREATE PROCEDURE sp\_GetEmployeesByDepartment  
 @department VARCHAR(50),  
 @min\_salary DECIMAL(10,2) = 0 -- Default value  
AS  
BEGIN  
 SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 email,  
 position,  
 salary,  
 hire\_date  
 FROM employees  
 WHERE department = @department   
 AND salary >= @min\_salary  
 AND is\_active = 1  
 ORDER BY salary DESC;  
END  
  
-- Execute with parameters  
EXEC sp\_GetEmployeesByDepartment @department = 'IT', @min\_salary = 60000;  
EXEC sp\_GetEmployeesByDepartment 'Sales'; -- Using default min\_salary

### Multiple Input Parameters

-- Procedure with multiple parameters  
CREATE PROCEDURE sp\_GetEmployeesFiltered  
 @department VARCHAR(50) = NULL,  
 @min\_salary DECIMAL(10,2) = 0,  
 @max\_salary DECIMAL(10,2) = 999999,  
 @hire\_date\_from DATE = '1900-01-01',  
 @hire\_date\_to DATE = NULL  
AS  
BEGIN  
 -- Set default for @hire\_date\_to if NULL  
 IF @hire\_date\_to IS NULL  
 SET @hire\_date\_to = GETDATE();  
   
 SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 department,  
 position,  
 salary,  
 hire\_date,  
 DATEDIFF(YEAR, hire\_date, GETDATE()) AS years\_of\_service  
 FROM employees  
 WHERE   
 (@department IS NULL OR department = @department)  
 AND salary BETWEEN @min\_salary AND @max\_salary  
 AND hire\_date BETWEEN @hire\_date\_from AND @hire\_date\_to  
 AND is\_active = 1  
 ORDER BY hire\_date DESC;  
END  
  
-- Various execution examples  
EXEC sp\_GetEmployeesFiltered @department = 'IT';  
EXEC sp\_GetEmployeesFiltered @min\_salary = 70000, @max\_salary = 90000;  
EXEC sp\_GetEmployeesFiltered @hire\_date\_from = '2020-01-01';

## Input and Output Parameters

### Output Parameters

-- Procedure with output parameters  
CREATE PROCEDURE sp\_GetDepartmentStats  
 @department VARCHAR(50),  
 @employee\_count INT OUTPUT,  
 @avg\_salary DECIMAL(10,2) OUTPUT,  
 @total\_salary DECIMAL(12,2) OUTPUT  
AS  
BEGIN  
 SELECT   
 @employee\_count = COUNT(\*),  
 @avg\_salary = AVG(salary),  
 @total\_salary = SUM(salary)  
 FROM employees  
 WHERE department = @department AND is\_active = 1;  
   
 -- Also return a result set  
 SELECT   
 first\_name + ' ' + last\_name AS employee\_name,  
 position,  
 salary,  
 salary - @avg\_salary AS salary\_diff\_from\_avg  
 FROM employees  
 WHERE department = @department AND is\_active = 1  
 ORDER BY salary DESC;  
END  
  
-- Execute with output parameters  
DECLARE @count INT, @avg DECIMAL(10,2), @total DECIMAL(12,2);  
EXEC sp\_GetDepartmentStats   
 @department = 'IT',  
 @employee\_count = @count OUTPUT,  
 @avg\_salary = @avg OUTPUT,  
 @total\_salary = @total OUTPUT;  
  
SELECT   
 @count AS EmployeeCount,  
 @avg AS AverageSalary,  
 @total AS TotalSalary;

### Input/Output Parameters

-- Procedure with input/output parameter  
CREATE PROCEDURE sp\_CalculateBonus  
 @employee\_id INT,  
 @bonus\_percentage DECIMAL(5,2) = 10.0,  
 @bonus\_amount DECIMAL(10,2) OUTPUT  
AS  
BEGIN  
 DECLARE @current\_salary DECIMAL(10,2);  
   
 -- Get current salary  
 SELECT @current\_salary = salary  
 FROM employees  
 WHERE employee\_id = @employee\_id AND is\_active = 1;  
   
 -- Calculate bonus  
 IF @current\_salary IS NOT NULL  
 BEGIN  
 SET @bonus\_amount = @current\_salary \* (@bonus\_percentage / 100.0);  
   
 -- Return employee info with bonus  
 SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 salary AS current\_salary,  
 @bonus\_percentage AS bonus\_percentage,  
 @bonus\_amount AS bonus\_amount,  
 salary + @bonus\_amount AS total\_compensation  
 FROM employees  
 WHERE employee\_id = @employee\_id;  
 END  
 ELSE  
 BEGIN  
 SET @bonus\_amount = 0;  
 SELECT 'Employee not found or inactive' AS message;  
 END  
END  
  
-- Execute  
DECLARE @bonus DECIMAL(10,2);  
EXEC sp\_CalculateBonus   
 @employee\_id = 1,  
 @bonus\_percentage = 15.0,  
 @bonus\_amount = @bonus OUTPUT;  
  
SELECT @bonus AS CalculatedBonus;

## Return Values and Status

### Using Return Values

-- Procedure with return values for status codes  
CREATE PROCEDURE sp\_UpdateEmployeeSalary  
 @employee\_id INT,  
 @new\_salary DECIMAL(10,2)  
AS  
BEGIN  
 DECLARE @current\_salary DECIMAL(10,2);  
   
 -- Check if employee exists  
 SELECT @current\_salary = salary  
 FROM employees  
 WHERE employee\_id = @employee\_id AND is\_active = 1;  
   
 IF @current\_salary IS NULL  
 RETURN -1; -- Employee not found  
   
 IF @new\_salary <= 0  
 RETURN -2; -- Invalid salary amount  
   
 IF @new\_salary < @current\_salary \* 0.8  
 RETURN -3; -- Salary decrease too large (more than 20%)  
   
 -- Update salary  
 UPDATE employees  
 SET salary = @new\_salary  
 WHERE employee\_id = @employee\_id;  
   
 -- Log the change  
 INSERT INTO audit\_log (table\_name, operation, user\_name, timestamp, details)  
 VALUES ('employees', 'UPDATE', USER\_NAME(), GETDATE(),   
 'Salary updated for employee ' + CAST(@employee\_id AS VARCHAR) +   
 ' from ' + CAST(@current\_salary AS VARCHAR) +   
 ' to ' + CAST(@new\_salary AS VARCHAR));  
   
 RETURN 0; -- Success  
END  
  
-- Execute and check return value  
DECLARE @return\_value INT;  
EXEC @return\_value = sp\_UpdateEmployeeSalary @employee\_id = 1, @new\_salary = 80000;  
  
SELECT   
 CASE @return\_value  
 WHEN 0 THEN 'Success: Salary updated'  
 WHEN -1 THEN 'Error: Employee not found'  
 WHEN -2 THEN 'Error: Invalid salary amount'  
 WHEN -3 THEN 'Error: Salary decrease too large'  
 ELSE 'Unknown error'  
 END AS Result;

## Control Flow in Stored Procedures

### IF-ELSE Statements

CREATE PROCEDURE sp\_PromoteEmployee  
 @employee\_id INT,  
 @new\_position VARCHAR(50),  
 @salary\_increase\_percent DECIMAL(5,2) = 0  
AS  
BEGIN  
 DECLARE @current\_salary DECIMAL(10,2), @current\_position VARCHAR(50);  
 DECLARE @new\_salary DECIMAL(10,2);  
   
 -- Get current employee info  
 SELECT @current\_salary = salary, @current\_position = position  
 FROM employees  
 WHERE employee\_id = @employee\_id AND is\_active = 1;  
   
 IF @current\_salary IS NULL  
 BEGIN  
 SELECT 'Employee not found' AS message;  
 RETURN;  
 END  
   
 -- Calculate new salary based on position  
 IF @new\_position LIKE '%Manager%' OR @new\_position LIKE '%Director%'  
 BEGIN  
 -- Management positions get at least 15% increase  
 IF @salary\_increase\_percent < 15  
 SET @salary\_increase\_percent = 15;  
 END  
 ELSE IF @new\_position LIKE '%Senior%'  
 BEGIN  
 -- Senior positions get at least 10% increase  
 IF @salary\_increase\_percent < 10  
 SET @salary\_increase\_percent = 10;  
 END  
 ELSE IF @salary\_increase\_percent = 0  
 BEGIN  
 -- Default 5% increase for other promotions  
 SET @salary\_increase\_percent = 5;  
 END  
   
 SET @new\_salary = @current\_salary \* (1 + @salary\_increase\_percent / 100.0);  
   
 -- Update employee  
 UPDATE employees  
 SET position = @new\_position, salary = @new\_salary  
 WHERE employee\_id = @employee\_id;  
   
 -- Return promotion details  
 SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 @current\_position AS old\_position,  
 @new\_position AS new\_position,  
 @current\_salary AS old\_salary,  
 @new\_salary AS new\_salary,  
 @salary\_increase\_percent AS increase\_percentage,  
 'Promotion successful' AS status;  
END  
  
-- Execute  
EXEC sp\_PromoteEmployee @employee\_id = 1, @new\_position = 'Senior Developer';

### WHILE Loops

CREATE PROCEDURE sp\_ProcessSalaryReview  
 @review\_year INT = NULL  
AS  
BEGIN  
 IF @review\_year IS NULL  
 SET @review\_year = YEAR(GETDATE());  
   
 DECLARE @employee\_id INT, @current\_salary DECIMAL(10,2), @years\_service INT;  
 DECLARE @new\_salary DECIMAL(10,2), @increase\_percent DECIMAL(5,2);  
 DECLARE @processed\_count INT = 0;  
   
 -- Cursor to process each employee  
 DECLARE salary\_cursor CURSOR FOR  
 SELECT employee\_id, salary, DATEDIFF(YEAR, hire\_date, GETDATE()) AS years\_service  
 FROM employees  
 WHERE is\_active = 1;  
   
 OPEN salary\_cursor;  
 FETCH NEXT FROM salary\_cursor INTO @employee\_id, @current\_salary, @years\_service;  
   
 -- Process each employee  
 WHILE @@FETCH\_STATUS = 0  
 BEGIN  
 -- Determine increase based on years of service  
 IF @years\_service >= 10  
 SET @increase\_percent = 8.0; -- 8% for 10+ years  
 ELSE IF @years\_service >= 5  
 SET @increase\_percent = 6.0; -- 6% for 5+ years  
 ELSE IF @years\_service >= 2  
 SET @increase\_percent = 4.0; -- 4% for 2+ years  
 ELSE  
 SET @increase\_percent = 2.0; -- 2% for newer employees  
   
 SET @new\_salary = @current\_salary \* (1 + @increase\_percent / 100.0);  
   
 -- Update salary  
 UPDATE employees  
 SET salary = @new\_salary  
 WHERE employee\_id = @employee\_id;  
   
 -- Log the review  
 INSERT INTO audit\_log (table\_name, operation, user\_name, timestamp, details)  
 VALUES ('employees', 'REVIEW', USER\_NAME(), GETDATE(),  
 'Annual salary review ' + CAST(@review\_year AS VARCHAR) +   
 ' - Employee ' + CAST(@employee\_id AS VARCHAR) +   
 ' increased by ' + CAST(@increase\_percent AS VARCHAR) + '%');  
   
 SET @processed\_count = @processed\_count + 1;  
   
 FETCH NEXT FROM salary\_cursor INTO @employee\_id, @current\_salary, @years\_service;  
 END  
   
 CLOSE salary\_cursor;  
 DEALLOCATE salary\_cursor;  
   
 SELECT   
 @processed\_count AS employees\_processed,  
 @review\_year AS review\_year,  
 'Salary review completed successfully' AS status;  
END  
  
-- Execute  
EXEC sp\_ProcessSalaryReview @review\_year = 2024;

## Error Handling

### TRY-CATCH Blocks

CREATE PROCEDURE sp\_TransferEmployee  
 @employee\_id INT,  
 @new\_department VARCHAR(50),  
 @effective\_date DATE = NULL  
AS  
BEGIN  
 SET NOCOUNT ON;  
   
 IF @effective\_date IS NULL  
 SET @effective\_date = GETDATE();  
   
 DECLARE @old\_department VARCHAR(50), @employee\_name VARCHAR(100);  
 DECLARE @error\_message VARCHAR(MAX);  
   
 BEGIN TRY  
 -- Start transaction  
 BEGIN TRANSACTION;  
   
 -- Get current employee info  
 SELECT   
 @old\_department = department,  
 @employee\_name = first\_name + ' ' + last\_name  
 FROM employees  
 WHERE employee\_id = @employee\_id AND is\_active = 1;  
   
 IF @employee\_name IS NULL  
 BEGIN  
 RAISERROR('Employee not found or inactive', 16, 1);  
 END  
   
 -- Check if department exists  
 IF NOT EXISTS (SELECT 1 FROM departments WHERE dept\_name = @new\_department)  
 BEGIN  
 RAISERROR('Department does not exist', 16, 1);  
 END  
   
 -- Check if it's actually a transfer  
 IF @old\_department = @new\_department  
 BEGIN  
 RAISERROR('Employee is already in the specified department', 16, 1);  
 END  
   
 -- Update employee department  
 UPDATE employees  
 SET department = @new\_department  
 WHERE employee\_id = @employee\_id;  
   
 -- Log the transfer  
 INSERT INTO audit\_log (table\_name, operation, user\_name, timestamp, details)  
 VALUES ('employees', 'TRANSFER', USER\_NAME(), GETDATE(),  
 'Employee ' + @employee\_name + ' transferred from ' +   
 @old\_department + ' to ' + @new\_department +   
 ' effective ' + CAST(@effective\_date AS VARCHAR));  
   
 -- Commit transaction  
 COMMIT TRANSACTION;  
   
 -- Return success message  
 SELECT   
 @employee\_id AS employee\_id,  
 @employee\_name AS employee\_name,  
 @old\_department AS from\_department,  
 @new\_department AS to\_department,  
 @effective\_date AS effective\_date,  
 'Transfer completed successfully' AS status;  
   
 END TRY  
 BEGIN CATCH  
 -- Rollback transaction on error  
 IF @@TRANCOUNT > 0  
 ROLLBACK TRANSACTION;  
   
 -- Get error information  
 SET @error\_message = 'Error: ' + ERROR\_MESSAGE() +   
 ' (Error Number: ' + CAST(ERROR\_NUMBER() AS VARCHAR) +   
 ', Line: ' + CAST(ERROR\_LINE() AS VARCHAR) + ')';  
   
 -- Log error  
 INSERT INTO audit\_log (table\_name, operation, user\_name, timestamp, details)  
 VALUES ('employees', 'ERROR', USER\_NAME(), GETDATE(), @error\_message);  
   
 -- Return error message  
 SELECT @error\_message AS error\_message;  
   
 -- Re-raise the error  
 THROW;  
 END CATCH  
END  
  
-- Execute  
EXEC sp\_TransferEmployee @employee\_id = 1, @new\_department = 'Sales';

## Dynamic SQL in Stored Procedures

### Basic Dynamic SQL

CREATE PROCEDURE sp\_GetEmployeesDynamic  
 @columns VARCHAR(MAX) = '\*',  
 @where\_clause VARCHAR(MAX) = NULL,  
 @order\_by VARCHAR(MAX) = 'last\_name'  
AS  
BEGIN  
 DECLARE @sql NVARCHAR(MAX);  
   
 SET @sql = 'SELECT ' + @columns + ' FROM employees WHERE is\_active = 1';  
   
 IF @where\_clause IS NOT NULL  
 SET @sql = @sql + ' AND (' + @where\_clause + ')';  
   
 SET @sql = @sql + ' ORDER BY ' + @order\_by;  
   
 -- Print the SQL for debugging  
 PRINT @sql;  
   
 -- Execute the dynamic SQL  
 EXEC sp\_executesql @sql;  
END  
  
-- Execute with different parameters  
EXEC sp\_GetEmployeesDynamic   
 @columns = 'first\_name, last\_name, department, salary',  
 @where\_clause = 'salary > 60000',  
 @order\_by = 'salary DESC';

### Advanced Dynamic SQL with Parameters

CREATE PROCEDURE sp\_SearchEmployees  
 @search\_term VARCHAR(100) = NULL,  
 @department VARCHAR(50) = NULL,  
 @min\_salary DECIMAL(10,2) = NULL,  
 @max\_salary DECIMAL(10,2) = NULL,  
 @sort\_column VARCHAR(50) = 'last\_name',  
 @sort\_direction VARCHAR(4) = 'ASC',  
 @page\_number INT = 1,  
 @page\_size INT = 20  
AS  
BEGIN  
 DECLARE @sql NVARCHAR(MAX);  
 DECLARE @where\_conditions NVARCHAR(MAX) = '';  
 DECLARE @params NVARCHAR(MAX);  
 DECLARE @offset INT;  
   
 SET @offset = (@page\_number - 1) \* @page\_size;  
   
 -- Build WHERE conditions dynamically  
 SET @where\_conditions = 'WHERE is\_active = 1';  
   
 IF @search\_term IS NOT NULL  
 SET @where\_conditions = @where\_conditions +   
 ' AND (first\_name LIKE ''%'' + @search\_term + ''%'' OR last\_name LIKE ''%'' + @search\_term + ''%'' OR email LIKE ''%'' + @search\_term + ''%'')';  
   
 IF @department IS NOT NULL  
 SET @where\_conditions = @where\_conditions + ' AND department = @department';  
   
 IF @min\_salary IS NOT NULL  
 SET @where\_conditions = @where\_conditions + ' AND salary >= @min\_salary';  
   
 IF @max\_salary IS NOT NULL  
 SET @where\_conditions = @where\_conditions + ' AND salary <= @max\_salary';  
   
 -- Build the complete SQL  
 SET @sql = '  
 SELECT   
 employee\_id,  
 first\_name + '' '' + last\_name AS full\_name,  
 email,  
 department,  
 position,  
 salary,  
 hire\_date  
 FROM employees ' + @where\_conditions + '  
 ORDER BY ' + QUOTENAME(@sort\_column) + ' ' + @sort\_direction + '  
 OFFSET @offset ROWS  
 FETCH NEXT @page\_size ROWS ONLY;  
   
 -- Also return total count  
 SELECT COUNT(\*) as total\_records  
 FROM employees ' + @where\_conditions + ';';  
   
 -- Define parameters  
 SET @params = '@search\_term VARCHAR(100), @department VARCHAR(50), @min\_salary DECIMAL(10,2), @max\_salary DECIMAL(10,2), @offset INT, @page\_size INT';  
   
 -- Execute with parameters  
 EXEC sp\_executesql @sql, @params,   
 @search\_term = @search\_term,  
 @department = @department,  
 @min\_salary = @min\_salary,  
 @max\_salary = @max\_salary,  
 @offset = @offset,  
 @page\_size = @page\_size;  
END  
  
-- Execute  
EXEC sp\_SearchEmployees   
 @search\_term = 'John',  
 @department = 'IT',  
 @min\_salary = 50000,  
 @sort\_column = 'salary',  
 @sort\_direction = 'DESC',  
 @page\_number = 1,  
 @page\_size = 10;

## Advanced Stored Procedures

### Procedure with Multiple Result Sets

CREATE PROCEDURE sp\_ComprehensiveEmployeeReport  
 @department VARCHAR(50) = NULL  
AS  
BEGIN  
 SET NOCOUNT ON;  
   
 -- Result Set 1: Employee List  
 SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 department,  
 position,  
 salary,  
 hire\_date,  
 DATEDIFF(YEAR, hire\_date, GETDATE()) AS years\_of\_service  
 FROM employees  
 WHERE (@department IS NULL OR department = @department) AND is\_active = 1  
 ORDER BY department, last\_name;  
   
 -- Result Set 2: Department Summary  
 SELECT   
 department,  
 COUNT(\*) AS employee\_count,  
 AVG(salary) AS avg\_salary,  
 MIN(salary) AS min\_salary,  
 MAX(salary) AS max\_salary,  
 SUM(salary) AS total\_salary  
 FROM employees  
 WHERE (@department IS NULL OR department = @department) AND is\_active = 1  
 GROUP BY department;  
   
 -- Result Set 3: Salary Distribution  
 SELECT   
 CASE   
 WHEN salary < 50000 THEN 'Under $50K'  
 WHEN salary < 70000 THEN '$50K - $70K'  
 WHEN salary < 90000 THEN '$70K - $90K'  
 ELSE 'Over $90K'  
 END AS salary\_range,  
 COUNT(\*) AS employee\_count,  
 AVG(salary) AS avg\_salary\_in\_range  
 FROM employees  
 WHERE (@department IS NULL OR department = @department) AND is\_active = 1  
 GROUP BY   
 CASE   
 WHEN salary < 50000 THEN 'Under $50K'  
 WHEN salary < 70000 THEN '$50K - $70K'  
 WHEN salary < 90000 THEN '$70K - $90K'  
 ELSE 'Over $90K'  
 END  
 ORDER BY avg\_salary\_in\_range;  
END  
  
-- Execute  
EXEC sp\_ComprehensiveEmployeeReport @department = 'IT';

### Recursive Stored Procedure

CREATE PROCEDURE sp\_GetEmployeeHierarchy  
 @manager\_id INT = NULL,  
 @level INT = 1  
AS  
BEGIN  
 -- Get employees reporting to the specified manager  
 SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 position,  
 department,  
 salary,  
 manager\_id,  
 @level AS hierarchy\_level,  
 REPLICATE(' ', @level - 1) + first\_name + ' ' + last\_name AS indented\_name  
 FROM employees  
 WHERE manager\_id = @manager\_id AND is\_active = 1;  
   
 -- Get all direct reports of the current level  
 DECLARE @emp\_id INT;  
 DECLARE hierarchy\_cursor CURSOR FOR  
 SELECT employee\_id  
 FROM employees  
 WHERE manager\_id = @manager\_id AND is\_active = 1;  
   
 OPEN hierarchy\_cursor;  
 FETCH NEXT FROM hierarchy\_cursor INTO @emp\_id;  
   
 WHILE @@FETCH\_STATUS = 0  
 BEGIN  
 -- Recursive call for each employee  
 EXEC sp\_GetEmployeeHierarchy @manager\_id = @emp\_id, @level = @level + 1;  
 FETCH NEXT FROM hierarchy\_cursor INTO @emp\_id;  
 END  
   
 CLOSE hierarchy\_cursor;  
 DEALLOCATE hierarchy\_cursor;  
END  
  
-- Execute (starting with top-level managers)  
EXEC sp\_GetEmployeeHierarchy @manager\_id = NULL;

## Performance Optimization

### Using Table Variables and Temp Tables

```sql CREATE PROCEDURE sp\_BulkSalaryUpdate @salary\_adjustments VARCHAR(MAX) – JSON or CSV format AS BEGIN SET NOCOUNT ON;

-- Create temp table for processing  
CREATE TABLE #salary\_updates (  
 employee\_id INT,  
 new\_salary DECIMAL(10,2),  
 adjustment\_reason VARCHAR(100)  
);  
  
-- For this example, manually insert test data  
-- In real scenario, you'd parse @salary\_adjustments parameter  
INSERT INTO #salary\_updates VALUES  
(1, 78000, 'Annual Review'),  
(2, 88000, 'Promotion'),  
(3, 57000, 'Market Adjustment');  
  
-- Table variable for results  
DECLARE @results TABLE (  
 employee\_id INT,  
 employee\_name VARCHAR(100),  
 old\_salary DECIMAL(10,2),  
 new\_salary DECIMAL(10,2),  
 adjustment\_amount DECIMAL(10,2),  
 adjustment\_percent DECIMAL(5,2),  
 status VARCHAR(50)  
);  
  
-- Process updates  
DECLARE @emp\_id INT, @new\_sal DECIMAL(10,2), @reason VARCHAR(100);  
DECLARE @old\_sal DECIMAL(10,2), @emp\_name VARCHAR(100);  
  
DECLARE update\_cursor CURSOR FOR  
SELECT employee\_id, new\_salary, adjustment\_reason FROM #salary\_updates;  
  
OPEN update\_cursor;  
FETCH NEXT FROM update\_cursor INTO @emp\_id, @new\_sal, @reason;  
  
WHILE @@FETCH\_STATUS = 0  
BEGIN  
 -- Get current salary  
 SELECT @old\_sal = salary, @emp\_name = first\_name + ' ' + last\_name  
 FROM employees  
 WHERE employee\_id = @emp\_id AND is\_active = 1;  
   
 IF @old\_sal IS NOT NULL  
 BEGIN  
 -- Update salary  
 UPDATE employees  
 SET salary = @new\_sal  
 WHERE employee\_id = @emp\_id;  
   
 -- Log result  
 INSERT INTO @results VALUES (  
 @emp\_id,  
 @emp\_name,  
 @old\_sal,  
 @new\_sal,  
 @new\_sal - @old\_sal,  
 CASE WHEN @old\_sal > 0 THEN ((@new\_sal - @old\_sal) / @old\_sal) \* 100 ELSE 0 END,  
 'Updated'  
 );  
 END