 SQL Advanced Notes & Practice Queries

# Already Covered Basics

* **Common Table Expression (CTE)**
* VIEWs
* Stored Procedures
* Indexes
* Triggers
* Window Functions
* Subqueries

# **Common Table Expression (CTE)**

# All About CTEs (Common Table Expressions)

## Table of Contents

1. [What is a CTE?](#what-is-a-cte)
2. [Basic CTE Syntax](#basic-cte-syntax)
3. [Simple CTE Examples](#simple-cte-examples)
4. [Multiple CTEs](#multiple-ctes)
5. [Finding Nth Values with CTE](#finding-nth-values-with-cte)
6. [CTE with Aggregations](#cte-with-aggregations)
7. [Data Transformation with CTE](#data-transformation-with-cte)
8. [Recursive CTE](#recursive-cte)
9. [Running Totals with CTE](#running-totals-with-cte)
10. [Date/Time Analysis](#datetime-analysis)
11. [Data Validation](#data-validation)
12. [Complex Filtering](#complex-filtering)
13. [CTE with CASE Statements](#cte-with-case-statements)
14. [CTE vs Alternatives](#cte-vs-alternatives)

## What is CTE?

A **Common Table Expression (CTE)** is a temporary named result set that exists only during the execution of a single SQL statement. It’s defined using the WITH clause and provides a way to create more readable and maintainable complex queries.

### Key Benefits:

* **Readability**: Makes complex queries more readable
* **Reusability**: Can reference the same CTE multiple times
* **Organization**: Breaks down complex logic into manageable steps
* **Recursion**: Supports recursive operations
* **Debugging**: Easier to test individual parts of complex queries

## Basic CTE Syntax

WITH cte\_name AS (  
 SELECT columns FROM table WHERE conditions  
)  
SELECT \* FROM cte\_name;

## Simple CTE Examples

### Basic Filtering

-- Find employees with salary > 50000  
WITH high\_earners AS (  
 SELECT name, salary, department   
 FROM employees   
 WHERE salary > 50000  
)  
SELECT \* FROM high\_earners ORDER BY salary DESC;

### CTE with Calculations

-- Calculate bonus (10% of salary) for each employee  
WITH employee\_bonus AS (  
 SELECT   
 name,   
 salary,   
 salary \* 0.10 as bonus,  
 salary + (salary \* 0.10) as total\_compensation  
 FROM employees  
)  
SELECT \* FROM employee\_bonus;

## Multiple CTEs

You can define multiple CTEs in a single query for complex operations:

WITH   
-- CTE 1: Department averages  
dept\_averages AS (  
 SELECT department, AVG(salary) as avg\_salary  
 FROM employees  
 GROUP BY department  
),  
-- CTE 2: Employee performance vs department average  
employee\_performance AS (  
 SELECT   
 e.name,  
 e.salary,  
 e.department,  
 d.avg\_salary,  
 CASE   
 WHEN e.salary > d.avg\_salary THEN 'Above Average'  
 ELSE 'Below Average'  
 END as performance  
 FROM employees e  
 JOIN dept\_averages d ON e.department = d.department  
)  
SELECT \* FROM employee\_performance;

## Finding Nth Values with CTE

### Find 3rd Highest Salary

WITH salary\_ranks AS (  
 SELECT   
 name,   
 salary,  
 ROW\_NUMBER() OVER(ORDER BY salary DESC) as rank  
 FROM employees  
)  
SELECT name, salary FROM salary\_ranks WHERE rank = 3;

### Find 2nd Lowest Salary by Department

WITH dept\_salary\_ranks AS (  
 SELECT   
 name,   
 department,   
 salary,  
 ROW\_NUMBER() OVER(PARTITION BY department ORDER BY salary ASC) as rank  
 FROM employees  
)  
SELECT name, department, salary   
FROM dept\_salary\_ranks   
WHERE rank = 2;

## CTE with Aggregations

### Department-wise Statistics

WITH dept\_stats AS (  
 SELECT   
 department,  
 COUNT(\*) as total\_employees,  
 SUM(salary) as total\_salary,  
 AVG(salary) as avg\_salary,  
 MAX(salary) as max\_salary,  
 MIN(salary) as min\_salary  
 FROM employees  
 GROUP BY department  
)  
SELECT   
 department,  
 total\_employees,  
 total\_salary,  
 ROUND(avg\_salary, 2) as average\_salary,  
 max\_salary - min\_salary as salary\_range  
FROM dept\_stats;

## Data Transformation with CTE

### Clean and Transform Employee Data

WITH clean\_employee\_data AS (  
 SELECT   
 UPPER(TRIM(name)) as clean\_name,  
 LOWER(TRIM(email)) as clean\_email,  
 salary,  
 CASE   
 WHEN department = 'IT' THEN 'Information Technology'  
 WHEN department = 'HR' THEN 'Human Resources'  
 ELSE department  
 END as full\_department\_name  
 FROM employees  
)  
SELECT \* FROM clean\_employee\_data;

## Recursive CTE

### Organizational Hierarchy

-- Find all employees in a management hierarchy  
WITH employee\_hierarchy AS (  
 -- Base case: Top-level managers (no manager)  
 SELECT employee\_id, name, manager\_id, 1 as level  
 FROM employees   
 WHERE manager\_id IS NULL  
   
 UNION ALL  
   
 -- Recursive case: Find employees under each manager  
 SELECT e.employee\_id, e.name, e.manager\_id, eh.level + 1  
 FROM employees e  
 INNER JOIN employee\_hierarchy eh ON e.manager\_id = eh.employee\_id  
)  
SELECT   
 employee\_id,  
 name,  
 level,  
 REPLICATE(' ', level - 1) + name as indented\_name  
FROM employee\_hierarchy  
ORDER BY level, name;

## Running Totals with CTE

### Calculate Running Total of Salaries

WITH salary\_running\_total AS (  
 SELECT   
 name,  
 salary,  
 SUM(salary) OVER(ORDER BY salary DESC) as running\_total  
 FROM employees  
)  
SELECT   
 name,  
 salary,  
 running\_total,  
 ROUND((running\_total \* 100.0) / (SELECT SUM(salary) FROM employees), 2) as percentage\_of\_total  
FROM salary\_running\_total;

## Date/Time Analysis

### Monthly Sales Analysis

-- Assuming we have a sales table  
WITH monthly\_sales AS (  
 SELECT   
 YEAR(sale\_date) as year,  
 MONTH(sale\_date) as month,  
 SUM(amount) as monthly\_total,  
 COUNT(\*) as transaction\_count  
 FROM sales  
 GROUP BY YEAR(sale\_date), MONTH(sale\_date)  
),  
sales\_with\_growth AS (  
 SELECT   
 year,  
 month,  
 monthly\_total,  
 transaction\_count,  
 LAG(monthly\_total) OVER(ORDER BY year, month) as previous\_month,  
 monthly\_total - LAG(monthly\_total) OVER(ORDER BY year, month) as growth  
 FROM monthly\_sales  
)  
SELECT   
 year,  
 month,  
 monthly\_total,  
 previous\_month,  
 COALESCE(growth, 0) as monthly\_growth,  
 CASE   
 WHEN previous\_month IS NULL THEN 'N/A'  
 WHEN growth > 0 THEN 'Increase'  
 WHEN growth < 0 THEN 'Decrease'  
 ELSE 'No Change'  
 END as trend  
FROM sales\_with\_growth;

## Data Validation

### Find Data Quality Issues

WITH data\_quality\_check AS (  
 SELECT   
 employee\_id,  
 name,  
 email,  
 salary,  
 CASE   
 WHEN name IS NULL OR TRIM(name) = '' THEN 'Missing Name'  
 WHEN email IS NULL OR email NOT LIKE '%@%' THEN 'Invalid Email'  
 WHEN salary <= 0 THEN 'Invalid Salary'  
 ELSE 'Valid'  
 END as data\_status  
 FROM employees  
)  
SELECT \* FROM data\_quality\_check   
WHERE data\_status != 'Valid';

## Complex Filtering

### Advanced Multi-Step Filtering

-- Find employees who earn more than department average   
-- and are in top 3 earners overall  
WITH   
dept\_avg AS (  
 SELECT department, AVG(salary) as avg\_salary  
 FROM employees  
 GROUP BY department  
),  
above\_dept\_avg AS (  
 SELECT e.\*, d.avg\_salary  
 FROM employees e  
 JOIN dept\_avg d ON e.department = d.department  
 WHERE e.salary > d.avg\_salary  
),  
top\_earners AS (  
 SELECT \*, ROW\_NUMBER() OVER(ORDER BY salary DESC) as overall\_rank  
 FROM above\_dept\_avg  
)  
SELECT name, department, salary, avg\_salary, overall\_rank  
FROM top\_earners  
WHERE overall\_rank <= 3;

## CTE with CASE Statements

### Categorize Employees and Calculate Adjusted Salary

WITH employee\_categories AS (  
 SELECT   
 name,  
 salary,  
 department,  
 CASE   
 WHEN salary >= 70000 THEN 'Senior'  
 WHEN salary >= 50000 THEN 'Mid-Level'  
 ELSE 'Junior'  
 END as category,  
 CASE   
 WHEN salary >= 70000 THEN salary \* 1.05 -- 5% bonus  
 WHEN salary >= 50000 THEN salary \* 1.03 -- 3% bonus  
 ELSE salary \* 1.02 -- 2% bonus  
 END as adjusted\_salary  
 FROM employees  
)  
SELECT   
 category,  
 COUNT(\*) as employee\_count,  
 AVG(salary) as avg\_current\_salary,  
 AVG(adjusted\_salary) as avg\_adjusted\_salary,  
 AVG(adjusted\_salary - salary) as avg\_bonus  
FROM employee\_categories  
GROUP BY category;

## Advanced CTE Examples

### CTE for Pivot-like Operations

-- Transform rows to columns using CTE  
WITH department\_salaries AS (  
 SELECT   
 name,  
 CASE WHEN department = 'IT' THEN salary END as IT\_salary,  
 CASE WHEN department = 'Sales' THEN salary END as Sales\_salary,  
 CASE WHEN department = 'HR' THEN salary END as HR\_salary  
 FROM employees  
)  
SELECT   
 SUM(IT\_salary) as total\_IT\_salary,  
 SUM(Sales\_salary) as total\_Sales\_salary,  
 SUM(HR\_salary) as total\_HR\_salary,  
 AVG(IT\_salary) as avg\_IT\_salary,  
 AVG(Sales\_salary) as avg\_Sales\_salary,  
 AVG(HR\_salary) as avg\_HR\_salary  
FROM department\_salaries;

### CTE for Ranking Analysis

-- Complex ranking with multiple criteria  
WITH employee\_rankings AS (  
 SELECT   
 name,  
 department,  
 salary,  
 ROW\_NUMBER() OVER(ORDER BY salary DESC) as overall\_rank,  
 ROW\_NUMBER() OVER(PARTITION BY department ORDER BY salary DESC) as dept\_rank,  
 DENSE\_RANK() OVER(ORDER BY salary DESC) as salary\_dense\_rank,  
 NTILE(4) OVER(ORDER BY salary DESC) as quartile  
 FROM employees  
)  
SELECT   
 name,  
 department,  
 salary,  
 overall\_rank,  
 dept\_rank,  
 salary\_dense\_rank,  
 CASE quartile  
 WHEN 1 THEN 'Top 25%'  
 WHEN 2 THEN 'Upper Middle 25%'  
 WHEN 3 THEN 'Lower Middle 25%'  
 WHEN 4 THEN 'Bottom 25%'  
 END as salary\_quartile  
FROM employee\_rankings  
ORDER BY overall\_rank;

### CTE for Gap Analysis

-- Find gaps in employee IDs  
WITH id\_gaps AS (  
 SELECT   
 employee\_id,  
 employee\_id - ROW\_NUMBER() OVER(ORDER BY employee\_id) as gap\_group  
 FROM employees  
),  
gap\_analysis AS (  
 SELECT   
 gap\_group,  
 MIN(employee\_id) as start\_id,  
 MAX(employee\_id) as end\_id,  
 COUNT(\*) as consecutive\_count  
 FROM id\_gaps  
 GROUP BY gap\_group  
)  
SELECT   
 start\_id,  
 end\_id,  
 consecutive\_count,  
 CASE   
 WHEN start\_id = end\_id THEN 'Single ID'  
 ELSE 'Range: ' + CAST(start\_id AS VARCHAR) + '-' + CAST(end\_id AS VARCHAR)  
 END as id\_range  
FROM gap\_analysis  
ORDER BY start\_id;

## CTE Best Practices

### 1. Use Meaningful Names

-- Good  
WITH high\_performing\_employees AS (...)  
  
-- Avoid  
WITH cte1 AS (...)

### 2. Break Complex Logic into Steps

-- Instead of one complex query, use multiple CTEs  
WITH   
step1\_filter AS (...),  
step2\_calculations AS (...),  
step3\_final\_format AS (...)  
SELECT \* FROM step3\_final\_format;

### 3. Add Comments for Complex CTEs

WITH   
-- Calculate department averages for performance comparison  
dept\_averages AS (  
 SELECT department, AVG(salary) as avg\_salary  
 FROM employees  
 GROUP BY department  
),  
-- Identify employees above department average  
above\_average\_performers AS (  
 SELECT e.\*, d.avg\_salary  
 FROM employees e  
 JOIN dept\_averages d ON e.department = d.department  
 WHERE e.salary > d.avg\_salary  
)  
SELECT \* FROM above\_average\_performers;

## CTE vs Alternatives

| Feature | CTE | Subquery | Temporary Table | View |
| --- | --- | --- | --- | --- |
| **Scope** | Single statement | Single statement | Session | Permanent |
| **Reusability** | Within query | No | Yes | Yes |
| **Performance** | Good | Variable | Best | Good |
| **Memory Usage** | Low | Low | High | Low |
| **Recursion** | Yes | No | No | No |
| **Readability** | Excellent | Poor for complex | Good | Good |
| **Maintenance** | Easy | Difficult | Moderate | Easy |

### When to Use CTEs:

* **Complex queries** that need to be broken down
* **Recursive operations** (hierarchical data)
* **Multiple references** to the same subquery
* **Improved readability** is priority
* **Temporary calculations** within a single query

### When to Avoid CTEs:

* **Simple queries** where a direct SELECT works
* **Performance critical** operations (consider temp tables)
* **Cross-session** data sharing (use temp tables or views)
* **Very large datasets** (temp tables might be better)

## Summary

CTEs are powerful tools for: - Making complex SQL queries more readable and maintainable - Breaking down complex logic into manageable steps - Handling recursive operations elegantly - Improving code organization and debugging - Creating reusable query components within a single statement

# **VIEW**

# Complete Guide to Views in SSMS

## Table of Contents

1. [What is a View?](#what-is-a-view)
2. [Types of Views](#types-of-views)
3. [Basic View Syntax](#basic-view-syntax)
4. [Creating Simple Views](#creating-simple-views)
5. [Views with Joins](#views-with-joins)
6. [Views with Aggregations](#views-with-aggregations)
7. [Views with Calculations](#views-with-calculations)
8. [Parameterized Views (Table-Valued Functions)](#parameterized-views)
9. [Indexed Views (Materialized Views)](#indexed-views)
10. [Updating Data Through Views](#updating-data-through-views)
11. [View Security and Permissions](#view-security-and-permissions)
12. [View Management in SSMS](#view-management-in-ssms)
13. [Best Practices](#best-practices)
14. [Views vs Alternatives](#views-vs-alternatives)

## What is a View?

A **View** in SQL Server is a virtual table that doesn’t store data physically. Instead, it stores a SQL query that generates data dynamically when the view is accessed. Views provide a way to present data from one or more tables in a specific format without duplicating the underlying data.

### Key Benefits:

* **Data Security**: Hide sensitive columns and restrict data access
* **Simplification**: Present complex joins in a simple format
* **Consistency**: Standardize data presentation across applications
* **Abstraction**: Hide database schema complexity from users
* **Reusability**: Reuse common query logic across multiple applications

## Types of Views

### 1. Simple Views

* Based on a single table
* Can perform DML operations (INSERT, UPDATE, DELETE)

### 2. Complex Views

* Based on multiple tables with joins
* May include GROUP BY, aggregate functions
* Limited DML operations

### 3. Indexed Views (Materialized Views)

* Physically store data for performance
* Automatically updated when base tables change

## Basic View Syntax

-- Create View  
CREATE VIEW view\_name AS  
SELECT column1, column2, ...  
FROM table\_name  
WHERE condition;  
  
-- Use View  
SELECT \* FROM view\_name;  
  
-- Modify View  
ALTER VIEW view\_name AS  
SELECT column1, column2, ...  
FROM table\_name  
WHERE new\_condition;  
  
-- Drop View  
DROP VIEW view\_name;

## Creating Simple Views

Let’s start with sample data:

-- Create sample tables  
CREATE TABLE employees (  
 employee\_id INT 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 PRIMARY KEY,  
 dept\_name VARCHAR(50),  
 location VARCHAR(50),  
 budget DECIMAL(12,2)  
);  
  
-- Insert sample data  
INSERT INTO employees VALUES  
(1, 'John', 'Doe', 'john.doe@company.com', '555-1001', 'IT', 'Developer', 75000, '2020-01-15', NULL, 1),  
(2, 'Jane', 'Smith', 'jane.smith@company.com', '555-1002', 'IT', 'Senior Developer', 85000, '2019-03-10', 1, 1),  
(3, 'Bob', 'Johnson', 'bob.johnson@company.com', '555-1003', 'Sales', 'Sales Rep', 55000, '2021-06-20', NULL, 1),  
(4, 'Alice', 'Brown', 'alice.brown@company.com', '555-1004', 'HR', 'HR Manager', 70000, '2018-09-05', NULL, 1),  
(5, 'Charlie', 'Wilson', 'charlie.wilson@company.com', '555-1005', 'IT', 'Junior Developer', 60000, '2022-02-14', 2, 1),  
(6, 'Diana', 'Ross', 'diana.ross@company.com', '555-1006', 'Sales', 'Sales Manager', 80000, '2017-11-30', NULL, 1),  
(7, 'Elvis', 'King', 'elvis.king@company.com', '555-1007', 'Finance', 'Accountant', 65000, '2020-08-12', NULL, 0);  
  
INSERT INTO departments VALUES  
(1, 'Information Technology', 'New York', 500000),  
(2, 'Sales', 'Chicago', 300000),  
(3, 'Human Resources', 'Los Angeles', 200000),  
(4, 'Finance', 'Boston', 250000);

### Basic Employee View

-- Create a view showing active employees only  
CREATE VIEW vw\_active\_employees AS  
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;  
  
-- Use the view  
SELECT \* FROM vw\_active\_employees;  
SELECT \* FROM vw\_active\_employees WHERE department = 'IT';

### Public Employee Directory View

-- Create a view hiding sensitive information  
CREATE VIEW vw\_employee\_directory AS  
SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS full\_name,  
 department,  
 position,  
 email,  
 -- Hide salary and other sensitive data  
 CASE   
 WHEN phone IS NOT NULL THEN 'Available'  
 ELSE 'Not Available'  
 END AS contact\_available  
FROM employees  
WHERE is\_active = 1;  
  
-- Usage  
SELECT \* FROM vw\_employee\_directory ORDER BY full\_name;

## Views with Joins

### Employee Department View

-- Create view joining employees with departments  
CREATE VIEW vw\_employee\_department AS  
SELECT   
 e.employee\_id,  
 e.first\_name + ' ' + e.last\_name AS employee\_name,  
 e.email,  
 e.position,  
 e.salary,  
 e.hire\_date,  
 d.dept\_name AS department\_name,  
 d.location AS department\_location,  
 d.budget AS department\_budget  
FROM employees e  
LEFT JOIN departments d ON e.department = d.dept\_name  
WHERE e.is\_active = 1;  
  
-- Usage  
SELECT \* FROM vw\_employee\_department;  
SELECT \* FROM vw\_employee\_department WHERE department\_location = 'New York';

### Employee Hierarchy View

-- Create view showing manager-employee relationships  
CREATE VIEW vw\_employee\_hierarchy AS  
SELECT   
 e.employee\_id,  
 e.first\_name + ' ' + e.last\_name AS employee\_name,  
 e.department,  
 e.position,  
 e.salary,  
 m.first\_name + ' ' + m.last\_name AS manager\_name,  
 m.position AS manager\_position  
FROM employees e  
LEFT JOIN employees m ON e.manager\_id = m.employee\_id  
WHERE e.is\_active = 1;  
  
-- Usage  
SELECT \* FROM vw\_employee\_hierarchy ORDER BY manager\_name, employee\_name;

### Complex Multi-Table View

-- Create comprehensive employee view  
CREATE VIEW vw\_employee\_complete AS  
SELECT   
 e.employee\_id,  
 e.first\_name + ' ' + e.last\_name AS employee\_name,  
 e.email,  
 e.phone,  
 e.position,  
 e.salary,  
 e.hire\_date,  
 DATEDIFF(YEAR, e.hire\_date, GETDATE()) AS years\_of\_service,  
 d.dept\_name AS department\_name,  
 d.location AS office\_location,  
 d.budget AS department\_budget,  
 m.first\_name + ' ' + m.last\_name AS manager\_name,  
 CASE   
 WHEN e.salary >= 80000 THEN 'Senior'  
 WHEN e.salary >= 65000 THEN 'Mid-Level'  
 ELSE 'Junior'  
 END AS employee\_level  
FROM employees e  
LEFT JOIN departments d ON e.department = d.dept\_name  
LEFT JOIN employees m ON e.manager\_id = m.employee\_id  
WHERE e.is\_active = 1;  
  
-- Usage  
SELECT \* FROM vw\_employee\_complete WHERE employee\_level = 'Senior';

## Views with Aggregations

### Department Statistics View

CREATE VIEW vw\_department\_stats AS  
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\_cost,  
 MAX(salary) - MIN(salary) AS salary\_range  
FROM employees  
WHERE is\_active = 1  
GROUP BY department;  
  
-- Usage  
SELECT \* FROM vw\_department\_stats ORDER BY avg\_salary DESC;

### Monthly Hiring Trends View

CREATE VIEW vw\_hiring\_trends AS  
SELECT   
 YEAR(hire\_date) AS hire\_year,  
 MONTH(hire\_date) AS hire\_month,  
 DATENAME(MONTH, hire\_date) AS month\_name,  
 COUNT(\*) AS employees\_hired,  
 AVG(salary) AS avg\_starting\_salary  
FROM employees  
GROUP BY YEAR(hire\_date), MONTH(hire\_date), DATENAME(MONTH, hire\_date);  
  
-- Usage  
SELECT \* FROM vw\_hiring\_trends ORDER BY hire\_year, hire\_month;

### Salary Analysis View

CREATE VIEW vw\_salary\_analysis AS  
SELECT   
 position,  
 department,  
 COUNT(\*) AS position\_count,  
 AVG(salary) AS avg\_salary,  
 MIN(salary) AS min\_salary,  
 MAX(salary) AS max\_salary,  
 STDEV(salary) AS salary\_std\_dev,  
 CASE   
 WHEN COUNT(\*) > 1 THEN (MAX(salary) - MIN(salary)) / NULLIF(AVG(salary), 0) \* 100  
 ELSE 0  
 END AS salary\_variation\_percent  
FROM employees  
WHERE is\_active = 1  
GROUP BY position, department;  
  
-- Usage  
SELECT \* FROM vw\_salary\_analysis   
WHERE position\_count > 1   
ORDER BY salary\_variation\_percent DESC;

## Views with Calculations

### Employee Performance Metrics View

CREATE VIEW vw\_employee\_metrics AS  
SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 department,  
 position,  
 salary,  
 hire\_date,  
 DATEDIFF(DAY, hire\_date, GETDATE()) AS days\_employed,  
 DATEDIFF(YEAR, hire\_date, GETDATE()) AS years\_of\_service,  
 salary / 12.0 AS monthly\_salary,  
 salary / 52.0 AS weekly\_salary,  
 salary / 2080.0 AS hourly\_rate, -- Assuming 40 hours/week \* 52 weeks  
 CASE   
 WHEN DATEDIFF(YEAR, hire\_date, GETDATE()) < 1 THEN 'New Hire'  
 WHEN DATEDIFF(YEAR, hire\_date, GETDATE()) < 3 THEN 'Junior'  
 WHEN DATEDIFF(YEAR, hire\_date, GETDATE()) < 8 THEN 'Experienced'  
 ELSE 'Veteran'  
 END AS experience\_level,  
 salary \* 1.15 AS total\_cost\_with\_benefits -- Assuming 15% benefits cost  
FROM employees  
WHERE is\_active = 1;  
  
-- Usage  
SELECT \* FROM vw\_employee\_metrics WHERE experience\_level = 'Veteran';

### Budget Analysis View

CREATE VIEW vw\_budget\_analysis AS  
SELECT   
 d.dept\_name AS department,  
 d.location,  
 d.budget AS allocated\_budget,  
 COUNT(e.employee\_id) AS employee\_count,  
 SUM(e.salary) AS total\_salaries,  
 SUM(e.salary \* 1.15) AS total\_cost\_with\_benefits,  
 d.budget - SUM(e.salary \* 1.15) AS remaining\_budget,  
 CASE   
 WHEN d.budget - SUM(e.salary \* 1.15) > 0 THEN 'Under Budget'  
 WHEN d.budget - SUM(e.salary \* 1.15) = 0 THEN 'On Budget'  
 ELSE 'Over Budget'  
 END AS budget\_status,  
 (SUM(e.salary \* 1.15) / NULLIF(d.budget, 0)) \* 100 AS budget\_utilization\_percent  
FROM departments d  
LEFT JOIN employees e ON d.dept\_name = e.department AND e.is\_active = 1  
GROUP BY d.dept\_name, d.location, d.budget;  
  
-- Usage  
SELECT \* FROM vw\_budget\_analysis ORDER BY budget\_utilization\_percent DESC;

## Parameterized Views (Table-Valued Functions)

Since views can’t accept parameters directly, we use Table-Valued Functions:

### Inline Table-Valued Function (Acts like a parameterized view)

-- Create a function that works like a parameterized view  
CREATE FUNCTION fn\_employees\_by\_department(@department VARCHAR(50))  
RETURNS TABLE  
AS  
RETURN  
(  
 SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 email,  
 position,  
 salary,  
 hire\_date  
 FROM employees  
 WHERE department = @department AND is\_active = 1  
);  
  
-- Usage  
SELECT \* FROM fn\_employees\_by\_department('IT');

### Multi-Statement Table-Valued Function

CREATE FUNCTION fn\_employee\_salary\_range(  
 @min\_salary DECIMAL(10,2),  
 @max\_salary DECIMAL(10,2)  
)  
RETURNS @result TABLE (  
 employee\_id INT,  
 employee\_name VARCHAR(100),  
 department VARCHAR(50),  
 position VARCHAR(50),  
 salary DECIMAL(10,2),  
 salary\_category VARCHAR(20)  
)  
AS  
BEGIN  
 INSERT INTO @result  
 SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name,  
 department,  
 position,  
 salary,  
 CASE   
 WHEN salary >= (@min\_salary + @max\_salary) / 2 THEN 'High'  
 ELSE 'Low'  
 END  
 FROM employees  
 WHERE salary BETWEEN @min\_salary AND @max\_salary  
 AND is\_active = 1;  
   
 RETURN;  
END;  
  
-- Usage  
SELECT \* FROM fn\_employee\_salary\_range(60000, 80000);

## Indexed Views (Materialized Views)

Indexed views physically store data and can significantly improve performance for complex queries.

### Creating an Indexed View

-- First create the view with specific requirements  
CREATE VIEW vw\_department\_summary  
WITH SCHEMABINDING  
AS  
SELECT   
 department,  
 COUNT\_BIG(\*) AS employee\_count,  
 SUM(salary) AS total\_salary,  
 AVG(salary) AS avg\_salary  
FROM dbo.employees  
WHERE is\_active = 1  
GROUP BY department;  
  
-- Create unique clustered index on the view  
CREATE UNIQUE CLUSTERED INDEX IX\_vw\_department\_summary   
ON vw\_department\_summary (department);  
  
-- Optionally create additional indexes  
CREATE NONCLUSTERED INDEX IX\_vw\_department\_summary\_salary   
ON vw\_department\_summary (total\_salary);  
  
-- Usage (SQL Server may automatically use the indexed view)  
SELECT \* FROM vw\_department\_summary;

### Requirements for Indexed Views:

* Must use SCHEMABINDING
* Must reference tables with schema prefix (dbo.table\_name)
* Cannot use certain functions (e.g., GETDATE(), USER)
* First index must be unique and clustered
* Must use COUNT\_BIG instead of COUNT

## Updating Data Through Views

### Simple View Updates

-- Create updatable view  
CREATE VIEW vw\_employee\_basic AS  
SELECT   
 employee\_id,  
 first\_name,  
 last\_name,  
 email,  
 phone,  
 position,  
 salary  
FROM employees  
WHERE is\_active = 1;  
  
-- Update through view  
UPDATE vw\_employee\_basic   
SET salary = 77000   
WHERE employee\_id = 1;  
  
-- Insert through view  
INSERT INTO vw\_employee\_basic (first\_name, last\_name, email, position, salary)  
VALUES ('New', 'Employee', 'new.employee@company.com', 'Analyst', 50000);  
  
-- Delete through view  
DELETE FROM vw\_employee\_basic WHERE employee\_id = 7;

### View with CHECK OPTION

-- Create view with CHECK OPTION to maintain data integrity  
CREATE VIEW vw\_high\_salary\_employees AS  
SELECT   
 employee\_id,  
 first\_name,  
 last\_name,  
 email,  
 salary,  
 department  
FROM employees  
WHERE salary >= 70000  
WITH CHECK OPTION;  
  
-- This will succeed  
UPDATE vw\_high\_salary\_employees SET salary = 75000 WHERE employee\_id = 1;  
  
-- This will fail due to CHECK OPTION  
UPDATE vw\_high\_salary\_employees SET salary = 65000 WHERE employee\_id = 1;

### INSTEAD OF Triggers for Complex Views

-- Create view that can't be directly updated  
CREATE VIEW vw\_employee\_department\_info AS  
SELECT   
 e.employee\_id,  
 e.first\_name + ' ' + e.last\_name AS full\_name,  
 e.email,  
 e.salary,  
 d.dept\_name,  
 d.location  
FROM employees e  
JOIN departments d ON e.department = d.dept\_name;  
  
-- Create INSTEAD OF trigger to handle updates  
CREATE TRIGGER tr\_update\_employee\_department  
ON vw\_employee\_department\_info  
INSTEAD OF UPDATE  
AS  
BEGIN  
 UPDATE employees   
 SET   
 email = i.email,  
 salary = i.salary  
 FROM employees e  
 INNER JOIN inserted i ON e.employee\_id = i.employee\_id;  
END;  
  
-- Now we can update through the view  
UPDATE vw\_employee\_department\_info   
SET salary = 80000   
WHERE employee\_id = 1;

## View Security and Permissions

### Column-Level Security

-- Create view that hides sensitive columns  
CREATE VIEW vw\_employee\_public AS  
SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS full\_name,  
 department,  
 position,  
 -- Hide exact salary, show only range  
 CASE   
 WHEN salary >= 80000 THEN '$80,000+'  
 WHEN salary >= 60000 THEN '$60,000-$79,999'  
 WHEN salary >= 40000 THEN '$40,000-$59,999'  
 ELSE 'Below $40,000'  
 END AS salary\_range,  
 hire\_date  
FROM employees  
WHERE is\_active = 1;

### Row-Level Security with Views

-- Create view for department managers (they can only see their department)  
CREATE VIEW vw\_manager\_employees AS  
SELECT   
 employee\_id,  
 first\_name,  
 last\_name,  
 email,  
 position,  
 salary,  
 hire\_date,  
 department  
FROM employees  
WHERE department = (  
 SELECT department   
 FROM employees   
 WHERE employee\_id = USER\_ID() -- Assuming USER\_ID() returns current user's employee\_id  
) AND is\_active = 1;

### Granting Permissions on Views

-- Grant permissions on view without giving access to underlying tables  
GRANT SELECT ON vw\_employee\_public TO [PublicUsers];  
GRANT SELECT, INSERT, UPDATE ON vw\_employee\_basic TO [HRManagers];  
GRANT SELECT ON vw\_department\_stats TO [Executives];  
  
-- Revoke permissions  
REVOKE SELECT ON vw\_employee\_public FROM [PublicUsers];

## View Management in SSMS

### Using SSMS Interface

1. **Creating Views through SSMS:**
   * Right-click on “Views” folder → “New View”
   * Use graphical query designer or write SQL directly
   * Save with appropriate naming convention
2. **Modifying Views:**
   * Right-click view → “Design” for graphical editor
   * Right-click view → “Script View as” → “ALTER To” for SQL script
3. **Viewing Dependencies:**
   * Right-click view → “View Dependencies”
   * Shows which objects depend on this view and what this view depends on
4. **Checking View Definition:**
   * Right-click view → “Script View as” → “CREATE To”
   * Use system views: SELECT \* FROM INFORMATION\_SCHEMA.VIEWS

### System Views for View Information

-- List all views in database  
SELECT   
 TABLE\_SCHEMA,  
 TABLE\_NAME,  
 VIEW\_DEFINITION  
FROM INFORMATION\_SCHEMA.VIEWS;  
  
-- Get view definition  
SELECT OBJECT\_DEFINITION(OBJECT\_ID('vw\_employee\_directory'));  
  
-- Check view dependencies  
SELECT   
 o.name AS dependent\_object,  
 o.type\_desc,  
 d.referenced\_entity\_name  
FROM sys.sql\_dependencies d  
JOIN sys.objects o ON d.object\_id = o.object\_id  
WHERE d.referenced\_entity\_name = 'employees';

## Advanced View Examples

### Pivot View for Salary Comparison

CREATE VIEW vw\_salary\_by\_department\_position AS  
SELECT   
 position,  
 ISNULL([IT], 0) AS IT\_salary,  
 ISNULL([Sales], 0) AS Sales\_salary,  
 ISNULL([HR], 0) AS HR\_salary,  
 ISNULL([Finance], 0) AS Finance\_salary  
FROM (  
 SELECT position, department, AVG(salary) AS avg\_salary  
 FROM employees  
 WHERE is\_active = 1  
 GROUP BY position, department  
) AS SourceTable  
PIVOT (  
 AVG(avg\_salary)  
 FOR department IN ([IT], [Sales], [HR], [Finance])  
) AS PivotTable;  
  
-- Usage  
SELECT \* FROM vw\_salary\_by\_department\_position;

### Ranking View

CREATE VIEW vw\_employee\_rankings AS  
SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 department,  
 salary,  
 ROW\_NUMBER() OVER(ORDER BY salary DESC) AS overall\_rank,  
 ROW\_NUMBER() OVER(PARTITION BY department ORDER BY salary DESC) AS dept\_rank,  
 DENSE\_RANK() OVER(ORDER BY salary DESC) AS salary\_dense\_rank,  
 PERCENT\_RANK() OVER(ORDER BY salary) AS salary\_percentile,  
 NTILE(4) OVER(ORDER BY salary) AS salary\_quartile  
FROM employees  
WHERE is\_active = 1;  
  
-- Usage  
SELECT \* FROM vw\_employee\_rankings WHERE dept\_rank <= 3;

### Time-Based Analysis View

CREATE VIEW vw\_employee\_tenure\_analysis AS  
SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 department,  
 salary,  
 hire\_date,  
 DATEDIFF(MONTH, hire\_date, GETDATE()) AS months\_employed,  
 CASE   
 WHEN DATEDIFF(MONTH, hire\_date, GETDATE()) <= 6 THEN 'New (0-6 months)'  
 WHEN DATEDIFF(MONTH, hire\_date, GETDATE()) <= 24 THEN 'Growing (6-24 months)'  
 WHEN DATEDIFF(MONTH, hire\_date, GETDATE()) <= 60 THEN 'Experienced (2-5 years)'  
 ELSE 'Veteran (5+ years)'  
 END AS tenure\_category,  
 salary / NULLIF(DATEDIFF(MONTH, hire\_date, GETDATE()), 0) AS salary\_per\_month\_tenure,  
 CASE   
 WHEN DATEPART(QUARTER, hire\_date) = 1 THEN 'Q1'  
 WHEN DATEPART(QUARTER, hire\_date) = 2 THEN 'Q2'  
 WHEN DATEPART(QUARTER, hire\_date) = 3 THEN 'Q3'  
 ELSE 'Q4'  
 END AS hire\_quarter  
FROM employees  
WHERE is\_active = 1;  
  
-- Usage  
SELECT tenure\_category, COUNT(\*) as employee\_count, AVG(salary) as avg\_salary  
FROM vw\_employee\_tenure\_analysis  
GROUP BY tenure\_category;

## Best Practices

### 1. Naming Conventions

-- Use consistent prefixes  
CREATE VIEW vw\_employee\_summary AS ... -- "vw\_" prefix  
CREATE VIEW v\_department\_stats AS ... -- "v\_" prefix (alternative)

### 2. Documentation and Comments

/\*  
View: vw\_employee\_complete  
Purpose: Comprehensive employee information for reporting  
Author: Database Team  
Created: 2023-01-01  
Last Modified: 2023-06-01  
Notes: Includes department and manager information  
\*/  
CREATE VIEW vw\_employee\_complete AS  
SELECT   
 -- Employee basic info  
 e.employee\_id,  
 e.first\_name + ' ' + e.last\_name AS employee\_name,  
 -- ... other columns  
FROM employees e  
-- Join with departments for location info  
LEFT JOIN departments d ON e.department = d.dept\_name  
-- Join with managers  
LEFT JOIN employees m ON e.manager\_id = m.employee\_id  
WHERE e.is\_active = 1;

### 3. Performance Considerations

-- Create indexes on base tables for columns used in views  
CREATE INDEX IX\_employees\_department ON employees(department) WHERE is\_active = 1;  
CREATE INDEX IX\_employees\_salary ON employees(salary) WHERE is\_active = 1;  
  
-- Use appropriate WHERE clauses in views  
CREATE VIEW vw\_active\_employees AS  
SELECT \* FROM employees   
WHERE is\_active = 1 -- Filter early for performance  
AND hire\_date IS NOT NULL;

### 4. Avoid SELECT \* in Views

-- Good: Specify columns explicitly  
CREATE VIEW vw\_employee\_basic AS  
SELECT   
 employee\_id,  
 first\_name,  
 last\_name,  
 email,  
 department  
FROM employees;  
  
-- Avoid: SELECT \* makes view fragile  
CREATE VIEW vw\_employee\_all AS  
SELECT \* FROM employees; -- Avoid this

## Views vs Alternatives

| Feature | Views | CTEs | Stored Procedures | Table Functions |
| --- | --- | --- | --- | --- |
| **Reusability** | High | Single Query | High | High |
| **Parameters** | No | No | Yes | Yes |
| **Performance** | Good | Good | Excellent | Good |
| **Security** | Excellent | N/A | Good | Good |
| **Data Persistence** | No | No | No | No |
| **Indexed** | Yes (with restrictions) | No | No | No |
| **DML Operations** | Limited | No | Yes | No |

### When to Use Views:

* **Data Security**: Hide sensitive columns or rows
* **Simplification**: Present complex joins simply
* **Standardization**: Consistent data presentation
* **Legacy System Integration**: Abstract schema changes
* **Reporting**: Pre-built queries for reports

### When Not to Use Views:

* **Parameter Requirements**: Use functions instead
* **Complex Logic**: Use stored procedures
* **Temporary Results**: Use CTEs
* **Performance Critical**: Consider indexed views or tables

## Troubleshooting Common Issues

### Issue 1: View Not Updatable

-- Problem: Complex view with joins can't be updated  
-- Solution: Create INSTEAD OF triggers or use stored procedures  
  
CREATE TRIGGER tr\_update\_complex\_view  
ON vw\_complex\_view  
INSTEAD OF UPDATE  
AS  
BEGIN  
 -- Custom update logic here  
END;

### Issue 2: View Performance Problems

-- Problem: View is slow  
-- Solutions:  
-- 1. Add indexes to base tables  
CREATE INDEX IX\_employees\_dept\_salary ON employees(department, salary);  
  
-- 2. Create indexed view (if possible)  
-- 3. Use NOEXPAND hint for indexed views  
SELECT \* FROM vw\_department\_summary WITH (NOEXPAND);

### Issue 3: Schema Binding Issues

-- Problem: Can't create indexed view  
-- Solution: Use SCHEMABINDING and proper syntax  
CREATE VIEW vw\_indexed\_view  
WITH SCHEMABINDING  
AS  
SELECT   
 department,  
 COUNT\_BIG(\*) AS employee\_count  
FROM dbo.employees -- Must use schema prefix  
GROUP BY department;

## Summary

Views in SQL Server are powerful tools for: - **Data Security**: Controlling access to sensitive information - **Code Reusability**: Creating reusable query logic - **Simplification**: Making complex queries accessible - **Performance**: Improving query performance with indexed views - **Maintenance**: Centralizing business logic in the database

# **Stored Procedure**

# 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

# **INDEXES**

# Complete Guide to Indexes in SSMS

## Table of Contents

1. [What is an Index?](#what-is-an-index)
2. [Types of Indexes](#types-of-indexes)
3. [Index Structure and Storage](#index-structure-and-storage)
4. [Creating Indexes - Basic Syntax](#creating-indexes---basic-syntax)
5. [Clustered Indexes](#clustered-indexes)
6. [Non-Clustered Indexes](#non-clustered-indexes)
7. [Unique Indexes](#unique-indexes)
8. [Composite Indexes](#composite-indexes)
9. [Covering Indexes](#covering-indexes)
10. [Filtered Indexes](#filtered-indexes)
11. [Column store Indexes](#columnstore-indexes)
12. [Index Management](#index-management)
13. [Index Performance Analysis](#index-performance-analysis)
14. [Index Optimization Strategies](#index-optimization-strategies)
15. [Managing Indexes in SSMS](#managing-indexes-in-ssms)
16. [Best Practices](#best-practices)

## What is an Index?

An **Index** in SQL Server is a database object that improves the speed of data retrieval operations on a table. It creates shortcuts to data, similar to an index in a book, allowing the database engine to find rows quickly without scanning the entire table.

### Key Benefits:

* **Faster Query Performance**: Dramatically reduces query execution time
* **Efficient Sorting**: ORDER BY operations execute faster
* **Quick Joins**: JOIN operations between tables are optimized
* **Unique Constraint Enforcement**: Ensures data uniqueness

### Costs:

* **Storage Space**: Indexes require additional disk space
* **Maintenance Overhead**: INSERT, UPDATE, DELETE operations become slower
* **Memory Usage**: Indexes consume memory when loaded

## Types of Indexes

### 1. Clustered Index

* **One per table**: Only one clustered index per table
* **Data Storage**: Table data is physically stored in index order
* **Leaf Level**: Contains actual table data

### 2. Non-Clustered Index

* **Multiple allowed**: Up to 999 non-clustered indexes per table
* **Separate Structure**: Index structure separate from table data
* **Leaf Level**: Contains pointers to table data

### 3. Unique Index

* **Data Integrity**: Enforces uniqueness of values
* **Can be Clustered or Non-Clustered**

### 4. Composite Index

* **Multiple Columns**: Index on two or more columns
* **Column Order Matters**: First column is most important

### 5. Covering Index

* **Includes Non-Key Columns**: Additional columns for covering queries
* **Reduces Key Lookups**: All required data in index

### 6. Filtered Index

* **Conditional**: Index on subset of rows based on WHERE condition
* **Space Efficient**: Smaller than full table index

### 7. Columnstore Index

* **Columnar Storage**: Data stored by columns instead of rows
* **Analytics Optimized**: Excellent for data warehousing queries

## Index Structure and Storage

### B-Tree Structure

Root Level  
 |  
 Intermediate Levels  
 / \  
 Leaf Level Leaf Level  
 (Data Pages) (Data Pages)

### Sample Tables Setup

-- Create sample tables for demonstrations  
CREATE TABLE employees (  
 employee\_id INT IDENTITY(1,1),  
 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,  
 last\_updated DATETIME DEFAULT GETDATE()  
);  
  
CREATE TABLE orders (  
 order\_id INT IDENTITY(1,1),  
 customer\_id INT,  
 employee\_id INT,  
 order\_date DATETIME,  
 ship\_date DATETIME,  
 total\_amount DECIMAL(10,2),  
 status VARCHAR(20),  
 region VARCHAR(50)  
);  
  
CREATE TABLE order\_details (  
 detail\_id INT IDENTITY(1,1),  
 order\_id INT,  
 product\_id INT,  
 quantity INT,  
 unit\_price DECIMAL(8,2),  
 discount DECIMAL(3,2)  
);  
  
-- Insert sample data  
INSERT INTO employees VALUES  
('John', 'Doe', 'john.doe@company.com', '555-1001', 'IT', 'Developer', 75000, '2020-01-15', NULL, 1, GETDATE()),  
('Jane', 'Smith', 'jane.smith@company.com', '555-1002', 'IT', 'Senior Developer', 85000, '2019-03-10', 1, 1, GETDATE()),  
('Bob', 'Johnson', 'bob.johnson@company.com', '555-1003', 'Sales', 'Sales Rep', 55000, '2021-06-20', NULL, 1, GETDATE()),  
('Alice', 'Brown', 'alice.brown@company.com', '555-1004', 'HR', 'HR Manager', 70000, '2018-09-05', NULL, 1, GETDATE());  
  
-- Generate more sample data for demonstration  
DECLARE @i INT = 1;  
WHILE @i <= 1000  
BEGIN  
 INSERT INTO orders VALUES  
 (@i % 100 + 1, @i % 4 + 1, DATEADD(day, -@i, GETDATE()),   
 DATEADD(day, -@i + 3, GETDATE()), (@i \* 123.45) % 10000,   
 CASE @i % 4 WHEN 0 THEN 'Completed' WHEN 1 THEN 'Pending' WHEN 2 THEN 'Shipped' ELSE 'Cancelled' END,  
 CASE @i % 3 WHEN 0 THEN 'North' WHEN 1 THEN 'South' ELSE 'West' END);  
 SET @i = @i + 1;  
END;

## Creating Indexes - Basic Syntax

### Basic Index Creation

-- Create Non-Clustered Index  
CREATE INDEX IX\_IndexName ON TableName (ColumnName);  
  
-- Create Unique Index  
CREATE UNIQUE INDEX IX\_IndexName ON TableName (ColumnName);  
  
-- Create Clustered Index  
CREATE CLUSTERED INDEX IX\_IndexName ON TableName (ColumnName);  
  
-- Create Composite Index  
CREATE INDEX IX\_IndexName ON TableName (Column1, Column2, Column3);  
  
-- Drop Index  
DROP INDEX IX\_IndexName ON TableName;  
  
-- Disable Index  
ALTER INDEX IX\_IndexName ON TableName DISABLE;  
  
-- Rebuild Index  
ALTER INDEX IX\_IndexName ON TableName REBUILD;  
  
-- Reorganize Index  
ALTER INDEX IX\_IndexName ON TableName REORGANIZE;

## Clustered Indexes

### Primary Key Clustered Index (Automatic)

-- When creating primary key, clustered index is created automatically  
ALTER TABLE employees   
ADD CONSTRAINT PK\_employees PRIMARY KEY CLUSTERED (employee\_id);  
  
-- View the clustered index  
SELECT   
 i.name AS index\_name,  
 i.type\_desc,  
 i.is\_unique,  
 i.is\_primary\_key,  
 c.name AS column\_name  
FROM sys.indexes i  
JOIN sys.index\_columns ic ON i.object\_id = ic.object\_id AND i.index\_id = ic.index\_id  
JOIN sys.columns c ON ic.object\_id = c.object\_id AND ic.column\_id = c.column\_id  
WHERE i.object\_id = OBJECT\_ID('employees')  
AND i.type\_desc = 'CLUSTERED';

### Custom Clustered Index

-- Create table without primary key first  
CREATE TABLE sales\_data (  
 sale\_id INT,  
 sale\_date DATE,  
 amount DECIMAL(10,2),  
 region VARCHAR(50)  
);  
  
-- Create clustered index on date column (for time-series data)  
CREATE CLUSTERED INDEX IX\_sales\_data\_date ON sales\_data (sale\_date, region);  
  
-- Insert data to see physical ordering  
INSERT INTO sales\_data VALUES  
(1, '2023-01-15', 1000.00, 'North'),  
(3, '2023-01-10', 1500.00, 'South'),  
(2, '2023-01-20', 1200.00, 'East'),  
(5, '2023-01-05', 800.00, 'West'),  
(4, '2023-01-25', 2000.00, 'North');  
  
-- Data will be physically stored ordered by sale\_date, then region  
SELECT \* FROM sales\_data; -- Notice the physical order

### Clustered Index Performance Benefits

-- Query that benefits from clustered index  
SELECT \* FROM sales\_data   
WHERE sale\_date BETWEEN '2023-01-10' AND '2023-01-20'  
ORDER BY sale\_date;  
  
-- Range queries are very efficient with clustered indexes  
SELECT \* FROM employees   
WHERE employee\_id BETWEEN 100 AND 200;

## Non-Clustered Indexes

### Single Column Indexes

-- Index for frequent WHERE clause searches  
CREATE INDEX IX\_employees\_last\_name ON employees (last\_name);  
  
-- Index for JOIN operations  
CREATE INDEX IX\_orders\_employee\_id ON orders (employee\_id);  
  
-- Index for date range queries  
CREATE INDEX IX\_orders\_order\_date ON orders (order\_date);  
  
-- Index for department searches  
CREATE INDEX IX\_employees\_department ON employees (department);

### Performance Comparison

-- Query without index (table scan)  
SELECT \* FROM employees WHERE last\_name = 'Johnson';  
  
-- After creating index IX\_employees\_last\_name, same query uses index seek  
-- Check execution plan to see the difference  
  
-- View index usage statistics  
SELECT   
 i.name AS index\_name,  
 s.user\_seeks,  
 s.user\_scans,  
 s.user\_lookups,  
 s.user\_updates,  
 s.last\_user\_seek,  
 s.last\_user\_scan  
FROM sys.indexes i  
LEFT JOIN sys.dm\_db\_index\_usage\_stats s   
 ON i.object\_id = s.object\_id AND i.index\_id = s.index\_id  
WHERE i.object\_id = OBJECT\_ID('employees');

## Unique Indexes

### Enforcing Data Uniqueness

-- Create unique index on email column  
CREATE UNIQUE INDEX IX\_employees\_email ON employees (email);  
  
-- Try to insert duplicate email (will fail)  
-- INSERT INTO employees VALUES ('Test', 'User', 'john.doe@company.com', ...);  
  
-- Create unique composite index  
CREATE UNIQUE INDEX IX\_employees\_name\_dept ON employees (first\_name, last\_name, department);  
  
-- Unique index with NULL handling  
CREATE UNIQUE INDEX IX\_employees\_phone ON employees (phone)   
WHERE phone IS NOT NULL; -- Filtered unique index

### Unique Index vs Unique Constraint

-- Using UNIQUE constraint (creates unique index automatically)  
ALTER TABLE employees   
ADD CONSTRAINT UQ\_employees\_email UNIQUE (email);  
  
-- View constraint and associated index  
SELECT   
 kc.name AS constraint\_name,  
 i.name AS index\_name,  
 i.type\_desc,  
 c.name AS column\_name  
FROM sys.key\_constraints kc  
JOIN sys.indexes i ON kc.parent\_object\_id = i.object\_id AND kc.unique\_index\_id = i.index\_id  
JOIN sys.index\_columns ic ON i.object\_id = ic.object\_id AND i.index\_id = ic.index\_id  
JOIN sys.columns c ON ic.object\_id = c.object\_id AND ic.column\_id = c.column\_id  
WHERE kc.parent\_object\_id = OBJECT\_ID('employees');

## Composite Indexes

### Multi-Column Index Creation

-- Composite index for complex WHERE clauses  
CREATE INDEX IX\_orders\_customer\_date ON orders (customer\_id, order\_date);  
  
-- Index for GROUP BY operations  
CREATE INDEX IX\_orders\_region\_status ON orders (region, status);  
  
-- Index with different sort orders  
CREATE INDEX IX\_employees\_dept\_salary ON employees (department ASC, salary DESC);

### Column Order Importance

-- Index: (department, salary, hire\_date)  
CREATE INDEX IX\_employees\_dept\_sal\_date ON employees (department, salary, hire\_date);  
  
-- These queries can use the index effectively:  
-- 1. Uses all columns  
SELECT \* FROM employees   
WHERE department = 'IT' AND salary > 70000 AND hire\_date > '2020-01-01';  
  
-- 2. Uses leading columns (department, salary)  
SELECT \* FROM employees   
WHERE department = 'IT' AND salary > 70000;  
  
-- 3. Uses first column only  
SELECT \* FROM employees WHERE department = 'IT';  
  
-- This query CANNOT use the index effectively (salary without department):  
SELECT \* FROM employees WHERE salary > 70000; -- May still use index but less efficiently  
  
-- This query CANNOT use the index effectively (non-leading column only):  
SELECT \* FROM employees WHERE hire\_date > '2020-01-01';

### Composite Index Best Practices

-- Order columns by selectivity (most selective first)  
CREATE INDEX IX\_orders\_optimal ON orders (  
 customer\_id, -- Most selective (specific customer)  
 status, -- Moderately selective (few status values)  
 region -- Least selective (only 3 regions)  
);  
  
-- Include frequently used sort columns  
CREATE INDEX IX\_employees\_search\_sort ON employees (  
 department, -- WHERE clause  
 last\_name, -- WHERE clause  
 salary -- ORDER BY clause  
);

## 

## Covering Indexes

### Index with INCLUDE Columns

-- Covering index to avoid key lookups  
CREATE INDEX IX\_employees\_dept\_covering ON employees (department)  
INCLUDE (first\_name, last\_name, salary, hire\_date);  
  
-- This query is completely covered by the index (no key lookup needed)  
SELECT first\_name, last\_name, salary, hire\_date  
FROM employees  
WHERE department = 'IT';  
  
-- Compare with non-covering index  
CREATE INDEX IX\_employees\_dept\_only ON employees (department);  
  
-- Same query now requires key lookup for additional columns  
SELECT first\_name, last\_name, salary, hire\_date  
FROM employees  
WHERE department = 'Sales';

### Covering Index for JOIN Operations

-- Covering index for order-employee joins  
CREATE INDEX IX\_orders\_emp\_covering ON orders (employee\_id)  
INCLUDE (order\_date, total\_amount, status);  
  
-- Covering index for employee side of join  
CREATE INDEX IX\_employees\_join\_covering ON employees (employee\_id)  
INCLUDE (first\_name, last\_name, department);  
  
-- Query that benefits from covering indexes  
SELECT   
 e.first\_name + ' ' + e.last\_name AS employee\_name,  
 e.department,  
 o.order\_date,  
 o.total\_amount,  
 o.status  
FROM employees e  
JOIN orders o ON e.employee\_id = o.employee\_id  
WHERE e.department = 'Sales'  
AND o.order\_date >= '2023-01-01';

## Filtered Indexes

### Indexes on Subsets of Data

-- Index only on active employees  
CREATE INDEX IX\_employees\_active\_dept ON employees (department)  
WHERE is\_active = 1;  
  
-- Index only on recent orders  
CREATE INDEX IX\_orders\_recent ON orders (order\_date, customer\_id)  
WHERE order\_date >= '2023-01-01';  
  
-- Index only on non-null values  
CREATE INDEX IX\_employees\_manager ON employees (manager\_id)  
WHERE manager\_id IS NOT NULL;  
  
-- Index only on high-value orders  
CREATE INDEX IX\_orders\_high\_value ON orders (order\_date)  
INCLUDE (customer\_id, total\_amount)  
WHERE total\_amount > 1000;

### Filtered Index Benefits

-- Statistics show filtered index efficiency  
SELECT   
 i.name AS index\_name,  
 i.type\_desc,  
 i.has\_filter,  
 i.filter\_definition,  
 p.rows AS index\_rows,  
 p.data\_compression\_desc  
FROM sys.indexes i  
JOIN sys.partitions p ON i.object\_id = p.object\_id AND i.index\_id = p.index\_id  
WHERE i.object\_id = OBJECT\_ID('employees')  
AND i.has\_filter = 1;  
  
-- Query that benefits from filtered index  
SELECT first\_name, last\_name, department  
FROM employees  
WHERE is\_active = 1 AND department = 'IT';

## Columnstore Indexes

### Clustered Columnstore Index

-- Create table for analytics workload  
CREATE TABLE sales\_fact (  
 sale\_id INT,  
 product\_id INT,  
 customer\_id INT,  
 sale\_date DATE,  
 quantity INT,  
 unit\_price DECIMAL(8,2),  
 total\_amount DECIMAL(10,2),  
 region VARCHAR(50)  
);  
  
-- Create clustered columnstore index  
CREATE CLUSTERED COLUMNSTORE INDEX CCI\_sales\_fact ON sales\_fact;  
  
-- Insert sample data  
INSERT INTO sales\_fact VALUES  
(1, 101, 1001, '2023-01-01', 5, 19.99, 99.95, 'North'),  
(2, 102, 1002, '2023-01-02', 3, 29.99, 89.97, 'South'),  
(3, 103, 1003, '2023-01-03', 2, 49.99, 99.98, 'East');

### Non-Clustered Columnstore Index

-- Add columnstore index to existing table  
CREATE NONCLUSTERED COLUMNSTORE INDEX NCCI\_orders\_analytics   
ON orders (customer\_id, order\_date, total\_amount, region);  
  
-- Queries that benefit from columnstore  
-- Aggregation queries  
SELECT   
 region,  
 YEAR(order\_date) AS order\_year,  
 COUNT(\*) AS order\_count,  
 SUM(total\_amount) AS total\_sales,  
 AVG(total\_amount) AS avg\_order\_value  
FROM orders  
GROUP BY region, YEAR(order\_date);  
  
-- Large range scans  
SELECT COUNT(\*)  
FROM orders  
WHERE order\_date BETWEEN '2022-01-01' AND '2023-12-31'  
AND total\_amount > 500;

## Index Management

### Rebuilding Indexes

-- Rebuild single index  
ALTER INDEX IX\_employees\_last\_name ON employees REBUILD;  
  
-- Rebuild all indexes on table  
ALTER INDEX ALL ON employees REBUILD;  
  
-- Rebuild with options  
ALTER INDEX IX\_employees\_last\_name ON employees REBUILD  
WITH (  
 FILLFACTOR = 80,  
 ONLINE = ON,  
 MAXDOP = 4  
);  
  
-- Rebuild columnstore index  
ALTER INDEX CCI\_sales\_fact ON sales\_fact REBUILD;

### Reorganizing Indexes

-- Reorganize single index (online operation)  
ALTER INDEX IX\_employees\_last\_name ON employees REORGANIZE;  
  
-- Reorganize all indexes  
ALTER INDEX ALL ON employees REORGANIZE;  
  
-- Reorganize with LOB compaction  
ALTER INDEX IX\_employees\_last\_name ON employees REORGANIZE  
WITH (LOB\_COMPACTION = ON);

### Index Fragmentation Analysis

-- Check index fragmentation  
SELECT   
 i.name AS index\_name,  
 ips.index\_type\_desc,  
 ips.avg\_fragmentation\_in\_percent,  
 ips.fragment\_count,  
 ips.page\_count,  
 CASE   
 WHEN ips.avg\_fragmentation\_in\_percent < 10 THEN 'No Action Needed'  
 WHEN ips.avg\_fragmentation\_in\_percent < 30 THEN 'Reorganize'  
 ELSE 'Rebuild'  
 END AS recommended\_action  
FROM sys.dm\_db\_index\_physical\_stats(DB\_ID(), OBJECT\_ID('employees'), NULL, NULL, 'DETAILED') ips  
JOIN sys.indexes i ON ips.object\_id = i.object\_id AND ips.index\_id = i.index\_id  
WHERE i.name IS NOT NULL  
ORDER BY ips.avg\_fragmentation\_in\_percent DESC;

### Automated Index Maintenance

-- Stored procedure for index maintenance  
CREATE PROCEDURE sp\_IndexMaintenance  
 @table\_name VARCHAR(128) = NULL,  
 @fragmentation\_threshold\_reorganize FLOAT = 10.0,  
 @fragmentation\_threshold\_rebuild FLOAT = 30.0  
AS  
BEGIN  
 DECLARE @sql NVARCHAR(MAX);  
 DECLARE @index\_name VARCHAR(128), @fragmentation FLOAT, @object\_id INT, @index\_id INT;  
   
 DECLARE index\_cursor CURSOR FOR  
 SELECT   
 i.name,  
 ips.avg\_fragmentation\_in\_percent,  
 ips.object\_id,  
 ips.index\_id  
 FROM sys.dm\_db\_index\_physical\_stats(DB\_ID(), OBJECT\_ID(@table\_name), NULL, NULL, 'DETAILED') ips  
 JOIN sys.indexes i ON ips.object\_id = i.object\_id AND ips.index\_id = i.index\_id  
 WHERE i.name IS NOT NULL  
 AND ips.avg\_fragmentation\_in\_percent > @fragmentation\_threshold\_reorganize;  
   
 OPEN index\_cursor;  
 FETCH NEXT FROM index\_cursor INTO @index\_name, @fragmentation, @object\_id, @index\_id;  
   
 WHILE @@FETCH\_STATUS = 0  
 BEGIN  
 IF @fragmentation >= @fragmentation\_threshold\_rebuild  
 BEGIN  
 SET @sql = 'ALTER INDEX ' + QUOTENAME(@index\_name) + ' ON ' + OBJECT\_NAME(@object\_id) + ' REBUILD';  
 PRINT 'Rebuilding: ' + @sql;  
 END  
 ELSE  
 BEGIN  
 SET @sql = 'ALTER INDEX ' + QUOTENAME(@index\_name) + ' ON ' + OBJECT\_NAME(@object\_id) + ' REORGANIZE';  
 PRINT 'Reorganizing: ' + @sql;  
 END  
   
 EXEC sp\_executesql @sql;  
   
 FETCH NEXT FROM index\_cursor INTO @index\_name, @fragmentation, @object\_id, @index\_id;  
 END  
   
 CLOSE index\_cursor;  
 DEALLOCATE index\_cursor;  
END  
  
-- Execute maintenance  
EXEC sp\_IndexMaintenance @table\_name = 'orders';

## Index Performance Analysis

### Index Usage Statistics

-- View index usage statistics  
SELECT   
 OBJECT\_NAME(i.object\_id) AS table\_name,  
 i.name AS index\_name,  
 i.type\_desc,  
 s.user\_seeks,  
 s.user\_scans,  
 s.user\_lookups,  
 s.user\_updates,  
 s.last\_user\_seek,  
 s.last\_user\_scan,  
 s.last\_user\_lookup,  
 s.last\_user\_update,  
 CASE   
 WHEN s.user\_seeks + s.user\_scans + s.user\_lookups = 0 THEN 'Unused'  
 WHEN s.user\_updates > (s.user\_seeks + s.user\_scans + s.user\_lookups) \* 2 THEN 'High Maintenance'  
 ELSE 'Good Usage'  
 END AS usage\_pattern  
FROM sys.indexes i  
LEFT JOIN sys.dm\_db\_index\_usage\_stats s   
 ON i.object\_id = s.object\_id AND i.index\_id = s.index\_id AND s.database\_id = DB\_ID()  
WHERE i.object\_id = OBJECT\_ID('employees')  
ORDER BY s.user\_seeks + s.user\_scans + s.user\_lookups DESC;

### Missing Index Suggestions

-- Find missing index suggestions  
SELECT   
 d.statement AS table\_name,  
 d.equality\_columns,  
 d.inequality\_columns,  
 d.included\_columns,  
 s.user\_seeks,  
 s.user\_scans,  
 s.last\_user\_seek,  
 s.avg\_total\_user\_cost,  
 s.avg\_user\_impact,  
 'CREATE INDEX IX\_' +   
 REPLACE(REPLACE(REPLACE(d.statement, '[', ''), ']', ''), '.', '\_') +   
 '\_Missing ON ' + d.statement +   
 ' (' + ISNULL(d.equality\_columns, '') +   
 CASE WHEN d.inequality\_columns IS NOT NULL THEN   
 CASE WHEN d.equality\_columns IS NOT NULL THEN ',' ELSE '' END + d.inequality\_columns   
 ELSE '' END + ')' +  
 CASE WHEN d.included\_columns IS NOT NULL THEN ' INCLUDE (' + d.included\_columns + ')' ELSE '' END  
 AS create\_statement  
FROM sys.dm\_db\_missing\_index\_details d  
JOIN sys.dm\_db\_missing\_index\_groups g ON d.index\_handle = g.index\_handle  
JOIN sys.dm\_db\_missing\_index\_group\_stats s ON g.index\_group\_handle = s.group\_handle  
WHERE d.database\_id = DB\_ID()  
ORDER BY s.avg\_total\_user\_cost \* s.avg\_user\_impact \* (s.user\_seeks + s.user\_scans) DESC;

### Index Size and Space Usage

-- Analyze index sizes  
SELECT   
 OBJECT\_NAME(i.object\_id) AS table\_name,  
 i.name AS index\_name,  
 i.type\_desc,  
 i.is\_unique,  
 i.fill\_factor,  
 ps.in\_row\_data\_page\_count,  
 ps.in\_row\_used\_page\_count,  
 ps.row\_count,  
 CAST(ps.in\_row\_used\_page\_count \* 8.0 / 1024 AS DECIMAL(10,2)) AS index\_size\_mb,  
 CAST(ps.in\_row\_data\_page\_count \* 8.0 / 1024 AS DECIMAL(10,2)) AS allocated\_size\_mb  
FROM sys.indexes i  
JOIN sys.dm\_db\_partition\_stats ps ON i.object\_id = ps.object\_id AND i.index\_id = ps.index\_id  
WHERE i.object\_id IN (OBJECT\_ID('employees'), OBJECT\_ID('orders'))  
ORDER BY ps.in\_row\_used\_page\_count DESC;

## Index Optimization Strategies

### Query-Specific Index Design

-- Analyze a specific query's execution plan  
-- Query 1: Employee search with sorting  
SELECT employee\_id, first\_name, last\_name, salary  
FROM employees  
WHERE department = 'IT' AND salary > 60000  
ORDER BY last\_name, first\_name;  
  
-- Optimal index for this query  
CREATE INDEX IX\_employees\_query1\_optimal ON employees (department, salary)  
INCLUDE (first\_name, last\_name);  
  
-- Query 2: Complex join with filtering  
SELECT   
 e.first\_name + ' ' + e.last\_name AS employee\_name,  
 o.order\_date,  
 o.total\_amount  
FROM employees e  
JOIN orders o ON e.employee\_id = o.employee\_id  
WHERE e.department = 'Sales'  
AND o.order\_date >= '2023-01-01'  
AND o.total\_amount > 500  
ORDER BY o.order\_date DESC;  
  
-- Optimal indexes for this query  
CREATE INDEX IX\_employees\_sales\_join ON employees (employee\_id)  
INCLUDE (first\_name, last\_name)  
WHERE department = 'Sales';  
  
CREATE INDEX IX\_orders\_date\_amount ON orders (employee\_id, order\_date, total\_amount)  
WHERE order\_date >= '2023-01-01' AND total\_amount > 500;

### Index Consolidation

-- Instead of multiple single-column indexes:  
-- CREATE INDEX IX\_orders\_customer ON orders (customer\_id);  
-- CREATE INDEX IX\_orders\_date ON orders (order\_date);  
-- CREATE INDEX IX\_orders\_status ON orders (status);  
  
-- Create one composite index that can serve multiple queries:  
CREATE INDEX IX\_orders\_consolidated ON orders (customer\_id, order\_date, status)  
INCLUDE (total\_amount, region);  
  
-- This index can efficiently support:  
-- 1. WHERE customer\_id = X  
-- 2. WHERE customer\_id = X AND order\_date = Y  
-- 3. WHERE customer\_id = X AND order\_date = Y AND status = Z  
-- 4. All with covered columns for SELECT lists

### Partitioned Index Strategy

-- Create partitioned table and indexes  
CREATE PARTITION FUNCTION pf\_order\_date (DATE)  
AS RANGE RIGHT FOR VALUES ('2023-01-01', '2023-04-01', '2023-07-01', '2023-10-01');  
  
CREATE PARTITION SCHEME ps\_order\_date  
AS PARTITION pf\_order\_date ALL TO ([PRIMARY]);  
  
-- Create partitioned table  
CREATE TABLE orders\_partitioned (  
 order\_id INT IDENTITY(1,1),  
 customer\_id INT,  
 employee\_id INT,  
 order\_date DATE,  
 ship\_date DATE,  
 total\_amount DECIMAL(10,2),  
 status VARCHAR(20),  
 region VARCHAR(50)  
) ON ps\_order\_date (order\_date);  
  
-- Create aligned partitioned index  
CREATE INDEX IX\_orders\_part\_customer ON orders\_partitioned (customer\_id, order\_date)  
ON ps\_order\_date (order\_date);

## Managing Indexes in SSMS

### Using SSMS Interface

#### Creating Indexes through GUI:

1. **Right-click table** → “Design”
2. **Right-click table** → “Indexes/Keys”
3. **Expand table** → Right-click “Indexes” → “New Index”

#### Index Properties in SSMS:

-- View index properties programmatically  
SELECT   
 i.name AS index\_name,  
 i.type\_desc,  
 i.is\_unique,  
 i.is\_primary\_key,  
 i.is\_unique\_constraint,  
 i.fill\_factor,  
 i.ignore\_dup\_key,  
 i.allow\_row\_locks,  
 i.allow\_page\_locks,  
 i.has\_filter,  
 i.filter\_definition  
FROM sys.indexes i  
WHERE i.object\_id = OBJECT\_ID('employees')  
AND i.name IS NOT NULL;

### Index Monitoring Queries

-- Complete index information for a table  
SELECT   
 t.name AS table\_name,  
 i.name AS index\_name,  
 i.type\_desc,  
 i.is\_unique,  
 i.is\_primary\_key,  
 STRING\_AGG(c.name, ', ') WITHIN GROUP (ORDER BY ic.key\_ordinal) AS key\_columns,  
 i.has\_filter,  
 i.filter\_definition,  
 ps.row\_count,  
 CAST(ps.in\_row\_used\_page\_count \* 8.0 / 1024 AS DECIMAL(10,2)) AS size\_mb  
FROM sys.tables t  
JOIN sys.indexes i ON t.object\_id = i.object\_id  
JOIN sys.index\_columns ic ON i.object\_id = ic.object\_id AND i.index\_id = ic.index\_id  
JOIN sys.columns c ON ic.object\_id = c.object\_id AND ic.column\_id = c.column\_id  
JOIN sys.dm\_db\_partition\_stats ps ON i.object\_id = ps.object\_id AND i.index\_id = ps.index\_id  
WHERE t.name = 'employees' AND i.name IS NOT NULL  
GROUP BY t.name, i.name, i.type\_desc, i.is\_unique, i.is\_primary\_key,   
 i.has\_filter, i.filter\_definition, ps.row\_count, ps.in\_row\_used\_page\_count  
ORDER BY i.type\_desc, i.name;

## Best Practices

### Index Design Guidelines

#### 1. Primary Key Selection

-- Good: Use identity column for primary key  
CREATE TABLE good\_table (  
 id INT IDENTITY(1,1) PRIMARY KEY, -- Clustered index  
 data VARCHAR(100)  
);  
  
-- Avoid: Wide or frequently changing primary key  
CREATE TABLE avoid\_table (  
 natural\_key VARCHAR(50) PRIMARY KEY, -- May cause fragmentation  
 data VARCHAR(100)  
);

#### 2. Index Column Selection

-- Good: Index on selective columns  
CREATE INDEX IX\_employees\_email ON employees (email); -- High selectivity  
CREATE INDEX IX\_employees\_dept\_salary ON employees (department, salary); -- Moderate selectivity  
  
-- Avoid: Index on non-selective columns  
-- CREATE INDEX IX\_employees\_gender ON employees (gender); -- Low selectivity (M/F)  
-- CREATE INDEX IX\_employees\_active ON employees (is\_active); -- Low selectivity (0/1)  
  
-- Better: Use filtered indexes for low-selectivity columns  
CREATE INDEX IX\_employees\_inactive ON employees (employee\_id)   
WHERE is\_active = 0; -- Only index inactive employees

#### 3. Composite Index Order

-- Order by: Equality first, then Inequality, then ORDER BY columns  
CREATE INDEX IX\_orders\_optimal\_order ON orders (  
 status, -- Equality condition (WHERE status = 'Completed')  
 customer\_id, -- Equality condition (WHERE customer\_id = 123)  
 order\_date, -- Range condition (WHERE order\_date >= '2023-01-01')  
 total\_amount -- ORDER BY clause  
);  
  
-- Include frequently accessed columns  
CREATE INDEX IX\_orders\_covering ON orders (status, customer\_id)  
INCLUDE (order\_date, total\_amount, region);

#### 4. Maintenance Considerations

-- Set appropriate fill factor for frequently updated tables  
CREATE INDEX IX\_orders\_updated\_frequently ON orders (order\_date)  
WITH (FILLFACTOR = 80); -- Leave 20% free space for updates  
  
-- For read-only or rarely updated tables  
CREATE INDEX IX\_archive\_data ON archive\_table (date\_column)  
WITH (FILLFACTOR = 100); -- No free space needed

### Index Naming Conventions

-- Consistent naming convention  
-- IX\_TableName\_ColumnName(s)  
CREATE INDEX IX\_employees\_last\_name ON employees (last\_name);  
CREATE INDEX IX\_employees\_dept\_salary ON employees (department, salary);  
CREATE INDEX IX\_orders\_customer\_date ON orders (customer\_id, order\_date);  
  
-- For unique indexes  
-- UX\_TableName\_ColumnName(s)  
CREATE UNIQUE INDEX UX\_employees\_email ON employees (email);  
  
-- For filtered indexes  
-- IX\_TableName\_ColumnName\_FilterDesc  
CREATE INDEX IX\_employees\_salary\_active ON employees (salary)  
WHERE is\_active = 1;

### Performance Monitoring

-- Regular index health check procedure  
CREATE PROCEDURE sp\_IndexHealthCheck  
AS  
BEGIN  
 -- 1. Index fragmentation  
 SELECT   
 OBJECT\_NAME(ips.object\_id) AS table\_name,  
 i.name AS index\_name,  
 ips.avg\_fragmentation\_in\_percent,  
 ips.page\_count,  
 CASE   
 WHEN ips.avg\_fragmentation\_in\_percent < 10 THEN 'Healthy'  
 WHEN ips.avg\_fragmentation\_in\_percent < 30 THEN 'Needs Reorganization'  
 ELSE 'Needs Rebuild'  
 END AS health\_status  
 FROM sys.dm\_db\_index\_physical\_stats(DB\_ID(), NULL, NULL, NULL, 'SAMPLED') ips  
 JOIN sys.indexes i ON ips.object\_id = i.object\_id AND ips.index\_id = i.index\_id  
 WHERE i.name IS NOT NULL AND ips.page\_count > 100  
 ORDER BY ips.avg\_fragmentation\_in\_percent DESC;  
   
 -- 2. Unused indexes  
 SELECT   
 OBJECT\_NAME(i.object\_id) AS table\_name,  
 i.name AS index\_name,  
 i.type\_desc,  
 COALESCE(s.user\_seeks + s.user\_scans + s.user\_lookups, 0) AS total\_reads,  
 COALESCE(s.user\_updates, 0) AS total\_writes,  
 CASE   
 WHEN s.user\_seeks + s.user\_scans + s.user\_lookups IS NULL THEN 'Never Used'  
 WHEN s.user\_updates > (s.user\_seeks + s.user\_scans + s.user\_lookups) \* 5 THEN 'High Maintenance Cost'  
 ELSE 'Good Usage'  
 END AS usage\_assessment  
 FROM sys.indexes i  
 LEFT JOIN sys.dm\_db\_index\_usage\_stats s   
 ON i.object\_id = s.object\_id AND i.index\_id = s.index\_id AND s.database\_id = DB\_ID()  
 WHERE i.name IS NOT NULL AND i.is\_primary\_key = 0  
 ORDER BY total\_reads;  
END  
  
-- Execute health check  
EXEC sp\_IndexHealthCheck;

## Advanced Index Scenarios

### Index for Data Warehousing

-- Create fact table with appropriate indexes  
CREATE TABLE sales\_fact (  
 fact\_id BIGINT IDENTITY(1,1) PRIMARY KEY,  
 date\_key INT,  
 product\_key INT,  
 customer\_key INT,  
 employee\_key INT,  
 quantity INT,  
 unit\_price DECIMAL(10,2),  
 total\_amount DECIMAL(12,2),  
 cost DECIMAL(12,2),  
 profit DECIMAL(12,2)  
);  
  
-- Columnstore index for analytical queries  
CREATE NONCLUSTERED COLUMNSTORE INDEX NCCI\_sales\_fact\_analytics   
ON sales\_fact (date\_key, product\_key, customer\_key, quantity, total\_amount, profit);  
  
-- Traditional B-tree indexes for operational queries  
CREATE INDEX IX\_sales\_fact\_date ON sales\_fact (date\_key, customer\_key)  
INCLUDE (total\_amount, profit);  
  
CREATE INDEX IX\_sales\_fact\_product ON sales\_fact (product\_key, date\_key)  
INCLUDE (quantity, total\_amount);  
  
-- Partitioned index for time-series data  
CREATE PARTITION FUNCTION pf\_sales\_date (INT)  
AS RANGE RIGHT FOR VALUES (20230101, 20230401, 20230701, 20231001);  
  
CREATE PARTITION SCHEME ps\_sales\_date  
AS PARTITION pf\_sales\_date ALL TO ([PRIMARY]);  
  
-- Create partitioned clustered index  
CREATE CLUSTERED INDEX CIX\_sales\_fact\_partitioned   
ON sales\_fact (date\_key, fact\_id)  
ON ps\_sales\_date (date\_key);

### Index for JSON Data

-- Table with JSON column  
CREATE TABLE user\_profiles (  
 user\_id INT IDENTITY(1,1) PRIMARY KEY,  
 username VARCHAR(50),  
 profile\_data NVARCHAR(MAX) CHECK (ISJSON(profile\_data) = 1),  
 created\_date DATETIME2 DEFAULT GETDATE()  
);  
  
-- Insert sample JSON data  
INSERT INTO user\_profiles VALUES  
('john\_doe', '{"age": 30, "city": "New York", "skills": ["SQL", "C#", "JavaScript"]}', GETDATE()),  
('jane\_smith', '{"age": 28, "city": "Chicago", "skills": ["Python", "R", "Machine Learning"]}', GETDATE());  
  
-- Create computed columns for JSON properties  
ALTER TABLE user\_profiles   
ADD age AS CAST(JSON\_VALUE(profile\_data, '$.age') AS INT);  
  
ALTER TABLE user\_profiles   
ADD city AS JSON\_VALUE(profile\_data, '$.city');  
  
-- Create indexes on computed columns  
CREATE INDEX IX\_user\_profiles\_age ON user\_profiles (age);  
CREATE INDEX IX\_user\_profiles\_city ON user\_profiles (city);  
  
-- Query using JSON indexes  
SELECT username, age, city  
FROM user\_profiles  
WHERE age BETWEEN 25 AND 35  
AND city = 'New York';

### Index for Temporal Tables

-- Create system-versioned temporal table  
CREATE TABLE employee\_history (  
 employee\_id INT,  
 first\_name VARCHAR(50),  
 last\_name VARCHAR(50),  
 salary DECIMAL(10,2),  
 department VARCHAR(50),  
   
 -- System columns for temporal functionality  
 valid\_from DATETIME2 GENERATED ALWAYS AS ROW START NOT NULL,  
 valid\_to DATETIME2 GENERATED ALWAYS AS ROW END NOT NULL,  
 PERIOD FOR SYSTEM\_TIME (valid\_from, valid\_to)  
)  
WITH (SYSTEM\_VERSIONING = ON (HISTORY\_TABLE = dbo.employee\_history\_archive));  
  
-- Create indexes optimized for temporal queries  
CREATE INDEX IX\_employee\_history\_time ON employee\_history (valid\_from, valid\_to, employee\_id);  
CREATE INDEX IX\_employee\_history\_archive\_time ON employee\_history\_archive (valid\_to, valid\_from, employee\_id);  
  
-- Temporal query examples  
-- Point-in-time query  
SELECT \* FROM employee\_history   
FOR SYSTEM\_TIME AS OF '2023-06-01'  
WHERE employee\_id = 1;  
  
-- Historical range query  
SELECT \* FROM employee\_history   
FOR SYSTEM\_TIME BETWEEN '2023-01-01' AND '2023-12-31'  
WHERE department = 'IT';

## Index Troubleshooting

### Common Index Problems and Solutions

#### Problem 1: High Index Fragmentation

-- Identify fragmented indexes  
SELECT   
 OBJECT\_NAME(ips.object\_id) AS table\_name,  
 i.name AS index\_name,  
 ips.avg\_fragmentation\_in\_percent,  
 ips.fragment\_count,  
 ips.page\_count,  
 'ALTER INDEX ' + QUOTENAME(i.name) + ' ON ' + QUOTENAME(OBJECT\_NAME(ips.object\_id)) +   
 CASE   
 WHEN ips.avg\_fragmentation\_in\_percent > 30 THEN ' REBUILD;'  
 WHEN ips.avg\_fragmentation\_in\_percent > 10 THEN ' REORGANIZE;'  
 ELSE ' -- No action needed'  
 END AS recommended\_action  
FROM sys.dm\_db\_index\_physical\_stats(DB\_ID(), NULL, NULL, NULL, 'DETAILED') ips  
JOIN sys.indexes i ON ips.object\_id = i.object\_id AND ips.index\_id = i.index\_id  
WHERE i.name IS NOT NULL   
AND ips.page\_count > 100  
AND ips.avg\_fragmentation\_in\_percent > 10  
ORDER BY ips.avg\_fragmentation\_in\_percent DESC;

#### Problem 2: Index Not Being Used

-- Check if indexes are being used  
WITH IndexUsage AS (  
 SELECT   
 OBJECT\_NAME(i.object\_id) AS table\_name,  
 i.name AS index\_name,  
 i.type\_desc,  
 COALESCE(s.user\_seeks, 0) + COALESCE(s.user\_scans, 0) + COALESCE(s.user\_lookups, 0) AS total\_reads,  
 COALESCE(s.user\_updates, 0) AS total\_writes,  
 s.last\_user\_seek,  
 s.last\_user\_scan,  
 CASE   
 WHEN s.user\_seeks IS NULL AND s.user\_scans IS NULL AND s.user\_lookups IS NULL THEN 'UNUSED'  
 WHEN s.user\_updates > (COALESCE(s.user\_seeks, 0) + COALESCE(s.user\_scans, 0) + COALESCE(s.user\_lookups, 0)) \* 2 THEN 'HIGH\_MAINTENANCE'  
 ELSE 'ACTIVE'  
 END AS usage\_pattern  
 FROM sys.indexes i  
 LEFT JOIN sys.dm\_db\_index\_usage\_stats s   
 ON i.object\_id = s.object\_id AND i.index\_id = s.index\_id AND s.database\_id = DB\_ID()  
 WHERE i.name IS NOT NULL  
 AND i.is\_primary\_key = 0  
 AND i.is\_unique\_constraint = 0  
)  
SELECT \*,  
 CASE usage\_pattern  
 WHEN 'UNUSED' THEN 'Consider dropping: DROP INDEX ' + QUOTENAME(index\_name) + ' ON ' + QUOTENAME(table\_name)  
 WHEN 'HIGH\_MAINTENANCE' THEN 'Review necessity - high update overhead'  
 ELSE 'Index is being used effectively'  
 END AS recommendation  
FROM IndexUsage  
ORDER BY usage\_pattern, total\_reads;

#### Problem 3: Too Many Indexes on a Table

-- Identify tables with excessive indexes  
SELECT   
 OBJECT\_NAME(i.object\_id) AS table\_name,  
 COUNT(\*) AS index\_count,  
 SUM(CASE WHEN i.type\_desc = 'CLUSTERED' THEN 1 ELSE 0 END) AS clustered\_count,  
 SUM(CASE WHEN i.type\_desc = 'NONCLUSTERED' THEN 1 ELSE 0 END) AS nonclustered\_count,  
 SUM(CASE WHEN i.type\_desc LIKE '%COLUMNSTORE%' THEN 1 ELSE 0 END) AS columnstore\_count,  
 CASE   
 WHEN COUNT(\*) > 10 THEN 'Consider index consolidation'  
 WHEN COUNT(\*) > 5 THEN 'Monitor performance'  
 ELSE 'Acceptable'  
 END AS assessment  
FROM sys.indexes i  
WHERE i.name IS NOT NULL  
GROUP BY i.object\_id  
HAVING COUNT(\*) > 5  
ORDER BY COUNT(\*) DESC;

### Index Performance Tuning

-- Create procedure to analyze query performance with indexes  
CREATE PROCEDURE sp\_AnalyzeQueryPerformance  
 @query NVARCHAR(MAX)  
AS  
BEGIN  
 -- Enable statistics  
 SET STATISTICS IO ON;  
 SET STATISTICS TIME ON;  
   
 PRINT 'Executing query with current indexes...';  
 PRINT @query;  
   
 -- Execute the query  
 EXEC sp\_executesql @query;  
   
 -- Get execution plan information  
 SELECT   
 qs.sql\_handle,  
 qs.plan\_handle,  
 qs.total\_logical\_reads,  
 qs.total\_physical\_reads,  
 qs.total\_elapsed\_time / 1000 AS total\_elapsed\_time\_ms,  
 qs.execution\_count,  
 qs.total\_logical\_reads / qs.execution\_count AS avg\_logical\_reads,  
 SUBSTRING(qt.text, (qs.statement\_start\_offset/2)+1,  
 ((CASE qs.statement\_end\_offset  
 WHEN -1 THEN DATALENGTH(qt.text)  
 ELSE qs.statement\_end\_offset  
 END - qs.statement\_start\_offset)/2) + 1) AS statement\_text  
 FROM sys.dm\_exec\_query\_stats qs  
 CROSS APPLY sys.dm\_exec\_sql\_text(qs.sql\_handle) qt  
 WHERE qt.text LIKE '%' + REPLACE(@query, '''', '''''') + '%'  
 ORDER BY qs.total\_logical\_reads DESC;  
   
 SET STATISTICS IO OFF;  
 SET STATISTICS TIME OFF;  
END  
  
-- Example usage  
EXEC sp\_AnalyzeQueryPerformance   
 @query = N'SELECT \* FROM employees WHERE department = ''IT'' AND salary > 70000';

## Index Security and Permissions

### Index-Related Permissions

-- Permissions needed for index operations  
-- CREATE/DROP INDEX requires ALTER permission on table  
GRANT ALTER ON employees TO IndexManager;  
  
-- VIEW DEFINITION required to see index definitions  
GRANT VIEW DEFINITION ON employees TO Developer;  
  
-- Example: Create role for index management  
CREATE ROLE db\_indexmanager;  
  
-- Grant necessary permissions  
GRANT ALTER ON SCHEMA::dbo TO db\_indexmanager;  
GRANT VIEW DEFINITION ON SCHEMA::dbo TO db\_indexmanager;  
  
-- Add user to role  
ALTER ROLE db\_indexmanager ADD MEMBER [domain\indexadmin];

### Monitoring Index Security

-- Check permissions on indexes  
SELECT   
 p.principal\_id,  
 pr.name AS principal\_name,  
 p.permission\_name,  
 p.state\_desc,  
 o.name AS object\_name,  
 i.name AS index\_name  
FROM sys.database\_permissions p  
JOIN sys.objects o ON p.major\_id = o.object\_id  
JOIN sys.database\_principals pr ON p.grantee\_principal\_id = pr.principal\_id  
LEFT JOIN sys.indexes i ON o.object\_id = i.object\_id  
WHERE p.permission\_name IN ('ALTER', 'VIEW DEFINITION')  
AND o.type = 'U';

## Index Maintenance Automation

### Automated Index Maintenance Job

-- Create comprehensive index maintenance procedure  
CREATE PROCEDURE sp\_AutomatedIndexMaintenance  
 @database\_name VARCHAR(128) = NULL,  
 @table\_name VARCHAR(128) = NULL,  
 @fragmentation\_threshold\_reorganize FLOAT = 10.0,  
 @fragmentation\_threshold\_rebuild FLOAT = 30.0,  
 @min\_page\_count INT = 1000,  
 @max\_duration\_minutes INT = 240,  
 @online\_rebuild BIT = 1  
AS  
BEGIN  
 SET NOCOUNT ON;  
   
 IF @database\_name IS NULL SET @database\_name = DB\_NAME();  
   
 DECLARE @start\_time DATETIME = GETDATE();  
 DECLARE @sql NVARCHAR(MAX);  
 DECLARE @msg NVARCHAR(255);  
   
 -- Create temp table for maintenance tasks  
 CREATE TABLE #MaintenanceTasks (  
 id INT IDENTITY(1,1),  
 database\_name VARCHAR(128),  
 schema\_name VARCHAR(128),  
 table\_name VARCHAR(128),  
 index\_name VARCHAR(128),  
 fragmentation\_percent FLOAT,  
 page\_count BIGINT,  
 action\_type VARCHAR(20),  
 sql\_command NVARCHAR(MAX),  
 executed BIT DEFAULT 0,  
 execution\_time DATETIME NULL,  
 error\_message NVARCHAR(MAX) NULL  
 );  
   
 -- Populate maintenance tasks  
 INSERT INTO #MaintenanceTasks (database\_name, schema\_name, table\_name, index\_name,   
 fragmentation\_percent, page\_count, action\_type, sql\_command)  
 SELECT   
 @database\_name,  
 OBJECT\_SCHEMA\_NAME(ips.object\_id),  
 OBJECT\_NAME(ips.object\_id),  
 i.name,  
 ips.avg\_fragmentation\_in\_percent,  
 ips.page\_count,  
 CASE   
 WHEN ips.avg\_fragmentation\_in\_percent >= @fragmentation\_threshold\_rebuild THEN 'REBUILD'  
 WHEN ips.avg\_fragmentation\_in\_percent >= @fragmentation\_threshold\_reorganize THEN 'REORGANIZE'  
 END,  
 CASE   
 WHEN ips.avg\_fragmentation\_in\_percent >= @fragmentation\_threshold\_rebuild THEN  
 'ALTER INDEX ' + QUOTENAME(i.name) + ' ON ' +   
 QUOTENAME(OBJECT\_SCHEMA\_NAME(ips.object\_id)) + '.' + QUOTENAME(OBJECT\_NAME(ips.object\_id)) +   
 ' REBUILD' + CASE WHEN @online\_rebuild = 1 THEN ' WITH (ONLINE = ON)' ELSE '' END  
 WHEN ips.avg\_fragmentation\_in\_percent >= @fragmentation\_threshold\_reorganize THEN  
 'ALTER INDEX ' + QUOTENAME(i.name) + ' ON ' +   
 QUOTENAME(OBJECT\_SCHEMA\_NAME(ips.object\_id)) + '.' + QUOTENAME(OBJECT\_NAME(ips.object\_id)) +   
 ' REORGANIZE'  
 END  
 FROM sys.dm\_db\_index\_physical\_stats(DB\_ID(@database\_name), OBJECT\_ID(@table\_name), NULL, NULL, 'SAMPLED') ips  
 JOIN sys.indexes i ON ips.object\_id = i.object\_id AND ips.index\_id = i.index\_id  
 WHERE i.name IS NOT NULL  
 AND ips.page\_count >= @min\_page\_count  
 AND ips.avg\_fragmentation\_in\_percent >= @fragmentation\_threshold\_reorganize  
 ORDER BY ips.avg\_fragmentation\_in\_percent DESC;  
   
 -- Execute maintenance tasks  
 DECLARE @task\_id INT, @task\_sql NVARCHAR(MAX), @task\_action VARCHAR(20);  
 DECLARE @task\_table VARCHAR(128), @task\_index VARCHAR(128);  
   
 DECLARE maintenance\_cursor CURSOR FOR  
 SELECT id, sql\_command, action\_type, table\_name, index\_name  
 FROM #MaintenanceTasks  
 WHERE executed = 0  
 ORDER BY fragmentation\_percent DESC;  
   
 OPEN maintenance\_cursor;  
 FETCH NEXT FROM maintenance\_cursor INTO @task\_id, @task\_sql, @task\_action, @task\_table, @task\_index;  
   
 WHILE @@FETCH\_STATUS = 0 AND DATEDIFF(MINUTE, @start\_time, GETDATE()) < @max\_duration\_minutes  
 BEGIN  
 BEGIN TRY  
 SET @msg = 'Executing ' + @task\_action + ' on ' + @task\_table + '.' + @task\_index;  
 RAISERROR(@msg, 10, 1) WITH NOWAIT;  
   
 EXEC sp\_executesql @task\_sql;  
   
 UPDATE #MaintenanceTasks   
 SET executed = 1, execution\_time = GETDATE()  
 WHERE id = @task\_id;  
   
 END TRY  
 BEGIN CATCH  
 UPDATE #MaintenanceTasks   
 SET error\_message = ERROR\_MESSAGE()  
 WHERE id = @task\_id;  
   
 SET @msg = 'Error executing ' + @task\_action + ' on ' + @task\_table + '.' + @task\_index + ': ' + ERROR\_MESSAGE();  
 RAISERROR(@msg, 16, 1) WITH NOWAIT;  
 END CATCH  
   
 FETCH NEXT FROM maintenance\_cursor INTO @task\_id, @task\_sql, @task\_action, @task\_table, @task\_index;  
 END  
   
 CLOSE maintenance\_cursor;  
 DEALLOCATE maintenance\_cursor;  
   
 -- Summary report  
 SELECT   
 'Maintenance Summary' AS report\_section,  
 COUNT(\*) AS total\_tasks,  
 SUM(CASE WHEN executed = 1 THEN 1 ELSE 0 END) AS completed\_tasks,  
 SUM(CASE WHEN error\_message IS NOT NULL THEN 1 ELSE 0 END) AS failed\_tasks,  
 DATEDIFF(MINUTE, @start\_time, GETDATE()) AS duration\_minutes  
 FROM #MaintenanceTasks  
   
 UNION ALL  
   
 SELECT   
 action\_type + ' Tasks',  
 COUNT(\*),  
 SUM(CASE WHEN executed = 1 THEN 1 ELSE 0 END),  
 SUM(CASE WHEN error\_message IS NOT NULL THEN 1 ELSE 0 END),  
 AVG(CASE WHEN executed = 1 THEN DATEDIFF(SECOND, @start\_time, execution\_time) ELSE NULL END)  
 FROM #MaintenanceTasks  
 GROUP BY action\_type;  
   
 -- Detailed results  
 SELECT \* FROM #MaintenanceTasks ORDER BY fragmentation\_percent DESC;  
   
 DROP TABLE #MaintenanceTasks;  
END  
  
-- Execute automated maintenance  
EXEC sp\_AutomatedIndexMaintenance   
 @fragmentation\_threshold\_reorganize = 10.0,  
 @fragmentation\_threshold\_rebuild = 30.0,  
 @online\_rebuild = 1;

## Summary

Indexes are crucial for SQL Server performance optimization:

### Key Takeaways:

* **Clustered indexes** determine physical data storage order
* **Non-clustered indexes** provide fast data lookup paths
* **Composite indexes** support complex query patterns
* **Covering indexes** eliminate key lookups
* **Filtered indexes** optimize storage for specific data subsets
* **Column store indexes** excel at analytical workloads

### Best Practices Summary:

1. **Design indexes based on query patterns**, not just table structure
2. **Monitor index usage** and remove unused indexes
3. **Maintain indexes regularly** to prevent fragmentation
4. **Use covering indexes** to avoid key lookups
5. **Consider filtered indexes** for selective data
6. **Balance read performance** against write overhead
7. **Test index changes** in non-production environments first

### Performance Impact:

* **Proper indexing** can improve query performance by 10-1000x
* **Poor indexing** can slow down DML operations significantly
* **Regular maintenance** prevents performance degradation over time

Master these indexing concepts to build high-performance SQL Server applications!

# **Triggers**

# Complete Guide to Triggers in SSMS

## Table of Contents

1. [What is a Trigger?](#what-is-a-trigger)
2. [Types of Triggers](#types-of-triggers)
3. [Trigger Execution Context](#trigger-execution-context)
4. [Basic Trigger Syntax](#basic-trigger-syntax)
5. [DML Triggers (AFTER)](#dml-triggers-after)
6. [DML Triggers (INSTEAD OF)](#dml-triggers-instead-of)
7. [DDL Triggers](#ddl-triggers)
8. [LOGON Triggers](#logon-triggers)
9. [Special Tables (INSERTED and DELETED)](#special-tables-inserted-and-deleted)
10. [Trigger Best Practices](#trigger-best-practices)
11. [Error Handling in Triggers](#error-handling-in-triggers)
12. [Nested and Recursive Triggers](#nested-and-recursive-triggers)
13. [Performance Considerations](#performance-considerations)
14. [Managing Triggers in SSMS](#managing-triggers-in-ssms)
15. [Troubleshooting Triggers](#troubleshooting-triggers)
16. [Advanced Trigger Scenarios](#advanced-trigger-scenarios)

## What is a Trigger?

A **Trigger** is a special type of stored procedure that automatically executes (fires) in response to specific events in a SQL Server database. Triggers cannot be directly invoked by users; they are automatically executed by the database engine when the triggering event occurs.

### Key Characteristics:

* **Automatic Execution**: Fires automatically when specific events occur
* **Event-Driven**: Responds to database events (INSERT, UPDATE, DELETE, DDL operations)
* **Transaction Context**: Executes within the same transaction as the triggering statement
* **No Parameters**: Cannot accept input parameters
* **Multiple Triggers**: Multiple triggers can exist on the same object

### Common Use Cases:

* **Auditing**: Track data changes and user activities
* **Data Validation**: Complex business rule enforcement
* **Logging**: Record database activities
* **Data Synchronization**: Keep related data in sync
* **Security**: Monitor and control database access

## Types of Triggers

### 1. DML (Data Manipulation Language) Triggers

* **AFTER Triggers**: Execute after the triggering event completes
* **INSTEAD OF Triggers**: Execute instead of the triggering event

### 2. DDL (Data Definition Language) Triggers

* **Database-Level**: Respond to database schema changes
* **Server-Level**: Respond to server-level events

### 3. LOGON Triggers

* **Server-Level**: Execute when user sessions are established

## Trigger Execution Context

### Transaction Behavior

-- Triggers execute within the same transaction as the triggering statement  
BEGIN TRANSACTION  
 INSERT INTO employees VALUES (...); -- This fires any INSERT triggers  
 -- If trigger fails, entire transaction rolls back  
COMMIT TRANSACTION

### Execution Order

-- Set trigger execution order (when multiple triggers exist)  
EXEC sp\_settriggerorder   
 @triggername = 'tr\_audit\_employees',  
 @order = 'first',  
 @stmttype = 'INSERT';  
  
-- Orders: 'first', 'last', 'none' (default)

## Basic Trigger Syntax

-- Basic DML Trigger Syntax  
CREATE TRIGGER trigger\_name  
ON table\_name  
AFTER | INSTEAD OF | FOR -- FOR is same as AFTER  
 INSERT | UPDATE | DELETE | INSERT, UPDATE | INSERT, DELETE | UPDATE, DELETE | INSERT, UPDATE, DELETE  
AS  
BEGIN  
 -- Trigger logic here  
 -- Can access INSERTED and DELETED tables  
END  
  
-- Drop Trigger  
DROP TRIGGER trigger\_name;  
  
-- Disable/Enable Trigger  
ALTER TABLE table\_name DISABLE TRIGGER trigger\_name;  
ALTER TABLE table\_name ENABLE TRIGGER trigger\_name;  
  
-- Disable all triggers on a table  
ALTER TABLE table\_name DISABLE TRIGGER ALL;

## Sample Tables Setup

-- Create sample tables for demonstrations  
CREATE TABLE employees (  
 employee\_id INT IDENTITY(1,1) PRIMARY KEY,  
 first\_name VARCHAR(50),  
 last\_name VARCHAR(50),  
 email VARCHAR(100),  
 department VARCHAR(50),  
 salary DECIMAL(10,2),  
 hire\_date DATE,  
 is\_active BIT DEFAULT 1,  
 created\_date DATETIME DEFAULT GETDATE(),  
 modified\_date DATETIME DEFAULT GETDATE()  
);  
  
CREATE TABLE employee\_audit (  
 audit\_id INT IDENTITY(1,1) PRIMARY KEY,  
 employee\_id INT,  
 operation VARCHAR(10), -- INSERT, UPDATE, DELETE  
 field\_name VARCHAR(50),  
 old\_value VARCHAR(MAX),  
 new\_value VARCHAR(MAX),  
 changed\_by VARCHAR(50),  
 changed\_date DATETIME DEFAULT GETDATE()  
);  
  
CREATE TABLE salary\_history (  
 history\_id INT IDENTITY(1,1) PRIMARY KEY,  
 employee\_id INT,  
 old\_salary DECIMAL(10,2),  
 new\_salary DECIMAL(10,2),  
 change\_date DATETIME DEFAULT GETDATE(),  
 change\_reason VARCHAR(100),  
 changed\_by VARCHAR(50)  
);  
  
CREATE TABLE department\_stats (  
 department VARCHAR(50) PRIMARY KEY,  
 employee\_count INT DEFAULT 0,  
 total\_salary DECIMAL(12,2) DEFAULT 0,  
 avg\_salary DECIMAL(10,2) DEFAULT 0,  
 last\_updated DATETIME DEFAULT GETDATE()  
);  
  
-- Insert sample data  
INSERT INTO employees VALUES  
('John', 'Doe', 'john.doe@company.com', 'IT', 75000, '2020-01-15', 1, GETDATE(), GETDATE()),  
('Jane', 'Smith', 'jane.smith@company.com', 'HR', 65000, '2019-03-10', 1, GETDATE(), GETDATE()),  
('Bob', 'Johnson', 'bob.johnson@company.com', 'Sales', 55000, '2021-06-20', 1, GETDATE(), GETDATE());  
  
INSERT INTO department\_stats VALUES  
('IT', 1, 75000, 75000, GETDATE()),  
('HR', 1, 65000, 65000, GETDATE()),  
('Sales', 1, 55000, 55000, GETDATE());

## DML Triggers (AFTER)

### AFTER INSERT Trigger

-- Trigger to audit new employee insertions  
CREATE TRIGGER tr\_employees\_after\_insert  
ON employees  
AFTER INSERT  
AS  
BEGIN  
 SET NOCOUNT ON;  
   
 -- Insert audit record for each new employee  
 INSERT INTO employee\_audit (employee\_id, operation, field\_name, new\_value, changed\_by, changed\_date)  
 SELECT   
 i.employee\_id,  
 'INSERT',  
 'NEW\_EMPLOYEE',  
 'Employee: ' + i.first\_name + ' ' + i.last\_name +   
 ', Email: ' + i.email +   
 ', Department: ' + i.department +   
 ', Salary: ' + CAST(i.salary AS VARCHAR(20)),  
 SYSTEM\_USER,  
 GETDATE()  
 FROM inserted i;  
   
 -- Update department statistics  
 UPDATE ds  
 SET employee\_count = employee\_count + emp\_counts.count\_added,  
 total\_salary = total\_salary + emp\_counts.salary\_added,  
 avg\_salary = CASE   
 WHEN (employee\_count + emp\_counts.count\_added) > 0   
 THEN (total\_salary + emp\_counts.salary\_added) / (employee\_count + emp\_counts.count\_added)  
 ELSE 0   
 END,  
 last\_updated = GETDATE()  
 FROM department\_stats ds  
 INNER JOIN (  
 SELECT   
 department,  
 COUNT(\*) as count\_added,  
 SUM(salary) as salary\_added  
 FROM inserted  
 GROUP BY department  
 ) emp\_counts ON ds.department = emp\_counts.department;  
   
 -- Insert new departments if they don't exist  
 INSERT INTO department\_stats (department, employee\_count, total\_salary, avg\_salary, last\_updated)  
 SELECT   
 i.department,  
 COUNT(\*),  
 SUM(i.salary),  
 AVG(i.salary),  
 GETDATE()  
 FROM inserted i  
 LEFT JOIN department\_stats ds ON i.department = ds.department  
 WHERE ds.department IS NULL  
 GROUP BY i.department;  
END  
  
-- Test the INSERT trigger  
INSERT INTO employees VALUES  
('Alice', 'Brown', 'alice.brown@company.com', 'Finance', 70000, '2023-01-10', 1, GETDATE(), GETDATE());  
  
-- Check audit and stats  
SELECT \* FROM employee\_audit WHERE operation = 'INSERT';  
SELECT \* FROM department\_stats;

### AFTER UPDATE Trigger

-- Comprehensive UPDATE trigger for employees  
CREATE TRIGGER tr\_employees\_after\_update  
ON employees  
AFTER UPDATE  
AS  
BEGIN  
 SET NOCOUNT ON;  
   
 DECLARE @field\_name VARCHAR(50), @old\_value VARCHAR(MAX), @new\_value VARCHAR(MAX);  
 DECLARE @employee\_id INT;  
   
 -- Check each updated employee  
 DECLARE update\_cursor CURSOR FOR  
 SELECT i.employee\_id FROM inserted i;  
   
 OPEN update\_cursor;  
 FETCH NEXT FROM update\_cursor INTO @employee\_id;  
   
 WHILE @@FETCH\_STATUS = 0  
 BEGIN  
 -- Check first\_name changes  
 IF UPDATE(first\_name)  
 BEGIN  
 SELECT @old\_value = d.first\_name, @new\_value = i.first\_name  
 FROM deleted d JOIN inserted i ON d.employee\_id = i.employee\_id  
 WHERE i.employee\_id = @employee\_id;  
   
 IF @old\_value != @new\_value  
 BEGIN  
 INSERT INTO employee\_audit VALUES  
 (@employee\_id, 'UPDATE', 'first\_name', @old\_value, @new\_value, SYSTEM\_USER, GETDATE());  
 END  
 END  
   
 -- Check last\_name changes  
 IF UPDATE(last\_name)  
 BEGIN  
 SELECT @old\_value = d.last\_name, @new\_value = i.last\_name  
 FROM deleted d JOIN inserted i ON d.employee\_id = i.employee\_id  
 WHERE i.employee\_id = @employee\_id;  
   
 IF @old\_value != @new\_value  
 BEGIN  
 INSERT INTO employee\_audit VALUES  
 (@employee\_id, 'UPDATE', 'last\_name', @old\_value, @new\_value, SYSTEM\_USER, GETDATE());  
 END  
 END  
   
 -- Check email changes  
 IF UPDATE(email)  
 BEGIN  
 SELECT @old\_value = d.email, @new\_value = i.email  
 FROM deleted d JOIN inserted i ON d.employee\_id = i.employee\_id  
 WHERE i.employee\_id = @employee\_id;  
   
 IF @old\_value != @new\_value  
 BEGIN  
 INSERT INTO employee\_audit VALUES  
 (@employee\_id, 'UPDATE', 'email', @old\_value, @new\_value, SYSTEM\_USER, GETDATE());  
 END  
 END  
   
 -- Check salary changes (with special handling)  
 IF UPDATE(salary)  
 BEGIN  
 DECLARE @old\_salary DECIMAL(10,2), @new\_salary DECIMAL(10,2);  
   
 SELECT @old\_salary = d.salary, @new\_salary = i.salary  
 FROM deleted d JOIN inserted i ON d.employee\_id = i.employee\_id  
 WHERE i.employee\_id = @employee\_id;  
   
 IF @old\_salary != @new\_salary  
 BEGIN  
 -- Log to audit table  
 INSERT INTO employee\_audit VALUES  
 (@employee\_id, 'UPDATE', 'salary', CAST(@old\_salary AS VARCHAR(20)),   
 CAST(@new\_salary AS VARCHAR(20)), SYSTEM\_USER, GETDATE());  
   
 -- Log to salary history  
 INSERT INTO salary\_history (employee\_id, old\_salary, new\_salary, changed\_by)  
 VALUES (@employee\_id, @old\_salary, @new\_salary, SYSTEM\_USER);  
 END  
 END  
   
 -- Check department changes  
 IF UPDATE(department)  
 BEGIN  
 DECLARE @old\_dept VARCHAR(50), @new\_dept VARCHAR(50);  
   
 SELECT @old\_dept = d.department, @new\_dept = i.department  
 FROM deleted d JOIN inserted i ON d.employee\_id = i.employee\_id  
 WHERE i.employee\_id = @employee\_id;  
   
 IF @old\_dept != @new\_dept  
 BEGIN  
 INSERT INTO employee\_audit VALUES  
 (@employee\_id, 'UPDATE', 'department', @old\_dept, @new\_dept, SYSTEM\_USER, GETDATE());  
   
 -- Update department statistics  
 -- Decrease from old department  
 UPDATE department\_stats  
 SET employee\_count = employee\_count - 1,  
 total\_salary = total\_salary - @old\_salary,  
 avg\_salary = CASE   
 WHEN (employee\_count - 1) > 0   
 THEN (total\_salary - @old\_salary) / (employee\_count - 1)  
 ELSE 0   
 END,  
 last\_updated = GETDATE()  
 WHERE department = @old\_dept;  
   
 -- Increase in new department  
 UPDATE department\_stats  
 SET employee\_count = employee\_count + 1,  
 total\_salary = total\_salary + @new\_salary,  
 avg\_salary = (total\_salary + @new\_salary) / (employee\_count + 1),  
 last\_updated = GETDATE()  
 WHERE department = @new\_dept;  
   
 -- Create new department if it doesn't exist  
 IF NOT EXISTS (SELECT 1 FROM department\_stats WHERE department = @new\_dept)  
 BEGIN  
 INSERT INTO department\_stats VALUES  
 (@new\_dept, 1, @new\_salary, @new\_salary, GETDATE());  
 END  
 END  
 END  
   
 FETCH NEXT FROM update\_cursor INTO @employee\_id;  
 END  
   
 CLOSE update\_cursor;  
 DEALLOCATE update\_cursor;  
   
 -- Update modified\_date for all updated employees  
 UPDATE e  
 SET modified\_date = GETDATE()  
 FROM employees e  
 INNER JOIN inserted i ON e.employee\_id = i.employee\_id;  
END  
  
-- Test the UPDATE trigger  
UPDATE employees SET salary = 80000 WHERE employee\_id = 1;  
UPDATE employees SET department = 'Engineering' WHERE employee\_id = 1;  
  
-- Check results  
SELECT \* FROM employee\_audit WHERE employee\_id = 1;  
SELECT \* FROM salary\_history WHERE employee\_id = 1;  
SELECT \* FROM department\_stats;

### AFTER DELETE Trigger

-- Trigger to handle employee deletions  
CREATE TRIGGER tr\_employees\_after\_delete  
ON employees  
AFTER DELETE  
AS  
BEGIN  
 SET NOCOUNT ON;  
   
 -- Audit the deletion  
 INSERT INTO employee\_audit (employee\_id, operation, field\_name, old\_value, changed\_by, changed\_date)  
 SELECT   
 d.employee\_id,  
 'DELETE',  
 'EMPLOYEE\_DELETED',  
 'Employee: ' + d.first\_name + ' ' + d.last\_name +   
 ', Email: ' + d.email +   
 ', Department: ' + d.department +   
 ', Salary: ' + CAST(d.salary AS VARCHAR(20)),  
 SYSTEM\_USER,  
 GETDATE()  
 FROM deleted d;  
   
 -- Update department statistics  
 UPDATE ds  
 SET employee\_count = employee\_count - emp\_counts.count\_deleted,  
 total\_salary = total\_salary - emp\_counts.salary\_deleted,  
 avg\_salary = CASE   
 WHEN (employee\_count - emp\_counts.count\_deleted) > 0   
 THEN (total\_salary - emp\_counts.salary\_deleted) / (employee\_count - emp\_counts.count\_deleted)  
 ELSE 0   
 END,  
 last\_updated = GETDATE()  
 FROM department\_stats ds  
 INNER JOIN (  
 SELECT   
 department,  
 COUNT(\*) as count\_deleted,  
 SUM(salary) as salary\_deleted  
 FROM deleted  
 GROUP BY department  
 ) emp\_counts ON ds.department = emp\_counts.department;  
   
 -- Archive salary history for deleted employees  
 INSERT INTO salary\_history (employee\_id, old\_salary, new\_salary, change\_date, change\_reason, changed\_by)  
 SELECT   
 employee\_id,  
 salary,  
 NULL,  
 GETDATE(),  
 'EMPLOYEE\_DELETED',  
 SYSTEM\_USER  
 FROM deleted;  
END  
  
-- Test the DELETE trigger (be careful!)  
-- INSERT INTO employees VALUES ('Test', 'Employee', 'test@company.com', 'IT', 50000, '2023-01-01', 1, GETDATE(), GETDATE());  
-- DELETE FROM employees WHERE first\_name = 'Test' AND last\_name = 'Employee';  
  
-- Check audit  
-- SELECT \* FROM employee\_audit WHERE operation = 'DELETE';

## DML Triggers (INSTEAD OF)

INSTEAD OF triggers are primarily used with views, especially views that join multiple tables.

### INSTEAD OF Trigger on View

-- Create a view that joins employees and departments  
CREATE VIEW vw\_employee\_details AS  
SELECT   
 e.employee\_id,  
 e.first\_name,  
 e.last\_name,  
 e.email,  
 e.department,  
 e.salary,  
 ds.employee\_count as dept\_employee\_count,  
 ds.avg\_salary as dept\_avg\_salary  
FROM employees e  
LEFT JOIN department\_stats ds ON e.department = ds.department;  
  
-- Create INSTEAD OF INSERT trigger for the view  
CREATE TRIGGER tr\_vw\_employee\_details\_instead\_of\_insert  
ON vw\_employee\_details  
INSTEAD OF INSERT  
AS  
BEGIN  
 SET NOCOUNT ON;  
   
 DECLARE @department VARCHAR(50);  
   
 -- Insert into employees table  
 INSERT INTO employees (first\_name, last\_name, email, department, salary, hire\_date, is\_active)  
 SELECT   
 first\_name,   
 last\_name,   
 email,   
 department,   
 salary,   
 GETDATE(),   
 1  
 FROM inserted;  
   
 -- The AFTER INSERT trigger on employees will handle department\_stats updates  
END  
  
-- Create INSTEAD OF UPDATE trigger for the view  
CREATE TRIGGER tr\_vw\_employee\_details\_instead\_of\_update  
ON vw\_employee\_details  
INSTEAD OF UPDATE  
AS  
BEGIN  
 SET NOCOUNT ON;  
   
 -- Only update the employees table (ignore department stats columns)  
 UPDATE e  
 SET first\_name = i.first\_name,  
 last\_name = i.last\_name,  
 email = i.email,  
 department = i.department,  
 salary = i.salary  
 FROM employees e  
 INNER JOIN inserted i ON e.employee\_id = i.employee\_id;  
   
 -- The AFTER UPDATE trigger on employees will handle auditing and stats  
END  
  
-- Create INSTEAD OF DELETE trigger for the view  
CREATE TRIGGER tr\_vw\_employee\_details\_instead\_of\_delete  
ON vw\_employee\_details  
INSTEAD OF DELETE  
AS  
BEGIN  
 SET NOCOUNT ON;  
   
 -- Delete from employees table  
 DELETE e  
 FROM employees e  
 INNER JOIN deleted d ON e.employee\_id = d.employee\_id;  
   
 -- The AFTER DELETE trigger on employees will handle cleanup  
END  
  
-- Test INSTEAD OF triggers  
INSERT INTO vw\_employee\_details (first\_name, last\_name, email, department, salary)  
VALUES ('Test', 'User', 'test.user@company.com', 'Marketing', 60000);  
  
UPDATE vw\_employee\_details   
SET salary = 65000   
WHERE first\_name = 'Test' AND last\_name = 'User';  
  
-- Check results  
SELECT \* FROM vw\_employee\_details WHERE first\_name = 'Test';  
SELECT \* FROM employee\_audit WHERE employee\_id = (SELECT employee\_id FROM employees WHERE first\_name = 'Test' AND last\_name = 'User');

## DDL Triggers

DDL triggers fire in response to schema changes in the database or server.

### Database-Level DDL Trigger

-- Create audit table for DDL events  
CREATE TABLE ddl\_audit (  
 audit\_id INT IDENTITY(1,1) PRIMARY KEY,  
 event\_type VARCHAR(50),  
 object\_name VARCHAR(128),  
 object\_type VARCHAR(50),  
 sql\_command NVARCHAR(MAX),  
 login\_name VARCHAR(128),  
 event\_date DATETIME DEFAULT GETDATE()  
);  
  
-- Create DDL trigger to audit schema changes  
CREATE TRIGGER tr\_ddl\_audit  
ON DATABASE  
FOR CREATE\_TABLE, ALTER\_TABLE, DROP\_TABLE,   
 CREATE\_VIEW, ALTER\_VIEW, DROP\_VIEW,  
 CREATE\_PROCEDURE, ALTER\_PROCEDURE, DROP\_PROCEDURE,  
 CREATE\_FUNCTION, ALTER\_FUNCTION, DROP\_FUNCTION,  
 CREATE\_INDEX, DROP\_INDEX  
AS  
BEGIN  
 SET NOCOUNT ON;  
   
 DECLARE @event\_data XML = EVENTDATA();  
   
 INSERT INTO ddl\_audit (event\_type, object\_name, object\_type, sql\_command, login\_name)  
 SELECT   
 @event\_data.value('(/EVENT\_INSTANCE/EventType)[1]', 'VARCHAR(50)'),  
 @event\_data.value('(/EVENT\_INSTANCE/ObjectName)[1]', 'VARCHAR(128)'),  
 @event\_data.value('(/EVENT\_INSTANCE/ObjectType)[1]', 'VARCHAR(50)'),  
 @event\_data.value('(/EVENT\_INSTANCE/TSQLCommand)[1]', 'NVARCHAR(MAX)'),  
 @event\_data.value('(/EVENT\_INSTANCE/LoginName)[1]', 'VARCHAR(128)');  
END  
  
-- Test DDL trigger  
CREATE TABLE test\_table (id INT, name VARCHAR(50));  
ALTER TABLE test\_table ADD description VARCHAR(100);  
DROP TABLE test\_table;  
  
-- Check audit  
SELECT \* FROM ddl\_audit ORDER BY event\_date DESC;

### Server-Level DDL Trigger

-- Create server-level audit table (in a specific database)  
USE master;  
GO  
  
CREATE TABLE server\_ddl\_audit (  
 audit\_id INT IDENTITY(1,1) PRIMARY KEY,  
 database\_name VARCHAR(128),  
 event\_type VARCHAR(50),  
 object\_name VARCHAR(128),  
 login\_name VARCHAR(128),  
 event\_date DATETIME DEFAULT GETDATE(),  
 sql\_command NVARCHAR(MAX)  
);  
  
-- Create server-level DDL trigger  
CREATE TRIGGER tr\_server\_ddl\_audit  
ON ALL SERVER  
FOR CREATE\_DATABASE, ALTER\_DATABASE, DROP\_DATABASE,  
 CREATE\_LOGIN, ALTER\_LOGIN, DROP\_LOGIN  
AS  
BEGIN  
 SET NOCOUNT ON;  
   
 DECLARE @event\_data XML = EVENTDATA();  
   
 INSERT INTO master.dbo.server\_ddl\_audit (database\_name, event\_type, object\_name, login\_name, sql\_command)  
 SELECT   
 @event\_data.value('(/EVENT\_INSTANCE/DatabaseName)[1]', 'VARCHAR(128)'),  
 @event\_data.value('(/EVENT\_INSTANCE/EventType)[1]', 'VARCHAR(50)'),  
 @event\_data.value('(/EVENT\_INSTANCE/ObjectName)[1]', 'VARCHAR(128)'),  
 @event\_data.value('(/EVENT\_INSTANCE/LoginName)[1]', 'VARCHAR(128)'),  
 @event\_data.value('(/EVENT\_INSTANCE/TSQLCommand)[1]', 'NVARCHAR(MAX)');  
END  
  
-- Test server-level trigger (be careful!)  
-- CREATE DATABASE test\_db;  
-- DROP DATABASE test\_db;  
  
-- Check server audit  
-- SELECT \* FROM master.dbo.server\_ddl\_audit ORDER BY event\_date DESC;  
  
-- Drop server-level trigger  
-- DROP TRIGGER tr\_server\_ddl\_audit ON ALL SERVER;

## LOGON Triggers

-- Create logon audit table  
USE master;  
GO  
  
CREATE TABLE logon\_audit (  
 audit\_id INT IDENTITY(1,1) PRIMARY KEY,  
 login\_name VARCHAR(128),  
 client\_host VARCHAR(128),  
 program\_name VARCHAR(128),  
 login\_time DATETIME DEFAULT GETDATE(),  
 is\_pooled BIT  
);  
  
-- Create logon trigger to audit connections  
CREATE TRIGGER tr\_logon\_audit  
ON ALL SERVER  
FOR LOGON  
AS  
BEGIN  
 SET NOCOUNT ON;  
   
 -- Skip system processes and SQL Agent  
 IF ORIGINAL\_LOGIN() IN ('sa', 'NT AUTHORITY\SYSTEM', 'NT SERVICE\SQLSERVERAGENT')  
 RETURN;  
   
 INSERT INTO master.dbo.logon\_audit (login\_name, client\_host, program\_name, is\_pooled)  
 SELECT   
 ORIGINAL\_LOGIN(),  
 HOST\_NAME(),  
 PROGRAM\_NAME(),  
 CASE WHEN PROGRAM\_NAME() LIKE '%pooled%' THEN 1 ELSE 0 END;  
END  
  
-- Create security logon trigger (limit connections)  
CREATE TRIGGER tr\_logon\_security  
ON ALL SERVER  
FOR LOGON  
AS  
BEGIN  
 SET NOCOUNT ON;  
   
 -- Block connections from specific hosts  
 IF HOST\_NAME() IN ('BLOCKED\_HOST1', 'BLOCKED\_HOST2')  
 BEGIN  
 ROLLBACK;  
 RETURN;  
 END  
   
 -- Limit connections per login during business hours  
 IF DATEPART(hour, GETDATE()) BETWEEN 9 AND 17  
 BEGIN  
 DECLARE @connection\_count INT;  
   
 SELECT @connection\_count = COUNT(\*)  
 FROM sys.dm\_exec\_sessions  
 WHERE login\_name = ORIGINAL\_LOGIN()  
 AND is\_user\_process = 1;  
   
 IF @connection\_count > 5 -- Limit to 5 concurrent connections  
 BEGIN  
 ROLLBACK;  
 RETURN;  
 END  
 END  
END  
  
-- Check logon audit  
-- SELECT \* FROM master.dbo.logon\_audit ORDER BY login\_time DESC;  
  
-- Drop logon triggers when not needed  
-- DROP TRIGGER tr\_logon\_audit ON ALL SERVER;  
-- DROP TRIGGER tr\_logon\_security ON ALL SERVER;

## Special Tables (INSERTED and DELETED)

### Understanding INSERTED and DELETED Tables

-- Comprehensive trigger showing INSERTED and DELETED usage  
CREATE TRIGGER tr\_employees\_comprehensive\_audit  
ON employees  
AFTER INSERT, UPDATE, DELETE  
AS  
BEGIN  
 SET NOCOUNT ON;  
   
 DECLARE @operation VARCHAR(10);  
 DECLARE @inserted\_count INT = (SELECT COUNT(\*) FROM inserted);  
 DECLARE @deleted\_count INT = (SELECT COUNT(\*) FROM deleted);  
   
 -- Determine operation type  
 IF @inserted\_count > 0 AND @deleted\_count > 0  
 SET @operation = 'UPDATE';  
 ELSE IF @inserted\_count > 0  
 SET @operation = 'INSERT';  
 ELSE  
 SET @operation = 'DELETE';  
   
 -- Handle INSERT operations  
 IF @operation = 'INSERT'  
 BEGIN  
 INSERT INTO employee\_audit (employee\_id, operation, field\_name, new\_value, changed\_by, changed\_date)  
 SELECT   
 employee\_id,  
 'INSERT',  
 'NEW\_RECORD',  
 'ID: ' + CAST(employee\_id AS VARCHAR(10)) +   
 ', Name: ' + first\_name + ' ' + last\_name +  
 ', Dept: ' + department +  
 ', Salary: $' + CAST(salary AS VARCHAR(20)),  
 SYSTEM\_USER,  
 GETDATE()  
 FROM inserted;  
 END  
   
 -- Handle UPDATE operations  
 IF @operation = 'UPDATE'  
 BEGIN  
 -- Compare each field and log changes  
 INSERT INTO employee\_audit (employee\_id, operation, field\_name, old\_value, new\_value, changed\_by, changed\_date)  
 SELECT   
 i.employee\_id,  
 'UPDATE',  
 'SALARY\_CHANGE',  
 '$' + CAST(d.salary AS VARCHAR(20)),  
 '$' + CAST(i.salary AS VARCHAR(20)),  
 SYSTEM\_USER,  
 GETDATE()  
 FROM inserted i  
 INNER JOIN deleted d ON i.employee\_id = d.employee\_id  
 WHERE i.salary != d.salary;  
   
 INSERT INTO employee\_audit (employee\_id, operation, field\_name, old\_value, new\_value, changed\_by, changed\_date)  
 SELECT   
 i.employee\_id,  
 'UPDATE',  
 'DEPARTMENT\_CHANGE',  
 d.department,  
 i.department,  
 SYSTEM\_USER,  
 GETDATE()  
 FROM inserted i  
 INNER JOIN deleted d ON i.employee\_id = d.employee\_id  
 WHERE i.department != d.department;  
   
 INSERT INTO employee\_audit (employee\_id, operation, field\_name, old\_value, new\_value, changed\_by, changed\_date)  
 SELECT   
 i.employee\_id,  
 'UPDATE',  
 'EMAIL\_CHANGE',  
 d.email,  
 i.email,  
 SYSTEM\_USER,  
 GETDATE()  
 FROM inserted i  
 INNER JOIN deleted d ON i.employee\_id = d.employee\_id  
 WHERE i.email != d.email;  
 END  
   
 -- Handle DELETE operations  
 IF @operation = 'DELETE'  
 BEGIN  
 INSERT INTO employee\_audit (employee\_id, operation, field\_name, old\_value, changed\_by, changed\_date)  
 SELECT   
 employee\_id,  
 'DELETE',  
 'DELETED\_RECORD',  
 'ID: ' + CAST(employee\_id AS VARCHAR(10)) +   
 ', Name: ' + first\_name + ' ' + last\_name +  
 ', Dept: ' + department +  
 ', Salary: $' + CAST(salary AS VARCHAR(20)),  
 SYSTEM\_USER,  
 GETDATE()  
 FROM deleted;  
 END  
   
 -- Log operation summary  
 INSERT INTO employee\_audit (employee\_id, operation, field\_name, new\_value, changed\_by, changed\_date)  
 VALUES (  
 NULL,  
 @operation,  
 'OPERATION\_SUMMARY',  
 @operation + ' operation completed. Rows affected: ' +   
 CASE @operation   
 WHEN 'INSERT' THEN CAST(@inserted\_count AS VARCHAR(10))  
 WHEN 'DELETE' THEN CAST(@deleted\_count AS VARCHAR(10))  
 WHEN 'UPDATE' THEN CAST(@inserted\_count AS VARCHAR(10))  
 END,  
 SYSTEM\_USER,  
 GETDATE()  
 );  
END

## Trigger Best Practices

### 1. Keep Triggers Fast and Simple

-- Good: Simple, fast trigger  
CREATE TRIGGER tr\_employees\_simple\_audit  
ON employees  
AFTER INSERT, UPDATE, DELETE  
AS  
BEGIN  
 SET NOCOUNT ON;  
   
 -- Simple logging without complex logic  
 INSERT INTO employee\_audit (employee\_id, operation, changed\_by, changed\_date)  
 SELECT   
 COALESCE(i.employee\_id, d.employee\_id),  
 CASE   
 WHEN i.employee\_id IS NOT NULL AND d.employee\_id IS NOT NULL THEN 'UPDATE'  
 WHEN i.employee\_id IS NOT NULL THEN 'INSERT'  
 ELSE 'DELETE'  
 END,  
 SYSTEM\_USER,  
 GETDATE()  
 FROM inserted i  
 FULL OUTER JOIN deleted d ON i.employee\_id = d.employee\_id;  
END  
  
-- Avoid: Complex logic in triggers  
-- Don't put heavy processing, external calls, or complex business logic in triggers

### 2. Handle Multiple Rows

-- Good: Handle multiple rows properly  
CREATE TRIGGER tr\_employees\_multi\_row\_safe  
ON employees  
AFTER INSERT  
AS  
BEGIN  
 SET NOCOUNT ON;  
   
 -- This works for both single and multiple row inserts  
 INSERT INTO employee\_audit (employee\_id, operation, changed\_by, changed\_date)  
 SELECT employee\_id, 'INSERT', SYSTEM\_USER, GETDATE()  
 FROM inserted;  
   
 -- Update department statistics for all affected departments  
 UPDATE ds  
 SET employee\_count = employee\_count + dept\_counts.new\_employees,  
 total\_salary = total\_salary + dept\_counts.total\_new\_salary,  
 avg\_salary = (total\_salary + dept\_counts.total\_new\_salary) / (employee\_count + dept\_counts.new\_employees),  
 last\_updated = GETDATE()  
 FROM department\_stats ds  
 INNER JOIN (  
 SELECT department, COUNT(\*) as new\_employees, SUM(salary) as total\_new\_salary  
 FROM inserted  
 GROUP BY department  
 ) dept\_counts ON ds.department = dept\_counts.department;  
END  
  
-- Avoid: Assuming single row operations  
-- CREATE TRIGGER tr\_bad\_single\_row  
-- ON employees  
-- AFTER INSERT  
-- AS  
-- BEGIN  
-- DECLARE @emp\_id INT, @dept VARCHAR(50);  
-- SELECT @emp\_id = employee\_id, @dept = department FROM inserted; -- Wrong! Might be multiple rows  
-- END

### 3. Use SET NOCOUNT ON

-- Always use SET NOCOUNT ON to prevent unnecessary messages  
CREATE TRIGGER tr\_employees\_best\_practice  
ON employees  
AFTER INSERT, UPDATE, DELETE  
AS  
BEGIN  
 SET NOCOUNT ON; -- Prevents "X rows affected" messages  
   
 -- Trigger logic here  
END

### 4. Consider Performance Impact

-- Good: Efficient trigger with proper indexing considerations  
CREATE TRIGGER tr\_employees\_efficient  
ON employees  
AFTER UPDATE  
AS  
BEGIN  
 SET NOCOUNT ON;  
   
 -- Only process if specific columns were updated  
 IF UPDATE(salary) OR UPDATE(department)  
 BEGIN  
 -- Efficient joins using primary keys  
 INSERT INTO employee\_audit (employee\_id, operation, field\_name, old\_value, new\_value, changed\_by, changed\_date)  
 SELECT   
 i.employee\_id,  
 'UPDATE',  
 'SALARY',  
 CAST(d.salary AS VARCHAR(20)),  
 CAST(i.salary AS VARCHAR(20)),  
 SYSTEM\_USER,  
 GETDATE()  
 FROM inserted i  
 INNER JOIN deleted d ON i.employee\_id = d.employee\_id  
 WHERE i.salary != d.salary;  
 END  
END

## Error Handling in Triggers

### Proper Error Handling

CREATE TRIGGER tr\_employees\_error\_handling  
ON employees  
AFTER INSERT, UPDATE  
AS  
BEGIN  
 SET NOCOUNT ON;  
   
 BEGIN TRY  
 -- Validate business rules  
 IF EXISTS (SELECT 1 FROM inserted WHERE salary < 0)  
 BEGIN  
 RAISERROR('Salary cannot be negative', 16, 1);  
 ROLLBACK TRANSACTION;  
 RETURN;  
 END  
   
 IF EXISTS (SELECT 1 FROM inserted WHERE salary > 500000)  
 BEGIN  
 RAISERROR('Salary exceeds maximum allowed amount of $500,000', 16, 1);  
 ROLLBACK TRANSACTION;  
 RETURN;  
 END  
   
 -- Check for duplicate emails  
 IF EXISTS (  
 SELECT email   
 FROM inserted   
 GROUP BY email   
 HAVING COUNT(\*) > 1  
 )  
 BEGIN  
 RAISERROR('Duplicate email addresses are not allowed', 16, 1);  
 ROLLBACK TRANSACTION;  
 RETURN;  
 END  
   
 -- Check against existing emails  
 IF EXISTS (  
 SELECT 1   
 FROM inserted i  
 INNER JOIN employees e ON i.email = e.email AND i.employee\_id != e.employee\_id  
 )  
 BEGIN  
 RAISERROR('Email address already exists for another employee', 16, 1);  
 ROLLBACK TRANSACTION;  
 RETURN;  
 END  
   
 -- If all validations pass, proceed with audit logging  
 INSERT INTO employee\_audit (employee\_id, operation, field\_name, new\_value, changed\_by, changed\_date)  
 SELECT   
 COALESCE(i.employee\_id, d.employee\_id),  
 CASE   
 WHEN i.employee\_id IS NOT NULL AND d.employee\_id IS NOT NULL THEN 'UPDATE'  
 WHEN i.employee\_id IS NOT NULL THEN 'INSERT'  
 ELSE 'DELETE'  
 END,  
 'VALIDATED\_OPERATION',  
 'Operation completed successfully',  
 SYSTEM\_USER,  
 GETDATE()  
 FROM inserted i  
 FULL OUTER JOIN deleted d ON i.employee\_id = d.employee\_id;  
   
 END TRY  
 BEGIN CATCH  
 -- Log the error  
 INSERT INTO employee\_audit (employee\_id, operation, field\_name, old\_value, changed\_by, changed\_date)  
 VALUES (  
 NULL,  
 'ERROR',  
 'TRIGGER\_ERROR',  
 'Error: ' + ERROR\_MESSAGE() + ' (Error Number: ' + CAST(ERROR\_NUMBER() AS VARCHAR) + ')',  
 SYSTEM\_USER,  
 GETDATE()  
 );  
   
 -- Re-raise the error  
 THROW;  
 END CATCH  
END  
  
-- Test error handling  
-- INSERT INTO employees VALUES ('Test', 'User', 'john.doe@company.com', 'IT', -1000, '2023-01-01', 1, GETDATE(), GETDATE());  
-- This should fail due to negative salary

### Complex Validation Trigger

CREATE TRIGGER tr\_employees\_complex\_validation  
ON employees  
AFTER INSERT, UPDATE  
AS  
BEGIN  
 SET NOCOUNT ON;  
   
 DECLARE @error\_messages VARCHAR(MAX) = '';  
 DECLARE @error\_count INT = 0;  
   
 -- Collect all validation errors  
   
 -- Check salary range  
 IF EXISTS (SELECT 1 FROM inserted WHERE salary < 20000 OR salary > 500000)  
 BEGIN  
 SET @error\_messages = @error\_messages + 'Salary must be between $20,000 and $500,000. ';  
 SET @error\_count = @error\_count + 1;  
 END  
   
 -- Check email format  
 IF EXISTS (SELECT 1 FROM inserted WHERE email NOT LIKE '%\_@\_%.\_%')  
 BEGIN  
 SET @error\_messages = @error\_messages + 'Invalid email format. ';  
 SET @error\_count = @error\_count + 1;  
 END  
   
 -- Check department exists  
 IF EXISTS (SELECT 1 FROM inserted i WHERE NOT EXISTS (SELECT 1 FROM department\_stats ds WHERE ds.department = i.department))  
 BEGIN  
 SET @error\_messages = @error\_messages + 'Invalid department specified. ';  
 SET @error\_count = @error\_count + 1;  
 END  
   
 -- Check hire date  
 IF EXISTS (SELECT 1 FROM inserted WHERE hire\_date > GETDATE())  
 BEGIN  
 SET @error\_messages = @error\_messages + 'Hire date cannot be in the future. ';  
 SET @error\_count = @error\_count + 1;  
 END  
   
 -- Check for reasonable name lengths  
 IF EXISTS (SELECT 1 FROM inserted WHERE LEN(first\_name) < 2 OR LEN(last\_name) < 2)  
 BEGIN  
 SET @error\_messages = @error\_messages + 'First name and last name must be at least 2 characters long. ';  
 SET @error\_count = @error\_count + 1;  
 END  
   
 -- If there are validation errors, rollback and raise error  
 IF @error\_count > 0  
 BEGIN  
 ROLLBACK TRANSACTION;  
 RAISERROR(@error\_messages, 16, 1);  
 RETURN;  
 END  
   
 -- If all validations pass, continue with normal processing  
 -- ... audit logging code here ...  
END

## Nested and Recursive Triggers

### Understanding Nested Triggers

-- Check current nested triggers setting  
SELECT name, value\_in\_use   
FROM sys.configurations   
WHERE name = 'nested triggers';  
  
-- Enable/disable nested triggers (server-wide setting)  
-- EXEC sp\_configure 'nested triggers', 1; -- Enable  
-- EXEC sp\_configure 'nested triggers', 0; -- Disable  
-- RECONFIGURE;  
  
-- Example: Trigger that causes another trigger to fire  
CREATE TABLE trigger\_log (  
 log\_id INT IDENTITY(1,1) PRIMARY KEY,  
 table\_name VARCHAR(50),  
 action VARCHAR(20),  
 trigger\_name VARCHAR(50),  
 log\_time DATETIME DEFAULT GETDATE()  
);  
  
CREATE TRIGGER tr\_employees\_nested\_example  
ON employees  
AFTER INSERT  
AS  
BEGIN  
 SET NOCOUNT ON;  
   
 -- This insert will fire triggers on trigger\_log table (if any exist)  
 INSERT INTO trigger\_log (table\_name, action, trigger\_name)  
 VALUES ('employees', 'INSERT', 'tr\_employees\_nested\_example');  
END

### Preventing Infinite Recursion

-- Example: Safe recursive trigger using @@NESTLEVEL  
CREATE TRIGGER tr\_employees\_safe\_recursive  
ON employees  
AFTER UPDATE  
AS  
BEGIN  
 SET NOCOUNT ON;  
   
 -- Prevent infinite recursion by checking nest level  
 IF @@NESTLEVEL > 2  
 RETURN;  
   
 -- Update modified\_date (this would normally cause recursion)  
 UPDATE employees   
 SET modified\_date = GETDATE()  
 WHERE employee\_id IN (SELECT employee\_id FROM inserted);  
   
 -- Log the operation  
 INSERT INTO trigger\_log (table\_name, action, trigger\_name)  
 VALUES ('employees', 'RECURSIVE\_UPDATE', 'tr\_employees\_safe\_recursive');  
END  
  
-- Alternative approach: Use a flag column or context\_info  
CREATE TRIGGER tr\_employees\_context\_safe  
ON employees  
AFTER UPDATE  
AS  
BEGIN  
 SET NOCOUNT ON;  
   
 -- Check if this trigger is already running using context\_info  
 DECLARE @context VARBINARY(128) = CONTEXT\_INFO();  
   
 IF @context = 0x01 -- Flag indicating trigger is running  
 RETURN;  
   
 -- Set context to prevent recursion  
 SET CONTEXT\_INFO 0x01;  
   
 -- Perform updates that might trigger this same trigger  
 UPDATE employees   
 SET modified\_date = GETDATE()  
 WHERE employee\_id IN (SELECT employee\_id FROM inserted);  
   
 -- Clear context  
 SET CONTEXT\_INFO 0x00;  
END

## Performance Considerations

### Measuring Trigger Performance

-- Create procedure to analyze trigger performance  
CREATE PROCEDURE sp\_AnalyzeTriggerPerformance  
AS  
BEGIN  
 -- Query to see trigger execution statistics  
 SELECT   
 o.name AS table\_name,  
 t.name AS trigger\_name,  
 t.type\_desc,  
 s.execution\_count,  
 s.total\_elapsed\_time / 1000.0 AS total\_elapsed\_time\_ms,  
 s.total\_elapsed\_time / s.execution\_count / 1000.0 AS avg\_elapsed\_time\_ms,  
 s.total\_logical\_reads,  
 s.total\_logical\_reads / s.execution\_count AS avg\_logical\_reads,  
 s.last\_execution\_time  
 FROM sys.triggers t  
 INNER JOIN sys.objects o ON t.parent\_id = o.object\_id  
 LEFT JOIN sys.dm\_exec\_procedure\_stats s ON t.object\_id = s.object\_id  
 WHERE t.is\_disabled = 0  
 ORDER BY s.total\_elapsed\_time DESC;  
   
 -- Show triggers that might be causing performance issues  
 SELECT   
 o.name AS table\_name,  
 t.name AS trigger\_name,  
 'High average execution time' AS issue  
 FROM sys.triggers t  
 INNER JOIN sys.objects o ON t.parent\_id = o.object\_id  
 INNER JOIN sys.dm\_exec\_procedure\_stats s ON t.object\_id = s.object\_id  
 WHERE s.total\_elapsed\_time / s.execution\_count > 100000 -- > 100ms average  
   
 UNION ALL  
   
 SELECT   
 o.name AS table\_name,  
 t.name AS trigger\_name,  
 'High logical reads' AS issue  
 FROM sys.triggers t  
 INNER JOIN sys.objects o ON t.parent\_id = o.object\_id  
 INNER JOIN sys.dm\_exec\_procedure\_stats s ON t.object\_id = s.object\_id  
 WHERE s.total\_logical\_reads / s.execution\_count > 1000 -- > 1000 reads average  
   
 ORDER BY table\_name, trigger\_name;  
END  
  
-- Execute performance analysis  
EXEC sp\_AnalyzeTriggerPerformance;

### Optimizing Trigger Performance

-- Example: Optimized trigger using efficient techniques  
CREATE TRIGGER tr\_employees\_optimized  
ON employees  
AFTER UPDATE  
AS  
BEGIN  
 SET NOCOUNT ON;  
   
 -- Only execute if relevant columns were updated  
 IF NOT (UPDATE(salary) OR UPDATE(department) OR UPDATE(email))  
 RETURN;  
   
 -- Use EXISTS instead of COUNT(\*) when possible  
 IF EXISTS (SELECT 1 FROM inserted)  
 BEGIN  
 -- Batch operations instead of row-by-row processing  
   
 -- Handle salary updates efficiently  
 IF UPDATE(salary)  
 BEGIN  
 INSERT INTO salary\_history (employee\_id, old\_salary, new\_salary, changed\_by)  
 SELECT   
 i.employee\_id,  
 d.salary,  
 i.salary,  
 SYSTEM\_USER  
 FROM inserted i  
 INNER JOIN deleted d ON i.employee\_id = d.employee\_id  
 WHERE i.salary != d.salary;  
 END  
   
 -- Handle department changes efficiently  
 IF UPDATE(department)  
 BEGIN  
 -- Update department stats in batch  
 WITH dept\_changes AS (  
 SELECT   
 d.department AS old\_dept,  
 i.department AS new\_dept,  
 d.salary AS salary,  
 COUNT(\*) as change\_count  
 FROM inserted i  
 INNER JOIN deleted d ON i.employee\_id = d.employee\_id  
 WHERE i.department != d.department  
 GROUP BY d.department, i.department, d.salary  
 )  
 UPDATE ds SET   
 employee\_count = employee\_count - dc.change\_count,  
 total\_salary = total\_salary - (dc.salary \* dc.change\_count),  
 last\_updated = GETDATE()  
 FROM department\_stats ds  
 INNER JOIN dept\_changes dc ON ds.department = dc.old\_dept;  
 END  
 END  
END

## Managing Triggers in SSMS

### Using SSMS Interface

-- View all triggers in database  
SELECT   
 OBJECT\_SCHEMA\_NAME(t.parent\_id) AS schema\_name,  
 OBJECT\_NAME(t.parent\_id) AS table\_name,  
 t.name AS trigger\_name,  
 t.type\_desc,  
 t.is\_disabled,  
 t.is\_not\_for\_replication,  
 OBJECT\_DEFINITION(t.object\_id) AS trigger\_definition  
FROM sys.triggers t  
WHERE t.parent\_class = 1 -- Table triggers  
ORDER BY schema\_name, table\_name, trigger\_name;  
  
-- View DDL triggers  
SELECT   
 t.name AS trigger\_name,  
 t.type\_desc,  
 t.is\_disabled,  
 CASE t.parent\_class  
 WHEN 0 THEN 'Database'  
 WHEN 100 THEN 'Server'  
 END AS scope,  
 OBJECT\_DEFINITION(t.object\_id) AS trigger\_definition  
FROM sys.triggers t  
WHERE t.parent\_class IN (0, 100) -- DDL triggers  
ORDER BY t.parent\_class, t.name;  
  
-- Get trigger execution order  
SELECT   
 OBJECT\_NAME(t.parent\_id) AS table\_name,  
 t.name AS trigger\_name,  
 e.type\_desc AS event\_type,  
 e.is\_first,  
 e.is\_last  
FROM sys.triggers t  
INNER JOIN sys.trigger\_events e ON t.object\_id = e.object\_id  
ORDER BY table\_name, event\_type, is\_first DESC, is\_last DESC;

### Trigger Information Queries

-- Comprehensive trigger information  
WITH TriggerInfo AS (  
 SELECT   
 t.object\_id,  
 OBJECT\_SCHEMA\_NAME(t.parent\_id) AS schema\_name,  
 OBJECT\_NAME(t.parent\_id) AS table\_name,  
 t.name AS trigger\_name,  
 t.type\_desc,  
 t.is\_disabled,  
 t.is\_not\_for\_replication,  
 STRING\_AGG(te.type\_desc, ', ') AS events,  
 t.create\_date,  
 t.modify\_date  
 FROM sys.triggers t  
 LEFT JOIN sys.trigger\_events te ON t.object\_id = te.object\_id  
 WHERE t.parent\_class = 1  
 GROUP BY t.object\_id, OBJECT\_SCHEMA\_NAME(t.parent\_id), OBJECT\_NAME(t.parent\_id),  
 t.name, t.type\_desc, t.is\_disabled, t.is\_not\_for\_replication,  
 t.create\_date, t.modify\_date  
)  
SELECT   
 ti.\*,  
 s.execution\_count,  
 s.total\_elapsed\_time / 1000.0 AS total\_elapsed\_time\_ms,  
 CASE   
 WHEN s.execution\_count > 0   
 THEN s.total\_elapsed\_time / s.execution\_count / 1000.0  
 ELSE NULL   
 END AS avg\_elapsed\_time\_ms,  
 s.last\_execution\_time  
FROM TriggerInfo ti  
LEFT JOIN sys.dm\_exec\_procedure\_stats s ON ti.object\_id = s.object\_id  
ORDER BY ti.schema\_name, ti.table\_name, ti.trigger\_name;

## Troubleshooting Triggers

### Common Trigger Problems and Solutions

#### Problem 1: Trigger Not Firing

-- Check if trigger is disabled  
SELECT   
 name AS trigger\_name,  
 is\_disabled,  
 is\_not\_for\_replication  
FROM sys.triggers   
WHERE parent\_id = OBJECT\_ID('employees');  
  
-- Enable trigger if disabled  
ALTER TABLE employees ENABLE TRIGGER tr\_employees\_after\_insert;  
  
-- Check trigger events  
SELECT   
 t.name AS trigger\_name,  
 te.type\_desc AS event\_type  
FROM sys.triggers t  
INNER JOIN sys.trigger\_events te ON t.object\_id = te.object\_id  
WHERE t.parent\_id = OBJECT\_ID('employees');

#### Problem 2: Performance Issues

-- Identify slow triggers  
SELECT   
 OBJECT\_NAME(parent\_id) AS table\_name,  
 name AS trigger\_name,  
 OBJECT\_DEFINITION(object\_id) AS definition  
FROM sys.triggers  
WHERE object\_id IN (  
 SELECT object\_id  
 FROM sys.dm\_exec\_procedure\_stats  
 WHERE total\_elapsed\_time / execution\_count > 50000 -- Slower than 50ms average  
);  
  
-- Check for triggers with many logical reads  
SELECT   
 o.name AS table\_name,  
 t.name AS trigger\_name,  
 s.total\_logical\_reads,  
 s.execution\_count,  
 s.total\_logical\_reads / s.execution\_count AS avg\_logical\_reads  
FROM sys.triggers t  
INNER JOIN sys.objects o ON t.parent\_id = o.object\_id  
INNER JOIN sys.dm\_exec\_procedure\_stats s ON t.object\_id = s.object\_id  
WHERE s.total\_logical\_reads / s.execution\_count > 1000  
ORDER BY avg\_logical\_reads DESC;

#### Problem 3: Trigger Errors

-- Create error logging for triggers  
CREATE TABLE trigger\_errors (  
 error\_id INT IDENTITY(1,1) PRIMARY KEY,  
 trigger\_name VARCHAR(128),  
 error\_number INT,  
 error\_message NVARCHAR(MAX),  
 error\_procedure VARCHAR(128),  
 error\_line INT,  
 error\_time DATETIME DEFAULT GETDATE()  
);  
  
-- Example trigger with comprehensive error logging  
CREATE TRIGGER tr\_employees\_error\_logged  
ON employees  
AFTER INSERT, UPDATE, DELETE  
AS  
BEGIN  
 SET NOCOUNT ON;  
   
 BEGIN TRY  
 -- Trigger logic here  
 INSERT INTO employee\_audit (employee\_id, operation, changed\_by, changed\_date)  
 SELECT   
 COALESCE(i.employee\_id, d.employee\_id),  
 CASE   
 WHEN i.employee\_id IS NOT NULL AND d.employee\_id IS NOT NULL THEN 'UPDATE'  
 WHEN i.employee\_id IS NOT NULL THEN 'INSERT'  
 ELSE 'DELETE'  
 END,  
 SYSTEM\_USER,  
 GETDATE()  
 FROM inserted i  
 FULL OUTER JOIN deleted d ON i.employee\_id = d.employee\_id;  
   
 END TRY  
 BEGIN CATCH  
 -- Log the error  
 INSERT INTO trigger\_errors (trigger\_name, error\_number, error\_message, error\_procedure, error\_line)  
 VALUES (  
 'tr\_employees\_error\_logged',  
 ERROR\_NUMBER(),  
 ERROR\_MESSAGE(),  
 ERROR\_PROCEDURE(),  
 ERROR\_LINE()  
 );  
   
 -- Optionally re-raise the error  
 -- THROW;  
 END CATCH  
END

### Debugging Triggers

-- Create debug table for trigger debugging  
CREATE TABLE trigger\_debug (  
 debug\_id INT IDENTITY(1,1) PRIMARY KEY,  
 trigger\_name VARCHAR(128),  
 debug\_message NVARCHAR(MAX),  
 debug\_time DATETIME DEFAULT GETDATE()  
);  
  
-- Example debug trigger  
CREATE TRIGGER tr\_employees\_debug  
ON employees  
AFTER INSERT, UPDATE, DELETE  
AS  
BEGIN  
 SET NOCOUNT ON;  
   
 DECLARE @inserted\_count INT = (SELECT COUNT(\*) FROM inserted);  
 DECLARE @deleted\_count INT = (SELECT COUNT(\*) FROM deleted);  
   
 -- Log entry into trigger  
 INSERT INTO trigger\_debug VALUES (  
 'tr\_employees\_debug',  
 'Trigger started. Inserted: ' + CAST(@inserted\_count AS VARCHAR(10)) +   
 ', Deleted: ' + CAST(@deleted\_count AS VARCHAR(10)),  
 GETDATE()  
 );  
   
 -- Log the operation type  
 INSERT INTO trigger\_debug VALUES (  
 'tr\_employees\_debug',  
 'Operation: ' + CASE   
 WHEN @inserted\_count > 0 AND @deleted\_count > 0 THEN 'UPDATE'  
 WHEN @inserted\_count > 0 THEN 'INSERT'  
 ELSE 'DELETE'  
 END,  
 GETDATE()  
 );  
   
 -- Your actual trigger logic here...  
   
 -- Log completion  
 INSERT INTO trigger\_debug VALUES (  
 'tr\_employees\_debug',  
 'Trigger completed successfully',  
 GETDATE()  
 );  
END  
  
-- Check debug information  
SELECT \* FROM trigger\_debug ORDER BY debug\_time DESC;

## Advanced Trigger Scenarios

### Data Replication Trigger

-- Create replication audit table  
CREATE TABLE replication\_log (  
 log\_id INT IDENTITY(1,1) PRIMARY KEY,  
 source\_table VARCHAR(128),  
 operation VARCHAR(10),  
 record\_id INT,  
 data\_snapshot NVARCHAR(MAX),  
 replicated\_at DATETIME DEFAULT GETDATE()  
);  
  
-- Trigger for data replication/synchronization  
CREATE TRIGGER tr\_employees\_replication  
ON employees  
AFTER INSERT, UPDATE, DELETE  
AS  
BEGIN  
 SET NOCOUNT ON;  
   
 -- Handle INSERT operations  
 IF EXISTS (SELECT 1 FROM inserted) AND NOT EXISTS (SELECT 1 FROM deleted)  
 BEGIN  
 INSERT INTO replication\_log (source\_table, operation, record\_id, data\_snapshot)  
 SELECT   
 'employees',  
 'INSERT',  
 employee\_id,  
 (SELECT \* FROM inserted i2 WHERE i2.employee\_id = i.employee\_id FOR JSON AUTO)  
 FROM inserted i;  
 END  
   
 -- Handle UPDATE operations   
 IF EXISTS (SELECT 1 FROM inserted) AND EXISTS (SELECT 1 FROM deleted)  
 BEGIN  
 INSERT INTO replication\_log (source\_table, operation, record\_id, data\_snapshot)  
 SELECT   
 'employees',  
 'UPDATE',  
 i.employee\_id,  
 JSON\_OBJECT(  
 'old': (SELECT \* FROM deleted d WHERE d.employee\_id = i.employee\_id FOR JSON AUTO),  
 'new': (SELECT \* FROM inserted i2 WHERE i2.employee\_id = i.employee\_id FOR JSON AUTO)  
 )  
 FROM inserted i;  
 END  
   
 -- Handle DELETE operations  
 IF NOT EXISTS (SELECT 1 FROM inserted) AND EXISTS (SELECT 1 FROM deleted)  
 BEGIN  
 INSERT INTO replication\_log (source\_table, operation, record\_id, data\_snapshot)  
 SELECT   
 'employees',  
 'DELETE',  
 employee\_id,  
 (SELECT \* FROM deleted d2 WHERE d2.employee\_id = d.employee\_id FOR JSON AUTO)  
 FROM deleted d;  
 END  
END

### Security Audit Trigger

-- Create security audit table  
CREATE TABLE security\_audit (  
 audit\_id INT IDENTITY(1,1) PRIMARY KEY,  
 table\_name VARCHAR(128),  
 operation VARCHAR(10),  
 user\_name VARCHAR(128),  
 host\_name VARCHAR(128),  
 program\_name VARCHAR(128),  
 sql\_command NVARCHAR(MAX),  
 affected\_rows INT,  
 audit\_timestamp DATETIME DEFAULT GETDATE()  
);  
  
-- Security audit trigger  
CREATE TRIGGER tr\_employees\_security\_audit  
ON employees  
AFTER INSERT, UPDATE, DELETE  
AS  
BEGIN  
 SET NOCOUNT ON;  
   
 DECLARE @operation VARCHAR(10);  
 DECLARE @affected\_rows INT;  
   
 IF EXISTS (SELECT 1 FROM inserted) AND EXISTS (SELECT 1 FROM deleted)  
 BEGIN  
 SET @operation = 'UPDATE';  
 SET @affected\_rows = (SELECT COUNT(\*) FROM inserted);  
 END  
 ELSE IF EXISTS (SELECT 1 FROM inserted)  
 BEGIN  
 SET @operation = 'INSERT';  
 SET @affected\_rows = (SELECT COUNT(\*) FROM inserted);  
 END  
 ELSE  
 BEGIN  
 SET @operation = 'DELETE';  
 SET @affected\_rows = (SELECT COUNT(\*) FROM deleted);  
 END  
   
 -- Log security information  
 INSERT INTO security\_audit (  
 table\_name, operation, user\_name, host\_name,   
 program\_name, affected\_rows  
 )  
 VALUES (  
 'employees',  
 @operation,  
 SYSTEM\_USER,  
 HOST\_NAME(),  
 PROGRAM\_NAME(),  
 @affected\_rows  
 );  
   
 -- Additional security checks  
 IF @operation = 'DELETE' AND @affected\_rows > 10  
 BEGIN  
 -- Alert for bulk deletions  
 INSERT INTO security\_audit (  
 table\_name, operation, user\_name, host\_name,  
 program\_name, sql\_command, affected\_rows  
 )  
 VALUES (  
 'employees',  
 'ALERT',  
 SYSTEM\_USER,  
 HOST\_NAME(),  
 PROGRAM\_NAME(),  
 'BULK DELETE OPERATION - ' + CAST(@affected\_rows AS VARCHAR(10)) + ' rows deleted',  
 @affected\_rows  
 );  
 END  
END

### Business Rule Enforcement Trigger

-- Complex business rule trigger  
CREATE TRIGGER tr\_employees\_business\_rules  
ON employees  
AFTER INSERT, UPDATE  
AS  
BEGIN  
 SET NOCOUNT ON;  
   
 -- Rule 1: Manager cannot have a salary lower than their direct reports  
 IF EXISTS (  
 SELECT 1   
 FROM inserted i  
 INNER JOIN employees e ON i.employee\_id = e.manager\_id  
 WHERE i.salary < e.salary  
 )  
 BEGIN  
 RAISERROR('Manager salary cannot be lower than direct report salaries', 16, 1);  
 ROLLBACK TRANSACTION;  
 RETURN;  
 END  
   
 -- Rule 2: Department salary budget enforcement  
 IF EXISTS (  
 SELECT 1  
 FROM (  
 SELECT   
 i.department,  
 SUM(CASE WHEN e.employee\_id = i.employee\_id THEN i.salary ELSE e.salary END) as total\_dept\_salary  
 FROM inserted i  
 CROSS JOIN employees e  
 WHERE e.department = i.department  
 GROUP BY i.department  
 ) dept\_totals  
 INNER JOIN department\_stats ds ON dept\_totals.department = ds.department  
 WHERE dept\_totals.total\_dept\_salary > ds.budget  
 )  
 BEGIN  
 RAISERROR('Department salary budget would be exceeded', 16, 1);  
 ROLLBACK TRANSACTION;  
 RETURN;  
 END  
   
 -- Rule 3: Salary increase limits (max 20% increase per update)  
 IF UPDATE(salary) AND EXISTS (  
 SELECT 1  
 FROM inserted i  
 INNER JOIN deleted d ON i.employee\_id = d.employee\_id  
 WHERE i.salary > d.salary \* 1.20  
 )  
 BEGIN  
 RAISERROR('Salary increase cannot exceed 20% in a single update', 16, 1);  
 ROLLBACK TRANSACTION;  
 RETURN;  
 END  
   
 -- Rule 4: Email domain restrictions  
 IF EXISTS (  
 SELECT 1   
 FROM inserted   
 WHERE email NOT LIKE '%@company.com'   
 AND email NOT LIKE '%@partner.com'  
 )  
 BEGIN  
 RAISERROR('Email must be from approved domains (@company.com or @partner.com)', 16, 1);  
 ROLLBACK TRANSACTION;  
 RETURN;  
 END  
END

## Summary

Triggers are powerful database objects that provide automatic execution of code in response to database events:

### Key Takeaways:

* **DML Triggers** respond to data changes (INSERT, UPDATE, DELETE)
* **DDL Triggers** respond to schema changes
* **LOGON Triggers** respond to user login events
* **AFTER Triggers** execute after the triggering event
* **INSTEAD OF Triggers** execute in place of the triggering event

### Best Practices Summary:

1. **Keep triggers simple and fast** - Complex logic belongs in stored procedures
2. **Handle multiple rows properly** - Don’t assume single row operations
3. **Use SET NOCOUNT ON** to prevent unnecessary messages
4. **Implement proper error handling** with TRY-CATCH blocks
5. **Consider performance impact** on DML operations
6. **Test thoroughly** including bulk operations
7. **Document trigger logic** and business rules
8. **Monitor trigger performance** regularly

### Common Use Cases:

* **Audit trails** and change tracking
* **Business rule enforcement** and data validation
* **Data synchronization** between related tables
* **Security monitoring** and access control
* **Automatic calculations** and derived values

### Performance Considerations:

* **Triggers add overhead** to DML operations
* **Batch operations** are more efficient than row-by-row processing
* **Minimize logical reads** and complex joins in triggers
* **Consider alternatives** like computed columns or constraints for simple rules

# **Window Function**

# Complete Guide to Window Functions **in** SSMS

## Table of Contents

1. [What are Window Functions?](#what-are-window-functions)
2. [Window Function Syntax](#window-function-syntax)
3. [OVER Clause Components](#over-clause-components)
4. [Ranking Functions](#ranking-functions)
5. [Aggregate Window Functions](#aggregate-window-functions)
6. [Analytic Functions](#analytic-functions)
7. [PARTITION BY Clause](#partition-by-clause)
8. [ORDER BY in Window Functions](#order-by-in-window-functions)
9. [Window Frame Specification](#window-frame-specification)
10. [Advanced Window Function Scenarios](#advanced-window-function-scenarios)
11. [Performance Considerations](#performance-considerations)
12. [Window Functions vs GROUP BY](#window-functions-vs-group-by)
13. [Common Use Cases](#common-use-cases)
14. [Troubleshooting Window Functions](#troubleshooting-window-functions)
15. [Best Practices](#best-practices)

## What are Window Functions?

**Window Functions** are a powerful feature in SQL Server that allow you to perform calculations across a set of table rows that are related to the current row. Unlike aggregate functions that return a single value for a group of rows, window functions return a value for each row while still having access to data from other rows in the “window.”

### Key Characteristics:

* **Row-by-row processing**: Each row gets its own calculated value
* **Access to related rows**: Can access data from other rows in the window
* **No grouping collapse**: All original rows remain in the result set
* **Flexible partitioning**: Define custom row groupings for calculations
* **Ordering support**: Define row order for calculations

### Benefits:

* **Advanced analytics**: Complex analytical calculations made simple
* **Ranking and numbering**: Easy row ranking and sequential numbering
* **Running totals**: Cumulative calculations without self-joins
* **Comparative analysis**: Compare each row against aggregates
* **Time-series analysis**: Lag, lead, and moving average calculations

## Window Function Syntax

### Basic Syntax Structure

-- General window function syntax  
function\_name() OVER (  
 [PARTITION BY column1, column2, ...]  
 [ORDER BY column1 [ASC|DESC], column2 [ASC|DESC], ...]  
 [ROWS|RANGE window\_frame\_specification]  
)

### Function Categories

1. **Ranking Functions**: ROW\_NUMBER(), RANK(), DENSE\_RANK(), NTILE()
2. **Aggregate Functions**: SUM(), AVG(), COUNT(), MIN(), MAX()
3. **Analytic Functions**: LAG(), LEAD(), FIRST\_VALUE(), LAST\_VALUE()
4. **Statistical Functions**: PERCENT\_RANK(), CUME\_DIST(), PERCENTILE\_CONT(), PERCENTILE\_DISC()

## Sample Data Setup

-- Create comprehensive sample tables for demonstrations  
CREATE TABLE employees (  
 employee\_id INT IDENTITY(1,1) PRIMARY KEY,  
 first\_name VARCHAR(50),  
 last\_name VARCHAR(50),  
 department VARCHAR(50),  
 position VARCHAR(50),  
 salary DECIMAL(10,2),  
 hire\_date DATE,  
 manager\_id INT,  
 performance\_rating DECIMAL(3,2), -- 1.0 to 5.0  
 region VARCHAR(50)  
);  
  
CREATE TABLE sales\_data (  
 sale\_id INT IDENTITY(1,1) PRIMARY KEY,  
 employee\_id INT,  
 sale\_date DATE,  
 product\_category VARCHAR(50),  
 amount DECIMAL(10,2),  
 quantity INT,  
 customer\_id INT,  
 region VARCHAR(50)  
);  
  
CREATE TABLE monthly\_targets (  
 target\_id INT IDENTITY(1,1) PRIMARY KEY,  
 employee\_id INT,  
 month\_year VARCHAR(7), -- Format: 2023-01  
 target\_amount DECIMAL(10,2),  
 actual\_amount DECIMAL(10,2)  
);  
  
-- Insert sample data  
INSERT INTO employees VALUES  
('John', 'Doe', 'Sales', 'Sales Rep', 55000, '2020-01-15', 1, 4.2, 'North'),  
('Jane', 'Smith', 'Sales', 'Senior Sales Rep', 65000, '2019-03-10', 1, 4.5, 'North'),  
('Bob', 'Johnson', 'Sales', 'Sales Rep', 52000, '2021-06-20', 1, 3.8, 'South'),  
('Alice', 'Brown', 'Marketing', 'Marketing Manager', 75000, '2018-09-05', NULL, 4.7, 'North'),  
('Charlie', 'Wilson', 'Sales', 'Sales Manager', 85000, '2017-11-30', NULL, 4.3, 'North'),  
('Diana', 'Ross', 'IT', 'Developer', 70000, '2020-08-12', 6, 4.1, 'East'),  
('Elvis', 'King', 'IT', 'Senior Developer', 80000, '2019-04-18', 6, 4.6, 'East'),  
('Frank', 'Miller', 'IT', 'IT Manager', 90000, '2016-12-01', NULL, 4.4, 'East'),  
('Grace', 'Lee', 'Marketing', 'Marketing Specialist', 58000, '2021-02-14', 4, 4.0, 'West'),  
('Henry', 'Ford', 'Sales', 'Sales Rep', 54000, '2022-01-10', 5, 3.9, 'South');  
  
-- Insert sales data  
DECLARE @i INT = 1;  
DECLARE @start\_date DATE = '2023-01-01';  
WHILE @i <= 100  
BEGIN  
 INSERT INTO sales\_data VALUES  
 ((@i % 10) + 1, -- employee\_id (1-10)  
 DATEADD(day, @i % 365, @start\_date), -- sale\_date  
 CASE (@i % 4)   
 WHEN 0 THEN 'Electronics'  
 WHEN 1 THEN 'Clothing'  
 WHEN 2 THEN 'Home & Garden'  
 ELSE 'Books'  
 END, -- product\_category  
 ROUND(RAND(CHECKSUM(NEWID())) \* 5000 + 500, 2), -- amount (500-5500)  
 ROUND(RAND(CHECKSUM(NEWID())) \* 10 + 1, 0), -- quantity (1-11)  
 (@i % 50) + 1, -- customer\_id  
 CASE (@i % 4)  
 WHEN 0 THEN 'North'  
 WHEN 1 THEN 'South'  
 WHEN 2 THEN 'East'  
 ELSE 'West'  
 END); -- region  
 SET @i = @i + 1;  
END;  
  
-- Insert monthly targets  
INSERT INTO monthly\_targets VALUES  
(1, '2023-01', 50000, 48000),  
(1, '2023-02', 50000, 52000),  
(1, '2023-03', 50000, 47000),  
(2, '2023-01', 60000, 62000),  
(2, '2023-02', 60000, 58000),  
(2, '2023-03', 60000, 65000),  
(3, '2023-01', 45000, 43000),  
(3, '2023-02', 45000, 46000),  
(3, '2023-03', 45000, 44000);

## OVER Clause Components

### Basic OVER Clause

-- Simple window function without partitioning  
SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 salary,  
 -- Total count of all employees  
 COUNT(\*) OVER() AS total\_employees,  
 -- Average salary of all employees  
 AVG(salary) OVER() AS overall\_avg\_salary,  
 -- Each employee's salary as percentage of total  
 salary / SUM(salary) OVER() \* 100 AS salary\_percentage\_of\_total  
FROM employees  
ORDER BY salary DESC;

### OVER with Different Components

-- Demonstrate different OVER clause components  
SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 department,  
 salary,  
   
 -- No OVER clause - aggregate function (would require GROUP BY)  
 -- AVG(salary) AS avg\_salary, -- This would cause error  
   
 -- Empty OVER() - window across entire result set  
 AVG(salary) OVER() AS overall\_avg\_salary,  
   
 -- OVER with PARTITION BY - window across department groups  
 AVG(salary) OVER(PARTITION BY department) AS dept\_avg\_salary,  
   
 -- OVER with ORDER BY - running average  
 AVG(salary) OVER(ORDER BY hire\_date) AS running\_avg\_salary,  
   
 -- OVER with both PARTITION BY and ORDER BY  
 AVG(salary) OVER(PARTITION BY department ORDER BY hire\_date) AS dept\_running\_avg\_salary  
FROM employees  
ORDER BY department, hire\_date;

## Ranking Functions

### ROW\_NUMBER()

-- ROW\_NUMBER(): Assigns unique sequential integers  
SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 department,  
 salary,  
   
 -- Overall ranking by salary (highest to lowest)  
 ROW\_NUMBER() OVER(ORDER BY salary DESC) AS overall\_salary\_rank,  
   
 -- Ranking within each department  
 ROW\_NUMBER() OVER(PARTITION BY department ORDER BY salary DESC) AS dept\_salary\_rank,  
   
 -- Ranking by hire date (earliest to latest)  
 ROW\_NUMBER() OVER(ORDER BY hire\_date) AS hire\_order,  
   
 -- Ranking by performance rating within department  
 ROW\_NUMBER() OVER(PARTITION BY department ORDER BY performance\_rating DESC) AS dept\_performance\_rank  
FROM employees  
ORDER BY department, salary DESC;

### RANK() and DENSE\_RANK()

-- Compare RANK() vs DENSE\_RANK() vs ROW\_NUMBER()  
SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 salary,  
 performance\_rating,  
   
 -- ROW\_NUMBER: Always unique (1,2,3,4,5...)  
 ROW\_NUMBER() OVER(ORDER BY performance\_rating DESC) AS row\_num,  
   
 -- RANK: Ties get same rank, skips next rank (1,2,2,4,5...)   
 RANK() OVER(ORDER BY performance\_rating DESC) AS rank\_num,  
   
 -- DENSE\_RANK: Ties get same rank, no skipping (1,2,2,3,4...)  
 DENSE\_RANK() OVER(ORDER BY performance\_rating DESC) AS dense\_rank\_num,  
   
 -- Ranking within department  
 RANK() OVER(PARTITION BY department ORDER BY performance\_rating DESC) AS dept\_performance\_rank  
FROM employees  
ORDER BY performance\_rating DESC;

### NTILE()

-- NTILE(): Divide rows into specified number of groups  
SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 department,  
 salary,  
   
 -- Divide all employees into 4 salary quartiles  
 NTILE(4) OVER(ORDER BY salary) AS salary\_quartile,  
   
 -- Divide each department into 3 performance tiers  
 NTILE(3) OVER(PARTITION BY department ORDER BY performance\_rating DESC) AS dept\_performance\_tier,  
   
 -- Create salary categories with labels  
 CASE NTILE(4) OVER(ORDER BY salary)  
 WHEN 1 THEN 'Bottom 25%'  
 WHEN 2 THEN 'Lower Middle 25%'  
 WHEN 3 THEN 'Upper Middle 25%'  
 WHEN 4 THEN 'Top 25%'  
 END AS salary\_category,  
   
 -- Percentile ranking  
 PERCENT\_RANK() OVER(ORDER BY salary) AS salary\_percentile\_rank,  
 CUME\_DIST() OVER(ORDER BY salary) AS salary\_cumulative\_dist  
FROM employees  
ORDER BY salary;

## Aggregate Window Functions

### Running Totals and Cumulative Calculations

-- Running totals and cumulative calculations  
SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 department,  
 salary,  
 hire\_date,  
   
 -- Running total of salaries by hire date  
 SUM(salary) OVER(ORDER BY hire\_date) AS running\_salary\_total,  
   
 -- Running total within each department  
 SUM(salary) OVER(PARTITION BY department ORDER BY hire\_date) AS dept\_running\_total,  
   
 -- Running count of employees hired  
 COUNT(\*) OVER(ORDER BY hire\_date) AS employees\_hired\_so\_far,  
   
 -- Running average salary  
 AVG(salary) OVER(ORDER BY hire\_date) AS running\_avg\_salary,  
   
 -- Cumulative percentage of total salary budget  
 SUM(salary) OVER(ORDER BY hire\_date) / SUM(salary) OVER() \* 100 AS cumulative\_salary\_percentage  
FROM employees  
ORDER BY hire\_date;

### Moving Averages and Statistical Functions

-- Moving averages and statistical calculations  
SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 department,  
 salary,  
 hire\_date,  
   
 -- 3-person moving average (current + 2 preceding)  
 AVG(salary) OVER(  
 ORDER BY hire\_date   
 ROWS BETWEEN 2 PRECEDING AND CURRENT ROW  
 ) AS moving\_avg\_3,  
   
 -- 5-person centered moving average (2 before + current + 2 after)  
 AVG(salary) OVER(  
 ORDER BY hire\_date   
 ROWS BETWEEN 2 PRECEDING AND 2 FOLLOWING  
 ) AS centered\_moving\_avg\_5,  
   
 -- Moving sum of last 3 hires  
 SUM(salary) OVER(  
 ORDER BY hire\_date   
 ROWS BETWEEN 2 PRECEDING AND CURRENT ROW  
 ) AS moving\_sum\_3,  
   
 -- Min and Max in moving window  
 MIN(salary) OVER(  
 ORDER BY hire\_date   
 ROWS BETWEEN 2 PRECEDING AND CURRENT ROW  
 ) AS moving\_min\_3,  
   
 MAX(salary) OVER(  
 ORDER BY hire\_date   
 ROWS BETWEEN 2 PRECEDING AND CURRENT ROW  
 ) AS moving\_max\_3  
FROM employees  
ORDER BY hire\_date;

### Comparative Analysis

-- Comparative analysis using window functions  
SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 department,  
 salary,  
 performance\_rating,  
   
 -- Compare individual salary to department average  
 AVG(salary) OVER(PARTITION BY department) AS dept\_avg\_salary,  
 salary - AVG(salary) OVER(PARTITION BY department) AS salary\_vs\_dept\_avg,  
   
 -- Compare to overall average  
 AVG(salary) OVER() AS overall\_avg\_salary,  
 salary - AVG(salary) OVER() AS salary\_vs\_overall\_avg,  
   
 -- Percentage above/below department average  
 (salary - AVG(salary) OVER(PARTITION BY department)) /   
 AVG(salary) OVER(PARTITION BY department) \* 100 AS pct\_vs\_dept\_avg,  
   
 -- Standard deviation analysis  
 STDEV(salary) OVER(PARTITION BY department) AS dept\_salary\_stdev,  
   
 -- Z-score (how many standard deviations from mean)  
 (salary - AVG(salary) OVER(PARTITION BY department)) /   
 NULLIF(STDEV(salary) OVER(PARTITION BY department), 0) AS salary\_z\_score  
FROM employees  
ORDER BY department, salary DESC;

## Analytic Functions

### LAG() and LEAD() Functions

-- LAG and LEAD for accessing previous and next row values  
SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 department,  
 salary,  
 hire\_date,  
   
 -- Previous employee's salary (by hire date)  
 LAG(salary) OVER(ORDER BY hire\_date) AS previous\_hire\_salary,  
   
 -- Next employee's salary  
 LEAD(salary) OVER(ORDER BY hire\_date) AS next\_hire\_salary,  
   
 -- Salary difference from previous hire  
 salary - LAG(salary) OVER(ORDER BY hire\_date) AS salary\_change\_from\_prev,  
   
 -- Previous employee within same department  
 LAG(salary) OVER(PARTITION BY department ORDER BY hire\_date) AS prev\_dept\_hire\_salary,  
   
 -- Get salary from 2 positions back  
 LAG(salary, 2) OVER(ORDER BY hire\_date) AS salary\_2\_hires\_ago,  
   
 -- Use default value when LAG returns NULL  
 LAG(salary, 1, 0) OVER(ORDER BY hire\_date) AS prev\_salary\_with\_default,  
   
 -- Previous and next hire dates  
 LAG(hire\_date) OVER(ORDER BY hire\_date) AS previous\_hire\_date,  
 LEAD(hire\_date) OVER(ORDER BY hire\_date) AS next\_hire\_date,  
   
 -- Days between consecutive hires  
 DATEDIFF(DAY, LAG(hire\_date) OVER(ORDER BY hire\_date), hire\_date) AS days\_since\_prev\_hire  
FROM employees  
ORDER BY hire\_date;

### FIRST\_VALUE() and LAST\_VALUE()

-- FIRST\_VALUE and LAST\_VALUE functions  
SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 department,  
 salary,  
 hire\_date,  
   
 -- First hired employee in company  
 FIRST\_VALUE(first\_name + ' ' + last\_name) OVER(  
 ORDER BY hire\_date   
 ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING  
 ) AS first\_employee\_hired,  
   
 -- Last hired employee in company   
 LAST\_VALUE(first\_name + ' ' + last\_name) OVER(  
 ORDER BY hire\_date   
 ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING  
 ) AS last\_employee\_hired,  
   
 -- Highest paid employee in each department  
 FIRST\_VALUE(first\_name + ' ' + last\_name) OVER(  
 PARTITION BY department   
 ORDER BY salary DESC   
 ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING  
 ) AS highest\_paid\_in\_dept,  
   
 -- Lowest paid employee in each department  
 LAST\_VALUE(first\_name + ' ' + last\_name) OVER(  
 PARTITION BY department   
 ORDER BY salary DESC   
 ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING  
 ) AS lowest\_paid\_in\_dept,  
   
 -- Salary of highest paid person in department  
 FIRST\_VALUE(salary) OVER(  
 PARTITION BY department   
 ORDER BY salary DESC   
 ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING  
 ) AS max\_dept\_salary,  
   
 -- Difference from highest paid in department  
 salary - FIRST\_VALUE(salary) OVER(  
 PARTITION BY department   
 ORDER BY salary DESC   
 ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING  
 ) AS diff\_from\_highest\_paid  
FROM employees  
ORDER BY department, salary DESC;

## PARTITION BY Clause

### Understanding Partitioning

-- Comprehensive examples of PARTITION BY usage  
SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 department,  
 region,  
 salary,  
 performance\_rating,  
   
 -- No partitioning - across entire result set  
 AVG(salary) OVER() AS overall\_avg\_salary,  
 RANK() OVER(ORDER BY salary DESC) AS overall\_salary\_rank,  
   
 -- Partition by department  
 AVG(salary) OVER(PARTITION BY department) AS dept\_avg\_salary,  
 RANK() OVER(PARTITION BY department ORDER BY salary DESC) AS dept\_salary\_rank,  
 COUNT(\*) OVER(PARTITION BY department) AS dept\_employee\_count,  
   
 -- Partition by region  
 AVG(salary) OVER(PARTITION BY region) AS region\_avg\_salary,  
 RANK() OVER(PARTITION BY region ORDER BY performance\_rating DESC) AS region\_performance\_rank,  
   
 -- Partition by multiple columns  
 AVG(salary) OVER(PARTITION BY department, region) AS dept\_region\_avg\_salary,  
 RANK() OVER(PARTITION BY department, region ORDER BY salary DESC) AS dept\_region\_salary\_rank,  
 COUNT(\*) OVER(PARTITION BY department, region) AS dept\_region\_count  
FROM employees  
ORDER BY department, region, salary DESC;

### Dynamic Partitioning Examples

-- Advanced partitioning scenarios  
SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 department,  
 salary,  
 hire\_date,  
 YEAR(hire\_date) AS hire\_year,  
   
 -- Partition by hire year  
 AVG(salary) OVER(PARTITION BY YEAR(hire\_date)) AS avg\_salary\_by\_hire\_year,  
 COUNT(\*) OVER(PARTITION BY YEAR(hire\_date)) AS hires\_in\_year,  
   
 -- Partition by salary range  
 AVG(salary) OVER(  
 PARTITION BY CASE   
 WHEN salary < 60000 THEN 'Low'  
 WHEN salary < 80000 THEN 'Medium'   
 ELSE 'High'   
 END  
 ) AS avg\_salary\_by\_range,  
   
 -- Partition by performance tier  
 COUNT(\*) OVER(  
 PARTITION BY CASE   
 WHEN performance\_rating >= 4.5 THEN 'Excellent'  
 WHEN performance\_rating >= 4.0 THEN 'Good'  
 WHEN performance\_rating >= 3.5 THEN 'Average'  
 ELSE 'Below Average'  
 END  
 ) AS count\_in\_performance\_tier,  
   
 -- Complex partitioning with multiple conditions  
 ROW\_NUMBER() OVER(  
 PARTITION BY department,   
 CASE WHEN salary >= 70000 THEN 'Senior' ELSE 'Junior' END  
 ORDER BY performance\_rating DESC  
 ) AS rank\_in\_dept\_level  
FROM employees  
ORDER BY department, salary DESC;

## ORDER BY in Window Functions

### Different Ordering Scenarios

-- Various ORDER BY scenarios in window functions  
SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 department,  
 salary,  
 hire\_date,  
 performance\_rating,  
   
 -- Order by single column  
 ROW\_NUMBER() OVER(ORDER BY salary DESC) AS salary\_rank,  
   
 -- Order by multiple columns  
 ROW\_NUMBER() OVER(ORDER BY department, salary DESC) AS dept\_then\_salary\_rank,  
   
 -- Order with different sort directions  
 ROW\_NUMBER() OVER(ORDER BY hire\_date ASC, salary DESC) AS hire\_date\_salary\_rank,  
   
 -- Running totals with different orderings  
 SUM(salary) OVER(ORDER BY hire\_date) AS running\_total\_by\_hire\_date,  
 SUM(salary) OVER(ORDER BY salary) AS running\_total\_by\_salary,  
 SUM(salary) OVER(ORDER BY performance\_rating DESC) AS running\_total\_by\_performance,  
   
 -- Moving averages with different orderings  
 AVG(salary) OVER(  
 ORDER BY hire\_date   
 ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING  
 ) AS moving\_avg\_by\_hire\_date,  
   
 AVG(performance\_rating) OVER(  
 ORDER BY salary DESC   
 ROWS BETWEEN 2 PRECEDING AND CURRENT ROW  
 ) AS moving\_avg\_performance\_by\_salary  
FROM employees  
ORDER BY hire\_date;

### Handling NULL Values in ORDER BY

-- Create test data with NULL values  
UPDATE employees SET performance\_rating = NULL WHERE employee\_id IN (2, 5);  
  
-- Demonstrate NULL handling in ORDER BY  
SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 salary,  
 performance\_rating,  
   
 -- Default NULL handling (NULLs typically come last in ASC)  
 ROW\_NUMBER() OVER(ORDER BY performance\_rating ASC) AS rank\_asc\_default,  
 ROW\_NUMBER() OVER(ORDER BY performance\_rating DESC) AS rank\_desc\_default,  
   
 -- Using ISNULL to handle NULLs  
 ROW\_NUMBER() OVER(ORDER BY ISNULL(performance\_rating, 0) DESC) AS rank\_null\_as\_zero,  
   
 -- Using COALESCE to handle NULLs  
 ROW\_NUMBER() OVER(ORDER BY COALESCE(performance\_rating, 2.5) DESC) AS rank\_null\_as\_avg,  
   
 -- Conditional ordering with CASE  
 ROW\_NUMBER() OVER(  
 ORDER BY CASE WHEN performance\_rating IS NULL THEN 1 ELSE 0 END,  
 performance\_rating DESC  
 ) AS rank\_nulls\_first,  
   
 LAG(performance\_rating) OVER(ORDER BY ISNULL(performance\_rating, 0) DESC) AS prev\_rating  
FROM employees  
ORDER BY performance\_rating DESC;  
  
-- Reset the NULL values for other examples  
UPDATE employees SET performance\_rating = 4.2 WHERE employee\_id = 2;  
UPDATE employees SET performance\_rating = 4.3 WHERE employee\_id = 5;

## Window Frame Specification

### Understanding Frame Specifications

-- Comprehensive window frame examples  
SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 salary,  
 hire\_date,  
   
 -- Default frame: RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW  
 SUM(salary) OVER(ORDER BY hire\_date) AS default\_running\_total,  
   
 -- Explicit same as default  
 SUM(salary) OVER(  
 ORDER BY hire\_date   
 RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW  
 ) AS explicit\_default,  
   
 -- ROWS vs RANGE  
 SUM(salary) OVER(  
 ORDER BY hire\_date   
 ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW  
 ) AS rows\_running\_total,  
   
 -- Fixed window size (last 3 rows including current)  
 AVG(salary) OVER(  
 ORDER BY hire\_date   
 ROWS BETWEEN 2 PRECEDING AND CURRENT ROW  
 ) AS moving\_avg\_3\_rows,  
   
 -- Centered window (1 before + current + 1 after)  
 AVG(salary) OVER(  
 ORDER BY hire\_date   
 ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING  
 ) AS centered\_avg\_3\_rows,  
   
 -- Future-looking window (current + next 2)  
 MAX(salary) OVER(  
 ORDER BY hire\_date   
 ROWS BETWEEN CURRENT ROW AND 2 FOLLOWING  
 ) AS max\_next\_3\_hires,  
   
 -- Entire partition window  
 MIN(salary) OVER(  
 ORDER BY hire\_date   
 ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING  
 ) AS min\_salary\_all,  
   
 -- Exclude current row  
 AVG(salary) OVER(  
 ORDER BY hire\_date   
 ROWS BETWEEN UNBOUNDED PRECEDING AND 1 PRECEDING  
 ) AS avg\_excluding\_current  
FROM employees  
ORDER BY hire\_date;

### Advanced Frame Specifications

-- Advanced window frame scenarios  
SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 department,  
 salary,  
 hire\_date,  
   
 -- Different frame types within partitions  
   
 -- Running total within each department  
 SUM(salary) OVER(  
 PARTITION BY department   
 ORDER BY hire\_date   
 ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW  
 ) AS dept\_running\_total,  
   
 -- Last 2 hires in department (including current)  
 AVG(salary) OVER(  
 PARTITION BY department   
 ORDER BY hire\_date   
 ROWS BETWEEN 1 PRECEDING AND CURRENT ROW  
 ) AS dept\_last\_2\_avg,  
   
 -- Next hire in department  
 LEAD(salary, 1) OVER(  
 PARTITION BY department   
 ORDER BY hire\_date  
 ) AS next\_hire\_in\_dept,  
   
 -- Compare with department's first and last hire  
 FIRST\_VALUE(salary) OVER(  
 PARTITION BY department   
 ORDER BY hire\_date   
 ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING  
 ) AS dept\_first\_hire\_salary,  
   
 LAST\_VALUE(salary) OVER(  
 PARTITION BY department   
 ORDER BY hire\_date   
 ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING  
 ) AS dept\_last\_hire\_salary,  
   
 -- Rolling maximum within department  
 MAX(salary) OVER(  
 PARTITION BY department   
 ORDER BY hire\_date   
 ROWS BETWEEN 2 PRECEDING AND CURRENT ROW  
 ) AS dept\_rolling\_max\_salary  
FROM employees  
ORDER BY department, hire\_date;

### Frame Specification with Different Data Types

-- Working with different data types in frames  
SELECT   
 sale\_id,  
 employee\_id,  
 sale\_date,  
 amount,  
   
 -- Date-based ranges (RANGE works with dates)  
 SUM(amount) OVER(  
 ORDER BY sale\_date   
 RANGE BETWEEN INTERVAL '7' DAY PRECEDING AND CURRENT ROW  
 ) AS sales\_last\_week,  
   
 SUM(amount) OVER(  
 ORDER BY sale\_date   
 RANGE BETWEEN INTERVAL '30' DAY PRECEDING AND CURRENT ROW  
 ) AS sales\_last\_month,  
   
 -- Moving average over last 5 sales  
 AVG(amount) OVER(  
 ORDER BY sale\_date   
 ROWS BETWEEN 4 PRECEDING AND CURRENT ROW  
 ) AS moving\_avg\_5\_sales,  
   
 -- Percentage of total sales so far  
 amount / SUM(amount) OVER(  
 ORDER BY sale\_date   
 ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW  
 ) \* 100 AS pct\_of\_running\_total,  
   
 -- Compare to same day last week (if exists)  
 LAG(amount, 7) OVER(ORDER BY sale\_date) AS same\_day\_last\_week  
FROM sales\_data  
WHERE employee\_id = 1  
ORDER BY sale\_date;

## Advanced Window Function Scenarios

### Complex Business Analytics

```sql – Advanced business analytics using window functions WITH daily\_sales AS ( SELECT employee\_id, sale\_date, SUM(amount) AS daily\_total, COUNT(\*) AS daily\_transactions FROM sales\_data GROUP BY employee\_id, sale\_date ) SELECT ds.employee\_id, e.first\_name + ’ ’ + e.last\_name AS employee\_name, ds.sale\_date, ds.daily\_total, ds.daily\_transactions,

-- Sales performance metrics  
AVG(ds.daily\_total) OVER(  
 PARTITION BY ds.employee\_id  
) AS employee\_avg\_daily\_sales,  
  
-- Rank daily performance within employee's history  
RANK() OVER(  
 PARTITION BY ds.employee\_id   
 ORDER BY ds.daily\_total DESC  
) AS employee\_daily\_rank,  
  
-- Moving 7-day average  
AVG(ds.daily\_total) OVER(  
 PARTITION BY ds.employee\_id   
 ORDER BY ds.sale\_date   
 ROWS BETWEEN 6 PRECEDING AND CURRENT ROW  
) AS moving\_7day\_avg,  
  
-- Growth rate (compared to previous day)  
(ds.daily\_total - LAG(ds.daily\_total) OVER(  
 PARTITION BY ds.employee\_id   
 ORDER BY ds.sale\_date  
)) / NULLIF(LAG(ds.daily\_total) OVER(  
 PARTITION BY ds.employee\_id   
 ORDER BY ds.sale\_date  
), 0) \* 100 AS daily\_growth\_rate,  
  
-- Consecutive days above average  
CASE   
 WHEN ds.daily\_total > AVG(ds.daily\_total) OVER(PARTITION BY ds.employee\_id) THEN 1   
 ELSE 0   
END AS above\_avg\_flag,  
  
-- Best and worst performing days  
FIRST\_VALUE(ds.daily\_total) OVER(  
 PARTITION BY ds.employee\_id   
 ORDER BY ds.daily\_total DESC   
 ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING  
) AS best\_day\_sales,  
  
LAST\_VALUE(ds.daily\_total) OVER(  
 PARTITION BY ds.employee\_id   
 ORDER BY ds.daily\_total DESC   
 ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING  
) AS worst\_day\_sales,  
  
-- Percentile ranking of daily performance  
PERCENT\_RANK() OVER(  
 PARTITION BY ds.employee\_id   
 ORDER BY ds.daily\_total  
) AS daily\_percentile\_rank

FROM daily\_sales ds JOIN employees e ON ds.employee\_id = e.employee\_id ORDER BY ds.employee\_id, ds.sale\_date;

### Time Series Analysis  
```sql  
-- Advanced time series analysis with window functions  
WITH monthly\_sales AS (  
 SELECT   
 employee\_id,  
 YEAR(sale\_date) AS year,  
 MONTH(sale\_date) AS month,  
 FORMAT(sale\_date, 'yyyy-MM') AS year\_month,  
 SUM(amount) AS monthly\_sales,  
 COUNT(\*) AS monthly\_transactions  
 FROM sales\_data  
 GROUP BY employee\_id, YEAR(sale\_date), MONTH(sale\_date)  
),  
sales\_with\_analytics AS (  
 SELECT   
 ms.\*,  
 e.first\_name + ' ' + e.last\_name AS employee\_name,  
 e.department,  
   
 -- Year-over-year comparison  
 LAG(ms.monthly\_sales, 12) OVER(  
 PARTITION BY ms.employee\_id   
 ORDER BY ms.year, ms.month  
 ) AS same\_month\_last\_year,  
   
 -- Month-over-month growth  
 LAG(ms.monthly\_sales, 1) OVER(  
 PARTITION BY ms.employee\_id   
 ORDER BY ms.year, ms.month  
 ) AS previous\_month\_sales,  
   
 -- 3-month moving average  
 AVG(ms.monthly\_sales) OVER(  
 PARTITION BY ms.employee\_id   
 ORDER BY ms.year, ms.month   
 ROWS BETWEEN 2 PRECEDING AND CURRENT ROW  
 ) AS moving\_3month\_avg,  
   
 -- Quarterly totals  
 SUM(ms.monthly\_sales) OVER(  
 PARTITION BY ms.employee\_id, ms.year, ((ms.month - 1) / 3)  
 ORDER BY ms.month  
 ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW  
 ) AS quarterly\_running\_total,  
   
 -- Peak and valley detection  
 LAG(ms.monthly\_sales, 1) OVER(  
 PARTITION BY ms.employee\_id   
 ORDER BY ms.year, ms.month  
 ) AS prev\_month,  
 LEAD(ms.monthly\_sales, 1) OVER(  
 PARTITION BY ms.employee\_id   
 ORDER BY ms.year, ms.month  
 ) AS next\_month,  
   
 -- Seasonal analysis (compare to average for same month across years)  
 AVG(ms.monthly\_sales) OVER(  
 PARTITION BY ms.employee\_id, ms.month  
 ) AS avg\_for\_this\_month,  
   
 -- Running total for the year  
 SUM(ms.monthly\_sales) OVER(  
 PARTITION BY ms.employee\_id, ms.year  
 ORDER BY ms.month  
 ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW  
 ) AS ytd\_sales  
 FROM monthly\_sales ms  
 JOIN employees e ON ms.employee\_id = e.employee\_id  
)  
SELECT   
 \*,  
 -- Calculate growth rates  
 CASE   
 WHEN previous\_month\_sales > 0 THEN  
 (monthly\_sales - previous\_month\_sales) / previous\_month\_sales \* 100  
 ELSE NULL  
 END AS mom\_growth\_rate,  
   
 CASE   
 WHEN same\_month\_last\_year > 0 THEN  
 (monthly\_sales - same\_month\_last\_year) / same\_month\_last\_year \* 100  
 ELSE NULL  
 END AS yoy\_growth\_rate,  
   
 -- Peak/Valley detection  
 CASE   
 WHEN monthly\_sales > ISNULL(prev\_month, 0) AND monthly\_sales > ISNULL(next\_month, 0) THEN 'Peak'  
 WHEN monthly\_sales < ISNULL(prev\_month, 999999) AND monthly\_sales < ISNULL(next\_month, 999999) THEN 'Valley'  
 ELSE 'Normal'  
 END AS trend\_indicator,  
   
 -- Seasonal variance  
 (monthly\_sales - avg\_for\_this\_month) / NULLIF(avg\_for\_this\_month, 0) \* 100 AS seasonal\_variance\_pct  
FROM sales\_with\_analytics  
ORDER BY employee\_id, year, month;

### Cohort Analysis

-- Cohort analysis using window functions  
WITH customer\_first\_purchase AS (  
 SELECT   
 customer\_id,  
 MIN(sale\_date) AS first\_purchase\_date,  
 FORMAT(MIN(sale\_date), 'yyyy-MM') AS cohort\_month  
 FROM sales\_data  
 GROUP BY customer\_id  
),  
customer\_purchases AS (  
 SELECT   
 sd.customer\_id,  
 sd.sale\_date,  
 sd.amount,  
 cfp.cohort\_month,  
 cfp.first\_purchase\_date,  
 DATEDIFF(MONTH, cfp.first\_purchase\_date, sd.sale\_date) AS months\_since\_first\_purchase  
 FROM sales\_data sd  
 JOIN customer\_first\_purchase cfp ON sd.customer\_id = cfp.customer\_id  
),  
cohort\_data AS (  
 SELECT   
 cohort\_month,  
 months\_since\_first\_purchase,  
 COUNT(DISTINCT customer\_id) AS active\_customers,  
 SUM(amount) AS revenue,  
 AVG(amount) AS avg\_order\_value  
 FROM customer\_purchases  
 GROUP BY cohort\_month, months\_since\_first\_purchase  
)  
SELECT   
 cohort\_month,  
 months\_since\_first\_purchase,  
 active\_customers,  
 revenue,  
 avg\_order\_value,  
   
 -- Retention rate calculation  
 active\_customers / FIRST\_VALUE(active\_customers) OVER(  
 PARTITION BY cohort\_month   
 ORDER BY months\_since\_first\_purchase  
 ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING  
 ) \* 100 AS retention\_rate,  
   
 -- Cumulative revenue per cohort  
 SUM(revenue) OVER(  
 PARTITION BY cohort\_month   
 ORDER BY months\_since\_first\_purchase  
 ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW  
 ) AS cumulative\_revenue,  
   
 -- Revenue per customer in cohort  
 SUM(revenue) OVER(  
 PARTITION BY cohort\_month   
 ORDER BY months\_since\_first\_purchase  
 ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW  
 ) / FIRST\_VALUE(active\_customers) OVER(  
 PARTITION BY cohort\_month   
 ORDER BY months\_since\_first\_purchase  
 ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING  
 ) AS revenue\_per\_original\_customer  
FROM cohort\_data  
WHERE months\_since\_first\_purchase <= 12 -- First year only  
ORDER BY cohort\_month, months\_since\_first\_purchase;

## Performance Considerations

### Optimizing Window Function Performance

-- Performance analysis of window functions  
-- Check execution plans for these queries  
  
-- 1. Efficient window function usage  
SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 salary,  
 -- Reuse window specification  
 ROW\_NUMBER() OVER(win) AS row\_num,  
 RANK() OVER(win) AS rank\_num,  
 DENSE\_RANK() OVER(win) AS dense\_rank\_num  
FROM employees  
WINDOW win AS (ORDER BY salary DESC) -- Note: SQL Server doesn't support WINDOW clause yet  
ORDER BY salary DESC;  
  
-- Alternative efficient approach (rewrite above for SQL Server)  
WITH ranked\_employees AS (  
 SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 salary,  
 ROW\_NUMBER() OVER(ORDER BY salary DESC) AS row\_num,  
 RANK() OVER(ORDER BY salary DESC) AS rank\_num,  
 DENSE\_RANK() OVER(ORDER BY salary DESC) AS dense\_rank\_num  
 FROM employees  
)  
SELECT \* FROM ranked\_employees  
ORDER BY salary DESC;  
  
-- 2. Avoid multiple scans with CTE  
WITH employee\_analytics AS (  
 SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 department,  
 salary,  
 AVG(salary) OVER(PARTITION BY department) AS dept\_avg\_salary,  
 COUNT(\*) OVER(PARTITION BY department) AS dept\_count,  
 SUM(salary) OVER(PARTITION BY department) AS dept\_total\_salary  
 FROM employees  
)  
SELECT   
 \*,  
 salary - dept\_avg\_salary AS salary\_vs\_dept\_avg,  
 salary / dept\_total\_salary \* 100 AS pct\_of\_dept\_budget  
FROM employee\_analytics;

### Index Considerations for Window Functions

-- Create indexes to optimize window function performance  
  
-- Index for ORDER BY in window functions  
CREATE INDEX IX\_employees\_salary\_desc ON employees (salary DESC);  
CREATE INDEX IX\_employees\_hire\_date ON employees (hire\_date);  
  
-- Covering index for partitioned window functions  
CREATE INDEX IX\_employees\_dept\_salary\_covering   
ON employees (department, salary DESC)   
INCLUDE (employee\_id, first\_name, last\_name, performance\_rating);  
  
-- Index for sales data analytics  
CREATE INDEX IX\_sales\_emp\_date\_covering   
ON sales\_data (employee\_id, sale\_date)   
INCLUDE (amount, quantity, product\_category);  
  
-- Test performance with indexes  
SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 department,  
 salary,  
 ROW\_NUMBER() OVER(PARTITION BY department ORDER BY salary DESC) AS dept\_salary\_rank,  
 AVG(salary) OVER(PARTITION BY department) AS dept\_avg\_salary  
FROM employees  
ORDER BY department, salary DESC;  
  
-- Performance monitoring query  
SELECT   
 creation\_time,  
 last\_execution\_time,  
 execution\_count,  
 total\_elapsed\_time / 1000.0 AS total\_elapsed\_time\_ms,  
 total\_logical\_reads,  
 SUBSTRING(qt.text, (qs.statement\_start\_offset/2)+1,  
 ((CASE qs.statement\_end\_offset  
 WHEN -1 THEN DATALENGTH(qt.text)  
 ELSE qs.statement\_end\_offset  
 END - qs.statement\_start\_offset)/2) + 1) AS statement\_text  
FROM sys.dm\_exec\_query\_stats qs  
CROSS APPLY sys.dm\_exec\_sql\_text(qs.sql\_handle) qt  
WHERE qt.text LIKE '%OVER%' -- Find queries with window functions  
ORDER BY total\_elapsed\_time DESC;

## Window Functions vs GROUP BY

### Comparison and When to Use Each

-- Traditional GROUP BY approach  
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  
GROUP BY department;  
  
-- Window function approach (keeps individual rows)  
SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 department,  
 salary,  
 COUNT(\*) OVER(PARTITION BY department) AS dept\_employee\_count,  
 AVG(salary) OVER(PARTITION BY department) AS dept\_avg\_salary,  
 MIN(salary) OVER(PARTITION BY department) AS dept\_min\_salary,  
 MAX(salary) OVER(PARTITION BY department) AS dept\_max\_salary,  
 SUM(salary) OVER(PARTITION BY department) AS dept\_total\_salary,  
   
 -- Individual vs. aggregate comparisons (impossible with GROUP BY alone)  
 salary - AVG(salary) OVER(PARTITION BY department) AS diff\_from\_dept\_avg,  
 salary / SUM(salary) OVER(PARTITION BY department) \* 100 AS pct\_of\_dept\_total,  
   
 -- Ranking within groups (impossible with GROUP BY alone)  
 RANK() OVER(PARTITION BY department ORDER BY salary DESC) AS dept\_salary\_rank  
FROM employees  
ORDER BY department, salary DESC;  
  
-- Combining GROUP BY with Window Functions  
WITH dept\_stats AS (  
 SELECT   
 department,  
 COUNT(\*) AS employee\_count,  
 AVG(salary) AS avg\_salary,  
 STDEV(salary) AS salary\_stdev  
 FROM employees  
 GROUP BY department  
),  
dept\_rankings AS (  
 SELECT   
 \*,  
 RANK() OVER(ORDER BY avg\_salary DESC) AS dept\_salary\_rank,  
 RANK() OVER(ORDER BY employee\_count DESC) AS dept\_size\_rank  
 FROM dept\_stats  
)  
SELECT \* FROM dept\_rankings  
ORDER BY dept\_salary\_rank;

### Complex Scenarios: GROUP BY + Window Functions

-- Advanced combination of GROUP BY and Window Functions  
WITH monthly\_employee\_sales AS (  
 -- First, aggregate to monthly level per employee  
 SELECT   
 sd.employee\_id,  
 e.department,  
 YEAR(sd.sale\_date) AS year,  
 MONTH(sd.sale\_date) AS month,  
 SUM(sd.amount) AS monthly\_sales,  
 COUNT(\*) AS monthly\_transactions  
 FROM sales\_data sd  
 JOIN employees e ON sd.employee\_id = e.employee\_id  
 GROUP BY sd.employee\_id, e.department, YEAR(sd.sale\_date), MONTH(sd.sale\_date)  
),  
enriched\_sales AS (  
 -- Then apply window functions  
 SELECT   
 \*,  
 -- Employee-level analytics  
 AVG(monthly\_sales) OVER(PARTITION BY employee\_id) AS employee\_avg\_monthly,  
 RANK() OVER(PARTITION BY employee\_id ORDER BY monthly\_sales DESC) AS employee\_month\_rank,  
   
 -- Department-level analytics  
 AVG(monthly\_sales) OVER(PARTITION BY department, year, month) AS dept\_monthly\_avg,  
 RANK() OVER(PARTITION BY department, year, month ORDER BY monthly\_sales DESC) AS dept\_monthly\_rank,  
   
 -- Time series analytics  
 LAG(monthly\_sales, 1) OVER(PARTITION BY employee\_id ORDER BY year, month) AS prev\_month\_sales,  
 LAG(monthly\_sales, 12) OVER(PARTITION BY employee\_id ORDER BY year, month) AS same\_month\_last\_year,  
   
 -- Rolling averages  
 AVG(monthly\_sales) OVER(  
 PARTITION BY employee\_id   
 ORDER BY year, month   
 ROWS BETWEEN 2 PRECEDING AND CURRENT ROW  
 ) AS rolling\_3month\_avg  
 FROM monthly\_employee\_sales  
)  
SELECT   
 \*,  
 -- Growth calculations  
 CASE   
 WHEN prev\_month\_sales > 0 THEN   
 (monthly\_sales - prev\_month\_sales) / prev\_month\_sales \* 100  
 ELSE NULL   
 END AS mom\_growth\_rate,  
   
 CASE   
 WHEN same\_month\_last\_year > 0 THEN   
 (monthly\_sales - same\_month\_last\_year) / same\_month\_last\_year \* 100  
 ELSE NULL   
 END AS yoy\_growth\_rate,  
   
 -- Performance vs. averages  
 (monthly\_sales - employee\_avg\_monthly) / employee\_avg\_monthly \* 100 AS vs\_personal\_avg\_pct,  
 (monthly\_sales - dept\_monthly\_avg) / dept\_monthly\_avg \* 100 AS vs\_dept\_avg\_pct  
FROM enriched\_sales  
ORDER BY employee\_id, year, month;

## Common Use Cases

### Top-N Analysis

-- Various Top-N scenarios using window functions  
  
-- Top 3 highest paid employees overall  
WITH ranked\_employees AS (  
 SELECT   
 \*,  
 ROW\_NUMBER() OVER(ORDER BY salary DESC) AS overall\_rank  
 FROM employees  
)  
SELECT \* FROM ranked\_employees   
WHERE overall\_rank <= 3;  
  
-- Top 2 employees per department by salary  
WITH dept\_rankings AS (  
 SELECT   
 \*,  
 ROW\_NUMBER() OVER(PARTITION BY department ORDER BY salary DESC) AS dept\_rank  
 FROM employees  
)  
SELECT \* FROM dept\_rankings   
WHERE dept\_rank <= 2  
ORDER BY department, dept\_rank;  
  
-- Top 20% performers using NTILE  
WITH performance\_tiers AS (  
 SELECT   
 \*,  
 NTILE(5) OVER(ORDER BY performance\_rating DESC) AS performance\_quintile  
 FROM employees  
)  
SELECT \* FROM performance\_tiers   
WHERE performance\_quintile = 1; -- Top 20%  
  
-- Dynamic Top-N with percentage  
WITH salary\_rankings AS (  
 SELECT   
 \*,  
 PERCENT\_RANK() OVER(ORDER BY salary DESC) AS salary\_percentile  
 FROM employees  
)  
SELECT \* FROM salary\_rankings   
WHERE salary\_percentile <= 0.3; -- Top 30%

### Gap Analysis

-- Finding gaps and outliers using window functions  
  
-- Salary gap analysis  
WITH salary\_analysis AS (  
 SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 department,  
 salary,  
 LAG(salary) OVER(ORDER BY salary DESC) AS next\_highest\_salary,  
 LEAD(salary) OVER(ORDER BY salary DESC) AS next\_lowest\_salary,  
   
 -- Department salary analysis  
 AVG(salary) OVER(PARTITION BY department) AS dept\_avg\_salary,  
 STDEV(salary) OVER(PARTITION BY department) AS dept\_salary\_stdev  
 FROM employees  
)  
SELECT   
 \*,  
 -- Gaps between consecutive salaries  
 ISNULL(next\_highest\_salary, salary) - salary AS gap\_to\_higher,  
 salary - ISNULL(next\_lowest\_salary, salary) AS gap\_to\_lower,  
   
 -- Z-score for outlier detection  
 (salary - dept\_avg\_salary) / NULLIF(dept\_salary\_stdev, 0) AS salary\_z\_score,  
   
 -- Flag outliers (beyond 2 standard deviations)  
 CASE   
 WHEN ABS((salary - dept\_avg\_salary) / NULLIF(dept\_salary\_stdev, 0)) > 2 THEN 'Outlier'  
 ELSE 'Normal'  
 END AS outlier\_flag  
FROM salary\_analysis  
ORDER BY salary DESC;  
  
-- Time gap analysis (hiring patterns)  
WITH hiring\_analysis AS (  
 SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 hire\_date,  
 LAG(hire\_date) OVER(ORDER BY hire\_date) AS prev\_hire\_date,  
 LEAD(hire\_date) OVER(ORDER BY hire\_date) AS next\_hire\_date  
 FROM employees  
)  
SELECT   
 \*,  
 DATEDIFF(DAY, prev\_hire\_date, hire\_date) AS days\_since\_prev\_hire,  
 DATEDIFF(DAY, hire\_date, next\_hire\_date) AS days\_to\_next\_hire,  
   
 -- Identify hiring clusters (within 30 days)  
 CASE   
 WHEN DATEDIFF(DAY, prev\_hire\_date, hire\_date) <= 30 THEN 'Cluster Hire'  
 WHEN DATEDIFF(DAY, prev\_hire\_date, hire\_date) > 180 THEN 'Gap After Long Break'  
 ELSE 'Normal'  
 END AS hiring\_pattern  
FROM hiring\_analysis  
ORDER BY hire\_date;

### Running Calculations

-- Various running calculation scenarios  
  
-- Financial running calculations  
WITH daily\_sales\_summary AS (  
 SELECT   
 sale\_date,  
 SUM(amount) AS daily\_total,  
 COUNT(\*) AS daily\_transactions,  
 AVG(amount) AS daily\_avg\_sale  
 FROM sales\_data  
 GROUP BY sale\_date  
)  
SELECT   
 sale\_date,  
 daily\_total,  
 daily\_transactions,  
 daily\_avg\_sale,  
   
 -- Running totals  
 SUM(daily\_total) OVER(ORDER BY sale\_date) AS running\_revenue,  
 SUM(daily\_transactions) OVER(ORDER BY sale\_date) AS running\_transaction\_count,  
   
 -- Running averages  
 AVG(daily\_total) OVER(ORDER BY sale\_date) AS running\_avg\_daily\_revenue,  
   
 -- Moving averages (7-day)  
 AVG(daily\_total) OVER(  
 ORDER BY sale\_date   
 ROWS BETWEEN 6 PRECEDING AND CURRENT ROW  
 ) AS moving\_7day\_avg,  
   
 -- Running max and min  
 MAX(daily\_total) OVER(ORDER BY sale\_date) AS running\_max\_daily,  
 MIN(daily\_total) OVER(ORDER BY sale\_date) AS running\_min\_daily,  
   
 -- Days since best/worst performance  
 sale\_date - FIRST\_VALUE(sale\_date) OVER(  
 ORDER BY daily\_total DESC, sale\_date  
 ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING  
 ) AS days\_since\_best\_day,  
   
 -- Percentage of total revenue so far  
 SUM(daily\_total) OVER(ORDER BY sale\_date) /   
 SUM(daily\_total) OVER() \* 100 AS pct\_of\_total\_revenue  
FROM daily\_sales\_summary  
ORDER BY sale\_date;

## Troubleshooting Window Functions

### Common Issues and Solutions

-- Issue 1: Understanding why LAST\_VALUE doesn't work as expected  
SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 salary,  
 hire\_date,  
   
 -- This might not give expected results due to default frame  
 LAST\_VALUE(salary) OVER(ORDER BY hire\_date) AS last\_value\_default,  
   
 -- Correct way to get last value  
 LAST\_VALUE(salary) OVER(  
 ORDER BY hire\_date   
 ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING  
 ) AS last\_value\_correct,  
   
 -- Alternative approach using FIRST\_VALUE with DESC order  
 FIRST\_VALUE(salary) OVER(ORDER BY hire\_date DESC) AS last\_value\_alternative  
FROM employees  
ORDER BY hire\_date;  
  
-- Issue 2: Handling NULLs in window functions  
UPDATE employees SET performance\_rating = NULL WHERE employee\_id = 3;  
  
SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 performance\_rating,  
   
 -- NULLs affect calculations  
 AVG(performance\_rating) OVER() AS avg\_with\_nulls,  
   
 -- Exclude NULLs explicitly  
 AVG(CASE WHEN performance\_rating IS NOT NULL THEN performance\_rating END) OVER() AS avg\_excluding\_nulls,  
   
 -- Using COALESCE for default values  
 AVG(COALESCE(performance\_rating, 3.0)) OVER() AS avg\_with\_default,  
   
 -- LAG with NULLs  
 LAG(performance\_rating) OVER(ORDER BY hire\_date) AS prev\_rating,  
 LAG(COALESCE(performance\_rating, 0)) OVER(ORDER BY hire\_date) AS prev\_rating\_with\_default  
FROM employees  
ORDER BY hire\_date;  
  
-- Reset the NULL value  
UPDATE employees SET performance\_rating = 3.8 WHERE employee\_id = 3;  
  
-- Issue 3: Performance problems with window functions  
-- Use CASE to avoid multiple window function calls  
SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 department,  
 salary,  
   
 -- Inefficient: Multiple similar window function calls  
 -- COUNT(\*) OVER(PARTITION BY department) AS dept\_count,  
 -- SUM(salary) OVER(PARTITION BY department) AS dept\_total,  
 -- AVG(salary) OVER(PARTITION BY department) AS dept\_avg,  
   
 -- More efficient: Use CTE to calculate once  
 dept\_stats.dept\_count,  
 dept\_stats.dept\_total,  
 dept\_stats.dept\_avg,  
   
 -- Individual calculations  
 salary - dept\_stats.dept\_avg AS diff\_from\_avg  
FROM employees e  
CROSS APPLY (  
 SELECT   
 COUNT(\*) AS dept\_count,  
 SUM(salary) AS dept\_total,  
 AVG(salary) AS dept\_avg  
 FROM employees e2  
 WHERE e2.department = e.department  
) AS dept\_stats  
ORDER BY department, salary DESC;

### Debugging Window Function Logic

-- Create a debugging framework for window functions  
WITH debug\_window\_functions AS (  
 SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 department,  
 salary,  
 hire\_date,  
   
 -- Debug information  
 'Original Order' AS sort\_method,  
 ROW\_NUMBER() OVER(ORDER BY employee\_id) AS debug\_original\_order,  
   
 -- Show partition information  
 COUNT(\*) OVER(PARTITION BY department) AS partition\_size,  
 ROW\_NUMBER() OVER(PARTITION BY department ORDER BY salary DESC) AS partition\_position,  
   
 -- Show frame information for running sum  
 SUM(salary) OVER(ORDER BY hire\_date) AS running\_sum,  
 salary AS current\_salary,  
   
 -- Previous and next values for verification  
 LAG(salary) OVER(ORDER BY hire\_date) AS prev\_salary,  
 LEAD(salary) OVER(ORDER BY hire\_date) AS next\_salary  
 FROM employees  
)  
SELECT   
 \*,  
 -- Verify running sum calculation  
 CASE   
 WHEN debug\_original\_order = 1 THEN current\_salary  
 ELSE prev\_salary + current\_salary  
 END AS manual\_running\_sum\_check  
FROM debug\_window\_functions  
ORDER BY hire\_date;  
  
-- Performance debugging query  
WITH window\_function\_performance AS (  
 SELECT   
 employee\_id,  
 department,  
 salary,  
   
 -- Time the execution of different approaches  
 AVG(salary) OVER(PARTITION BY department) AS window\_avg,  
 (SELECT AVG(salary) FROM employees e2 WHERE e2.department = e.department) AS subquery\_avg  
 FROM employees e  
)  
SELECT \* FROM window\_function\_performance;

## Best Practices

### Performance Best Practices

-- Best Practice 1: Use appropriate indexing  
-- CREATE INDEX IX\_employees\_dept\_salary ON employees (department, salary DESC);  
-- CREATE INDEX IX\_sales\_emp\_date ON sales\_data (employee\_id, sale\_date);  
  
-- Best Practice 2: Minimize window function calls  
WITH base\_calculations AS (  
 SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 department,  
 salary,  
 COUNT(\*) OVER(PARTITION BY department) AS dept\_count,  
 SUM(salary) OVER(PARTITION BY department) AS dept\_total,  
 AVG(salary) OVER(PARTITION BY department) AS dept\_avg,  
 STDEV(salary) OVER(PARTITION BY department) AS dept\_stdev  
 FROM employees  
)  
SELECT   
 \*,  
 -- Derived calculations using base results  
 salary - dept\_avg AS salary\_diff\_from\_avg,  
 salary / dept\_total \* 100 AS salary\_pct\_of\_total,  
 (salary - dept\_avg) / NULLIF(dept\_stdev, 0) AS salary\_z\_score  
FROM base\_calculations;  
  
-- Best Practice 3: Use CTEs for complex window function logic  
WITH employee\_rankings AS (  
 SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 department,  
 salary,  
 performance\_rating,  
 ROW\_NUMBER() OVER(PARTITION BY department ORDER BY salary DESC) AS dept\_salary\_rank,  
 ROW\_NUMBER() OVER(PARTITION BY department ORDER BY performance\_rating DESC) AS dept\_perf\_rank  
 FROM employees  
),  
top\_performers AS (  
 SELECT \*  
 FROM employee\_rankings  
 WHERE dept\_salary\_rank <= 2 OR dept\_perf\_rank <= 2  
)  
SELECT \* FROM top\_performers  
ORDER BY department, dept\_salary\_rank;

### Code Organization Best Practices

-- Best Practice 4: Clear naming and documentation  
SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 department,  
 salary,  
   
 -- Department-level analytics  
 COUNT(\*) OVER(PARTITION BY department) AS dept\_employee\_count,  
 AVG(salary) OVER(PARTITION BY department) AS dept\_average\_salary,  
   
 -- Individual performance metrics  
 RANK() OVER(PARTITION BY department ORDER BY salary DESC) AS dept\_salary\_rank,  
 PERCENT\_RANK() OVER(ORDER BY salary) AS overall\_salary\_percentile,  
   
 -- Comparative analysis  
 salary - AVG(salary) OVER(PARTITION BY department) AS salary\_vs\_dept\_average,  
   
 -- Time-based analysis  
 DATEDIFF(MONTH, hire\_date, GETDATE()) AS months\_employed,  
 ROW\_NUMBER() OVER(PARTITION BY department ORDER BY hire\_date) AS hire\_sequence\_in\_dept  
FROM employees  
ORDER BY department, salary DESC;  
  
-- Best Practice 5: Handle edge cases  
SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 salary,  
 performance\_rating,  
   
 -- Handle division by zero  
 CASE   
 WHEN STDEV(salary) OVER() > 0 THEN  
 (salary - AVG(salary) OVER()) / STDEV(salary) OVER()  
 ELSE 0  
 END AS salary\_z\_score,  
   
 -- Handle NULLs in calculations  
 COALESCE(  
 LAG(performance\_rating) OVER(ORDER BY hire\_date),  
 AVG(performance\_rating) OVER()  
 ) AS prev\_rating\_or\_avg,  
   
 -- Ensure meaningful percentiles  
 CASE   
 WHEN COUNT(\*) OVER() > 1 THEN  
 PERCENT\_RANK() OVER(ORDER BY salary)  
 ELSE 0.5  
 END AS salary\_percentile  
FROM employees;

# **Subqueries**

# Complete Guide to Subqueries in SSMS

## Table of Contents

1. [What are Subqueries?](#what-are-subqueries)
2. [Types of Subqueries](#types-of-subqueries)
3. [Subquery Locations](#subquery-locations)
4. [Single-Row Subqueries](#single-row-subqueries)
5. [Multiple-Row Subqueries](#multiple-row-subqueries)
6. [Correlated Subqueries](#correlated-subqueries)
7. [Subqueries in SELECT Clause](#subqueries-in-select-clause)
8. [Subqueries in FROM Clause (Derived Tables)](#subqueries-in-from-clause)
9. [Subqueries in WHERE Clause](#subqueries-in-where-clause)
10. [Subqueries with EXISTS and NOT EXISTS](#subqueries-with-exists-and-not-exists)
11. [Subqueries with IN and NOT IN](#subqueries-with-in-and-not-in)
12. [Subqueries with ANY and ALL](#subqueries-with-any-and-all)
13. [Nested Subqueries](#nested-subqueries)
14. [Subqueries vs JOINs](#subqueries-vs-joins)
15. [Performance Considerations](#performance-considerations)
16. [Common Use Cases](#common-use-cases)
17. [Best Practices](#best-practices)
18. [Troubleshooting Subqueries](#troubleshooting-subqueries)

## What are Subqueries?

A **Subquery** (also called an inner query or nested query) is a query nested inside another SQL statement. Subqueries can be used in SELECT, INSERT, UPDATE, and DELETE statements, providing a powerful way to perform complex data retrieval and manipulation.

### Key Characteristics:

* **Nested Execution**: Inner query executes first, result used by outer query
* **Encapsulation**: Enclosed in parentheses
* **Independence**: Can often be tested separately
* **Flexibility**: Can return single value, single row, or multiple rows
* **Scope**: Can reference outer query columns (correlated) or be independent

### Benefits:

* **Modularity**: Break complex queries into logical parts
* **Readability**: Sometimes clearer than complex JOINs
* **Dynamic Filtering**: Filter based on calculated values
* **Data Validation**: Check for existence of related data
* **Intermediate Results**: Use calculated values in further operations

## Types of Subqueries

### 1. Single-Row Subqueries

Return exactly one row with one or more columns.

### 2. Multiple-Row Subqueries

Return multiple rows (used with IN, ANY, ALL, EXISTS).

### 3. Multiple-Column Subqueries

Return multiple columns in each row.

### 4. Correlated Subqueries

Reference columns from the outer query.

### 5. Non-Correlated Subqueries

Independent of outer query, execute once.

## Sample Data Setup

-- Create comprehensive 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\_id INT,  
 manager\_id INT,  
 salary DECIMAL(10,2),  
 hire\_date DATE,  
 job\_title VARCHAR(50),  
 commission\_pct DECIMAL(3,2),  
 is\_active BIT DEFAULT 1  
);  
  
CREATE TABLE departments (  
 department\_id INT PRIMARY KEY,  
 department\_name VARCHAR(50),  
 location VARCHAR(50),  
 budget DECIMAL(12,2),  
 manager\_id INT  
);  
  
CREATE TABLE projects (  
 project\_id INT IDENTITY(1,1) PRIMARY KEY,  
 project\_name VARCHAR(100),  
 department\_id INT,  
 start\_date DATE,  
 end\_date DATE,  
 budget DECIMAL(12,2),  
 status VARCHAR(20)  
);  
  
CREATE TABLE project\_assignments (  
 assignment\_id INT IDENTITY(1,1) PRIMARY KEY,  
 employee\_id INT,  
 project\_id INT,  
 assigned\_date DATE,  
 hours\_allocated INT,  
 role VARCHAR(50)  
);  
  
CREATE TABLE salaries\_history (  
 history\_id INT IDENTITY(1,1) PRIMARY KEY,  
 employee\_id INT,  
 salary DECIMAL(10,2),  
 effective\_date DATE,  
 end\_date DATE,  
 change\_reason VARCHAR(100)  
);  
  
-- Insert sample data  
INSERT INTO departments VALUES  
(10, 'IT', 'New York', 500000, NULL),  
(20, 'Sales', 'Chicago', 400000, NULL),  
(30, 'HR', 'Los Angeles', 250000, NULL),  
(40, 'Finance', 'Boston', 350000, NULL),  
(50, 'Marketing', 'San Francisco', 300000, NULL);  
  
INSERT INTO employees VALUES  
('John', 'Doe', 'john.doe@company.com', '555-0001', 10, NULL, 85000, '2018-01-15', 'IT Manager', NULL, 1),  
('Jane', 'Smith', 'jane.smith@company.com', '555-0002', 10, 1, 75000, '2019-03-20', 'Senior Developer', NULL, 1),  
('Bob', 'Johnson', 'bob.johnson@company.com', '555-0003', 20, NULL, 90000, '2017-06-10', 'Sales Manager', 0.10, 1),  
('Alice', 'Brown', 'alice.brown@company.com', '555-0004', 20, 3, 65000, '2020-02-14', 'Sales Rep', 0.15, 1),  
('Charlie', 'Wilson', 'charlie.wilson@company.com', '555-0005', 30, NULL, 70000, '2019-08-05', 'HR Manager', NULL, 1),  
('Diana', 'Ross', 'diana.ross@company.com', '555-0006', 10, 1, 72000, '2020-01-10', 'Developer', NULL, 1),  
('Elvis', 'King', 'elvis.king@company.com', '555-0007', 40, NULL, 95000, '2016-11-20', 'Finance Manager', NULL, 1),  
('Frank', 'Miller', 'frank.miller@company.com', '555-0008', 20, 3, 68000, '2021-04-15', 'Sales Rep', 0.12, 1),  
('Grace', 'Lee', 'grace.lee@company.com', '555-0009', 50, NULL, 80000, '2019-09-01', 'Marketing Manager', NULL, 1),  
('Henry', 'Ford', 'henry.ford@company.com', '555-0010', 10, 1, 70000, '2021-07-20', 'Developer', NULL, 1);  
  
INSERT INTO projects VALUES  
('Website Redesign', 10, '2023-01-01', '2023-06-30', 150000, 'Completed'),  
('CRM Implementation', 20, '2023-02-15', '2023-12-31', 200000, 'In Progress'),  
('HR Portal', 30, '2023-03-01', '2023-09-30', 100000, 'In Progress'),  
('Financial Audit System', 40, '2023-01-15', '2023-08-31', 180000, 'Completed'),  
('Marketing Campaign', 50, '2023-04-01', '2023-10-31', 120000, 'In Progress'),  
('Mobile App', 10, '2023-05-01', '2024-03-31', 250000, 'Planning');  
  
INSERT INTO project\_assignments VALUES  
(1, 1, '2023-01-01', 160, 'Project Lead'),  
(2, 1, '2023-01-01', 120, 'Developer'),  
(6, 1, '2023-01-01', 120, 'Developer'),  
(3, 2, '2023-02-15', 80, 'Project Lead'),  
(4, 2, '2023-02-15', 40, 'Sales Analyst'),  
(5, 3, '2023-03-01', 100, 'Project Lead'),  
(7, 4, '2023-01-15', 140, 'Project Lead'),  
(9, 5, '2023-04-01', 120, 'Project Lead'),  
(1, 6, '2023-05-01', 160, 'Project Lead'),  
(2, 6, '2023-05-01', 80, 'Developer');  
  
INSERT INTO salaries\_history VALUES  
(1, 75000, '2018-01-15', '2020-01-14', 'Initial Hire'),  
(1, 80000, '2020-01-15', '2022-01-14', 'Annual Review'),  
(1, 85000, '2022-01-15', NULL, 'Promotion'),  
(2, 65000, '2019-03-20', '2021-03-19', 'Initial Hire'),  
(2, 70000, '2021-03-20', '2023-03-19', 'Annual Review'),  
(2, 75000, '2023-03-20', NULL, 'Performance Increase');

## Single-Row Subqueries

Single-row subqueries return exactly one row and can be used with comparison operators (=, >, <, >=, <=, <>).

-- Basic single-row subquery examples  
  
-- 1. Find employees earning more than the average salary  
SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 salary,  
 (SELECT AVG(salary) FROM employees) AS avg\_salary  
FROM employees  
WHERE salary > (SELECT AVG(salary) FROM employees);  
  
-- 2. Find employee with the highest salary  
SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 salary  
FROM employees  
WHERE salary = (SELECT MAX(salary) FROM employees);  
  
-- 3. Find employees in the department with the largest budget  
SELECT   
 e.employee\_id,  
 e.first\_name + ' ' + e.last\_name AS employee\_name,  
 e.department\_id,  
 d.department\_name,  
 d.budget  
FROM employees e  
JOIN departments d ON e.department\_id = d.department\_id  
WHERE d.budget = (SELECT MAX(budget) FROM departments);  
  
-- 4. Compare individual salary to average in their department  
SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 department\_id,  
 salary,  
 (SELECT AVG(salary)   
 FROM employees e2   
 WHERE e2.department\_id = e.department\_id) AS dept\_avg\_salary,  
 salary - (SELECT AVG(salary)   
 FROM employees e2   
 WHERE e2.department\_id = e.department\_id) AS diff\_from\_dept\_avg  
FROM employees e  
ORDER BY department\_id, salary DESC;  
  
-- 5. Find projects with budget higher than average project budget  
SELECT   
 project\_id,  
 project\_name,  
 budget,  
 (SELECT AVG(budget) FROM projects) AS avg\_project\_budget  
FROM projects  
WHERE budget > (SELECT AVG(budget) FROM projects)  
ORDER BY budget DESC;

## Multiple-Row Subqueries

Multiple-row subqueries return more than one row and must be used with operators that can handle multiple values (IN, ANY, ALL, EXISTS).

-- Multiple-row subquery examples  
  
-- 1. Find employees working in departments located in specific cities  
SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 department\_id  
FROM employees  
WHERE department\_id IN (  
 SELECT department\_id   
 FROM departments   
 WHERE location IN ('New York', 'Chicago')  
);  
  
-- 2. Find employees who are managers (appear in manager\_id column)  
SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 job\_title  
FROM employees  
WHERE employee\_id IN (  
 SELECT DISTINCT manager\_id   
 FROM employees   
 WHERE manager\_id IS NOT NULL  
);  
  
-- 3. Find departments that have at least one active project  
SELECT   
 department\_id,  
 department\_name,  
 location  
FROM departments  
WHERE department\_id IN (  
 SELECT DISTINCT department\_id   
 FROM projects   
 WHERE status IN ('In Progress', 'Planning')  
);  
  
-- 4. Find employees assigned to multiple projects  
SELECT   
 e.employee\_id,  
 e.first\_name + ' ' + e.last\_name AS employee\_name,  
 COUNT(pa.project\_id) AS project\_count  
FROM employees e  
WHERE employee\_id IN (  
 SELECT employee\_id   
 FROM project\_assignments   
 GROUP BY employee\_id   
 HAVING COUNT(project\_id) > 1  
)  
GROUP BY e.employee\_id, e.first\_name, e.last\_name;  
  
-- 5. Find projects in departments with more than 2 employees  
SELECT   
 project\_id,  
 project\_name,  
 department\_id,  
 status  
FROM projects  
WHERE department\_id IN (  
 SELECT department\_id   
 FROM employees   
 GROUP BY department\_id   
 HAVING COUNT(\*) > 2  
);

## Correlated Subqueries

Correlated subqueries reference columns from the outer query and execute once for each row processed by the outer query.

-- Correlated subquery examples  
  
-- 1. Find employees earning more than average in their department  
SELECT   
 e.employee\_id,  
 e.first\_name + ' ' + e.last\_name AS employee\_name,  
 e.department\_id,  
 e.salary,  
 (SELECT AVG(salary)   
 FROM employees e2   
 WHERE e2.department\_id = e.department\_id) AS dept\_avg  
FROM employees e  
WHERE e.salary > (  
 SELECT AVG(salary)   
 FROM employees e2   
 WHERE e2.department\_id = e.department\_id  
);  
  
-- 2. Find employees who earn more than their manager  
SELECT   
 e.employee\_id,  
 e.first\_name + ' ' + e.last\_name AS employee\_name,  
 e.salary AS employee\_salary,  
 e.manager\_id,  
 (SELECT salary   
 FROM employees m   
 WHERE m.employee\_id = e.manager\_id) AS manager\_salary  
FROM employees e  
WHERE e.salary > (  
 SELECT salary   
 FROM employees m   
 WHERE m.employee\_id = e.manager\_id  
);  
  
-- 3. Find departments with above-average number of employees  
SELECT   
 d.department\_id,  
 d.department\_name,  
 (SELECT COUNT(\*)   
 FROM employees e   
 WHERE e.department\_id = d.department\_id) AS employee\_count,  
 (SELECT AVG(emp\_count)   
 FROM (  
 SELECT COUNT(\*) AS emp\_count   
 FROM employees   
 GROUP BY department\_id  
 ) AS dept\_counts) AS avg\_employees\_per\_dept  
FROM departments d  
WHERE (SELECT COUNT(\*)   
 FROM employees e   
 WHERE e.department\_id = d.department\_id) >   
 (SELECT AVG(emp\_count)   
 FROM (  
 SELECT COUNT(\*) AS emp\_count   
 FROM employees   
 GROUP BY department\_id  
 ) AS dept\_counts);  
  
-- 4. Find latest salary for each employee  
SELECT   
 e.employee\_id,  
 e.first\_name + ' ' + e.last\_name AS employee\_name,  
 (SELECT TOP 1 salary   
 FROM salaries\_history sh   
 WHERE sh.employee\_id = e.employee\_id   
 ORDER BY effective\_date DESC) AS latest\_historical\_salary,  
 e.salary AS current\_salary  
FROM employees e  
WHERE EXISTS (  
 SELECT 1   
 FROM salaries\_history sh   
 WHERE sh.employee\_id = e.employee\_id  
);  
  
-- 5. Find employees with salary higher than all employees in HR department  
SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 department\_id,  
 salary  
FROM employees e  
WHERE salary > ALL (  
 SELECT salary   
 FROM employees   
 WHERE department\_id = (  
 SELECT department\_id   
 FROM departments   
 WHERE department\_name = 'HR'  
 )  
);

## Subqueries in SELECT Clause

Subqueries in the SELECT clause (also called scalar subqueries) return a single value for each row of the outer query.

-- Subqueries in SELECT clause examples  
  
-- 1. Show employee info with department details  
SELECT   
 e.employee\_id,  
 e.first\_name + ' ' + e.last\_name AS employee\_name,  
 e.salary,  
 (SELECT department\_name   
 FROM departments d   
 WHERE d.department\_id = e.department\_id) AS department\_name,  
 (SELECT location   
 FROM departments d   
 WHERE d.department\_id = e.department\_id) AS department\_location,  
 (SELECT COUNT(\*)   
 FROM employees e2   
 WHERE e2.department\_id = e.department\_id) AS dept\_employee\_count  
FROM employees e;  
  
-- 2. Calculate various aggregates for each employee  
SELECT   
 e.employee\_id,  
 e.first\_name + ' ' + e.last\_name AS employee\_name,  
 e.salary,  
 (SELECT AVG(salary) FROM employees) AS overall\_avg\_salary,  
 (SELECT AVG(salary)   
 FROM employees e2   
 WHERE e2.department\_id = e.department\_id) AS dept\_avg\_salary,  
 (SELECT MAX(salary)   
 FROM employees e2   
 WHERE e2.department\_id = e.department\_id) AS dept\_max\_salary,  
 (SELECT COUNT(\*)   
 FROM project\_assignments pa   
 WHERE pa.employee\_id = e.employee\_id) AS projects\_assigned  
FROM employees e;  
  
-- 3. Show project information with related counts  
SELECT   
 p.project\_id,  
 p.project\_name,  
 p.budget,  
 (SELECT department\_name   
 FROM departments d   
 WHERE d.department\_id = p.department\_id) AS department\_name,  
 (SELECT COUNT(\*)   
 FROM project\_assignments pa   
 WHERE pa.project\_id = p.project\_id) AS team\_size,  
 (SELECT SUM(hours\_allocated)   
 FROM project\_assignments pa   
 WHERE pa.project\_id = p.project\_id) AS total\_hours\_allocated,  
 p.budget / NULLIF((SELECT COUNT(\*)   
 FROM project\_assignments pa   
 WHERE pa.project\_id = p.project\_id), 0) AS budget\_per\_team\_member  
FROM projects p;  
  
-- 4. Calculate salary percentiles  
SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 salary,  
 (SELECT COUNT(\*)   
 FROM employees e2   
 WHERE e2.salary < e.salary) AS employees\_earning\_less,  
 (SELECT COUNT(\*)   
 FROM employees e2   
 WHERE e2.salary > e.salary) AS employees\_earning\_more,  
 CAST((SELECT COUNT(\*)   
 FROM employees e2   
 WHERE e2.salary < e.salary) AS FLOAT) /   
 (SELECT COUNT(\*) FROM employees) \* 100 AS salary\_percentile  
FROM employees e  
ORDER BY salary DESC;  
  
-- 5. Show manager information for each employee  
SELECT   
 e.employee\_id,  
 e.first\_name + ' ' + e.last\_name AS employee\_name,  
 e.job\_title,  
 e.salary,  
 CASE   
 WHEN e.manager\_id IS NOT NULL THEN  
 (SELECT m.first\_name + ' ' + m.last\_name   
 FROM employees m   
 WHERE m.employee\_id = e.manager\_id)  
 ELSE 'No Manager'  
 END AS manager\_name,  
 CASE   
 WHEN e.manager\_id IS NOT NULL THEN  
 (SELECT m.salary   
 FROM employees m   
 WHERE m.employee\_id = e.manager\_id)  
 ELSE NULL  
 END AS manager\_salary  
FROM employees e;

## Subqueries in FROM Clause (Derived Tables)

Subqueries in the FROM clause create temporary result sets (derived tables) that can be queried like regular tables.

-- Derived table examples  
  
-- 1. Calculate department statistics and use them in outer query  
SELECT   
 dept\_stats.department\_id,  
 dept\_stats.department\_name,  
 dept\_stats.employee\_count,  
 dept\_stats.avg\_salary,  
 dept\_stats.total\_salary,  
 dept\_stats.salary\_range,  
 CASE   
 WHEN dept\_stats.avg\_salary > overall\_stats.overall\_avg THEN 'Above Average'  
 WHEN dept\_stats.avg\_salary < overall\_stats.overall\_avg THEN 'Below Average'  
 ELSE 'Average'  
 END AS salary\_category  
FROM (  
 SELECT   
 d.department\_id,  
 d.department\_name,  
 COUNT(e.employee\_id) AS employee\_count,  
 AVG(e.salary) AS avg\_salary,  
 SUM(e.salary) AS total\_salary,  
 MAX(e.salary) - MIN(e.salary) AS salary\_range  
 FROM departments d  
 LEFT JOIN employees e ON d.department\_id = e.department\_id  
 GROUP BY d.department\_id, d.department\_name  
) AS dept\_stats  
CROSS JOIN (  
 SELECT AVG(salary) AS overall\_avg FROM employees  
) AS overall\_stats  
ORDER BY dept\_stats.avg\_salary DESC;  
  
-- 2. Rank departments by various metrics  
SELECT   
 dept\_metrics.\*,  
 ROW\_NUMBER() OVER(ORDER BY avg\_salary DESC) AS salary\_rank,  
 ROW\_NUMBER() OVER(ORDER BY employee\_count DESC) AS size\_rank,  
 ROW\_NUMBER() OVER(ORDER BY project\_count DESC) AS project\_rank  
FROM (  
 SELECT   
 d.department\_id,  
 d.department\_name,  
 COUNT(DISTINCT e.employee\_id) AS employee\_count,  
 AVG(e.salary) AS avg\_salary,  
 COUNT(DISTINCT p.project\_id) AS project\_count,  
 SUM(p.budget) AS total\_project\_budget  
 FROM departments d  
 LEFT JOIN employees e ON d.department\_id = e.department\_id  
 LEFT JOIN projects p ON d.department\_id = p.department\_id  
 GROUP BY d.department\_id, d.department\_name  
) AS dept\_metrics;  
  
-- 3. Join multiple derived tables  
SELECT   
 emp\_proj.employee\_id,  
 emp\_proj.employee\_name,  
 emp\_proj.project\_count,  
 emp\_proj.total\_hours,  
 sal\_hist.salary\_changes,  
 sal\_hist.total\_salary\_increase  
FROM (  
 SELECT   
 e.employee\_id,  
 e.first\_name + ' ' + e.last\_name AS employee\_name,  
 COUNT(pa.project\_id) AS project\_count,  
 SUM(pa.hours\_allocated) AS total\_hours  
 FROM employees e  
 LEFT JOIN project\_assignments pa ON e.employee\_id = pa.employee\_id  
 GROUP BY e.employee\_id, e.first\_name, e.last\_name  
) AS emp\_proj  
LEFT JOIN (  
 SELECT   
 employee\_id,  
 COUNT(\*) AS salary\_changes,  
 MAX(salary) - MIN(salary) AS total\_salary\_increase  
 FROM salaries\_history  
 GROUP BY employee\_id  
) AS sal\_hist ON emp\_proj.employee\_id = sal\_hist.employee\_id;  
  
-- 4. Complex aggregation with derived tables  
SELECT   
 high\_earners.department\_name,  
 high\_earners.high\_earner\_count,  
 high\_earners.high\_earner\_avg\_salary,  
 all\_emps.total\_employees,  
 all\_emps.dept\_avg\_salary,  
 CAST(high\_earners.high\_earner\_count AS FLOAT) / all\_emps.total\_employees \* 100 AS pct\_high\_earners  
FROM (  
 SELECT   
 d.department\_name,  
 COUNT(\*) AS high\_earner\_count,  
 AVG(e.salary) AS high\_earner\_avg\_salary  
 FROM employees e  
 JOIN departments d ON e.department\_id = d.department\_id  
 WHERE e.salary > (SELECT AVG(salary) FROM employees)  
 GROUP BY d.department\_name  
) AS high\_earners  
JOIN (  
 SELECT   
 d.department\_name,  
 COUNT(\*) AS total\_employees,  
 AVG(e.salary) AS dept\_avg\_salary  
 FROM employees e  
 JOIN departments d ON e.department\_id = d.department\_id  
 GROUP BY d.department\_name  
) AS all\_emps ON high\_earners.department\_name = all\_emps.department\_name;  
  
-- 5. Nested derived tables  
SELECT   
 final\_results.\*,  
 RANK() OVER(ORDER BY performance\_score DESC) AS overall\_rank  
FROM (  
 SELECT   
 dept\_performance.department\_name,  
 dept\_performance.avg\_salary,  
 dept\_performance.project\_count,  
 dept\_performance.employee\_count,  
 (dept\_performance.project\_count \* 0.4 +   
 dept\_performance.employee\_count \* 0.3 +   
 dept\_performance.avg\_salary / 1000 \* 0.3) AS performance\_score  
 FROM (  
 SELECT   
 d.department\_name,  
 AVG(e.salary) AS avg\_salary,  
 COUNT(DISTINCT p.project\_id) AS project\_count,  
 COUNT(DISTINCT e.employee\_id) AS employee\_count  
 FROM departments d  
 LEFT JOIN employees e ON d.department\_id = e.department\_id  
 LEFT JOIN projects p ON d.department\_id = p.department\_id  
 GROUP BY d.department\_name  
 ) AS dept\_performance  
) AS final\_results  
ORDER BY performance\_score DESC;

## Subqueries in WHERE Clause

Subqueries in WHERE clause are the most common type, used for filtering data based on complex conditions.

-- WHERE clause subquery examples  
  
-- 1. Find employees in departments with budget > $300,000  
SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 department\_id,  
 salary  
FROM employees  
WHERE department\_id IN (  
 SELECT department\_id   
 FROM departments   
 WHERE budget > 300000  
);  
  
-- 2. Find employees who haven't been assigned to any project  
SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 job\_title  
FROM employees  
WHERE employee\_id NOT IN (  
 SELECT DISTINCT employee\_id   
 FROM project\_assignments  
);  
  
-- 3. Find departments with above-average budgets  
SELECT   
 department\_id,  
 department\_name,  
 budget  
FROM departments  
WHERE budget > (SELECT AVG(budget) FROM departments)  
ORDER BY budget DESC;  
  
-- 4. Find employees hired after the average hire date  
SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 hire\_date  
FROM employees  
WHERE hire\_date > (SELECT AVG(CAST(hire\_date AS FLOAT)) FROM employees)  
ORDER BY hire\_date;  
  
-- 5. Complex filtering with multiple subqueries  
SELECT   
 employee\_id,  
 first\_name + ' ' + last\_name AS employee\_name,  
 department\_id,  
 salary  
FROM employees  
WHERE department\_id IN (  
 SELECT department\_id   
 FROM departments   
 WHERE location IN ('New York', 'Chicago')  
)  
AND salary > (SELECT AVG(salary) FROM employees)  
AND employee\_id IN (  
 SELECT employee\_id   
 FROM project\_assignments   
 GROUP BY employee\_id   
 HAVING COUNT(project\_id) >= 2  
);  
  
-- 6. Find projects in departments with more employees than average  
SELECT   
 project\_id,  
 project\_name,  
 department\_id  
FROM projects  
WHERE department\_id IN (  
 SELECT department\_id  
 FROM (  
 SELECT   
 department\_id,  
 COUNT(\*) AS emp\_count  
 FROM employees  
 GROUP BY department\_id  
 HAVING COUNT(\*) > (  
 SELECT AVG(dept\_count)  
 FROM (  
 SELECT COUNT(\*) AS dept\_count  
 FROM employees  
 GROUP BY department\_id  
 ) AS dept\_counts  
 )  
 ) AS large\_depts  
);

## Subqueries with EXISTS and NOT EXISTS

EXISTS checks for the existence of rows returned by a subquery. It’s often more efficient than IN for large datasets.

```sql – EXISTS and NOT EXISTS examples

– 1. Find employees who are assigned to at least one project SELECT e.employee\_id, e.first\_name + ’ ’ + e.last\_name AS employee\_name, e.department\_id, e.job\_title FROM employees e WHERE EXISTS ( SELECT 1 FROM project\_assignments pa WHERE pa.employee\_id = e.employee\_id );

– 2. Find employees who are NOT assigned to any project SELECT e.employee\_id, e.first\_name + ’ ’ + e.last\_name AS employee\_name, e.department\_id, e.job\_title FROM employees e WHERE NOT EXISTS ( SELECT 1 FROM project\_assignments pa WHERE pa.employee\_id = e.employee\_id );

– 3. Find departments that have at least one completed project SELECT d.department\_id, d.department\_name, d.location FROM departments d WHERE EXISTS ( SELECT 1 FROM projects p WHERE p.department\_id = d.department\_id AND p.status = ‘Completed’ );

– 4. Find employees who have received salary increases SELECT e.employee\_id, e.first\_name + ’ ’ + e.last\_name AS employee\_name, e.salary AS current\_salary FROM employees e WHERE EXISTS ( SELECT 1 FROM salaries\_history sh WHERE sh.employee\_id = e.employee\_id );

– 5. Find employees earning more than any employee in HR department SELECT e1.employee\_id, e1.first\_name + ’ ’ + e1.last\_name AS employee\_name, e1.department\_id, e1.salary FROM employees e1 WHERE EXISTS ( SELECT 1 FROM employees e2 WHERE e2.department\_id = ( SELECT department\_id FROM departments WHERE department\_name = ‘HR’ ) AND e1.salary > e2.salary );

– 6. Find projects with all team members from same department SELECT p.project\_id, p.project\_name, p.department\_id FROM projects p WHERE NOT EXISTS ( SELECT 1 FROM project\_assignments pa JOIN employees e ON pa.employee\_id = e.employee\_id WHERE pa.project\_id = p.project\_id AND e.department\_id != p.department\_id ) AND EXISTS ( SELECT 1 FROM project\_assignments pa WHERE pa.project\_id = p.project\_id );

– 7. Find managers who manage at least one high-earning employee (>$75,000) SELECT DISTINCT m.employee\_id, m.first\_name + ’ ’ + m.last\_name AS manager\_name, m.job\_title FROM employees m WHERE EXISTS ( SELECT 1 FROM employees e WHERE e.manager\_id = m.employee\_id AND e.salary > 75000 );