

Prepared by Randy Fadler

August 2025

Table of Contents

[Chapter 1: Introduction to SQL Server 6](#_Toc207197754)

[1.1 Overview of SQL Server 6](#_Toc207197755)

[1.2 SQL Server Architecture 6](#_Toc207197756)

[1.3 Setting Up Your Environment 7](#_Toc207197757)

[1.4 Creating Your First Database 7](#_Toc207197758)

[1.5 Basic T-SQL Syntax 8](#_Toc207197759)

[1.6 Basic Data Types 9](#_Toc207197760)

[1.7 Constraints and Relationships 9](#_Toc207197761)

[1.8 Basic Querying with SELECT 10](#_Toc207197762)

[1.9 Error Handling Basics 11](#_Toc207197763)

[1.10 Best Practices for Beginners 11](#_Toc207197764)

[1.11 Advanced: Exploring System Views 12](#_Toc207197765)

[**1.12 Exercises** 12](#_Toc207197766)

[**1.13 Conclusion** 12](#_Toc207197767)

[Chapter 2: Basic SQL Queries 13](#_Toc207197768)

[2.1 Introduction to CRUD Operations 13](#_Toc207197769)

[2.2 INSERT: Creating Data 14](#_Toc207197770)

[2.4 UPDATE: Modifying Data 17](#_Toc207197771)

[2.5 DELETE: Removing Data 18](#_Toc207197772)

[2.6 Aggregation and Grouping 18](#_Toc207197773)

[2.7 Handling NULLs 19](#_Toc207197774)

[2.8 Best Practices 20](#_Toc207197775)

[2.9 Advanced: Common Table Expressions (CTEs) 21](#_Toc207197776)

[2.10 Exercises 21](#_Toc207197777)

[2.11 Conclusion 21](#_Toc207197778)

[Chapter 3: Joins and Subqueries 22](#_Toc207197779)

[3.1 Introduction to Joins and Subqueries 22](#_Toc207197780)

[Recap of CompanyDB Schema 22](#_Toc207197781)

[3.2 Understanding Joins 23](#_Toc207197782)

[3.2.1 INNER JOIN 23](#_Toc207197783)

[3.2.2 LEFT JOIN (LEFT OUTER JOIN) 23](#_Toc207197784)

[3.2.3 RIGHT JOIN (RIGHT OUTER JOIN) 24](#_Toc207197785)

[3.2.4 FULL OUTER JOIN 24](#_Toc207197786)

[3.2.5 Multiple Joins 25](#_Toc207197787)

[3.2.6 CROSS JOIN 25](#_Toc207197788)

[3.3 Subqueries 26](#_Toc207197789)

[3.3.1 Scalar Subqueries 26](#_Toc207197790)

[3.3.2 Correlated Subqueries 26](#_Toc207197791)

[3.3.3 Subqueries in SELECT 27](#_Toc207197792)

[3.3.4 IN vs EXISTS 27](#_Toc207197793)

[3.4 Performance Considerations 28](#_Toc207197794)

[3.5 Best Practices 28](#_Toc207197795)

[3.6 Advanced: Self Joins 28](#_Toc207197796)

[3.7 Exercises 29](#_Toc207197797)

[3.8 Conclusion 29](#_Toc207197798)

[Chapter 4: Stored Procedures 30](#_Toc207197799)

[4.1 Introduction to Stored Procedures 30](#_Toc207197800)

[4.2 Creating Basic Stored Procedures 30](#_Toc207197801)

[4.3 Procedures with Input Parameters 31](#_Toc207197802)

[4.4 Procedures with Output Parameters 33](#_Toc207197803)

[4.5 Returning Result Sets and Multiple Results 34](#_Toc207197804)

[4.6 Error Handling in Stored Procedures 34](#_Toc207197805)

[4.7 Modifying and Dropping Procedures 36](#_Toc207197806)

[4.9 Debugging and Profiling 37](#_Toc207197807)

[4.10 Security and Permissions 37](#_Toc207197808)

[4.11 Best Practices 37](#_Toc207197809)

[4.12 Advanced: Nested Procedures and Recursion 38](#_Toc207197810)

[4.13 Exercises 38](#_Toc207197811)

[4.14 Conclusion 39](#_Toc207197812)

[Chapter 5: User-Defined Functions 40](#_Toc207197813)

[5.1 Introduction to User-Defined Functions 40](#_Toc207197814)

[5.2 Scalar Functions 41](#_Toc207197815)

[5.4 Multi-Statement Table-Valued Functions 44](#_Toc207197816)

[5.5 Using UDFs in Other Constructs 47](#_Toc207197817)

[5.10 Exercises 49](#_Toc207197818)

[5.11 Conclusion 50](#_Toc207197819)

[Chapter 6: Triggers 51](#_Toc207197820)

[6.1 Introduction to Triggers 51](#_Toc207197821)

[6.2 DML Triggers: AFTER Triggers 52](#_Toc207197822)

[6.4 DDL Triggers 55](#_Toc207197823)

[6.7 Debugging Triggers 59](#_Toc207197824)

[6.9 Exercises 59](#_Toc207197825)

[6.10 Conclusion 60](#_Toc207197826)

[Chapter 7: Indexes and Performance Optimization 61](#_Toc207197827)

[7.1 Introduction to Indexes and Performance 61](#_Toc207197828)

[7.2 Index Types in SQL Server 61](#_Toc207197829)

[7.2.1 Clustered Index 61](#_Toc207197830)

[7.2.2 Nonclustered Index 62](#_Toc207197831)

[7.2.3 Included Columns 62](#_Toc207197832)

[7.2.4 Filtered Index 63](#_Toc207197833)

[7.2.5 Composite Index 63](#_Toc207197834)

[7.3 Analyzing Query Performance 63](#_Toc207197835)

[7.4 Query Optimization Techniques 64](#_Toc207197836)

[7.4.1 Write Sargable Queries 64](#_Toc207197837)

[7.4.2 Avoid SELECT \* 64](#_Toc207197838)

[7.4.3 Use Joins Efficiently 64](#_Toc207197839)

[7.5 Advanced Example: Index for Complex Reporting 65](#_Toc207197840)

[7.6 Index Maintenance 66](#_Toc207197841)

[7.7 Advanced Example: Partitioned Index 67](#_Toc207197842)

[7.8 Best Practices 67](#_Toc207197843)

[7.9 Exercises 67](#_Toc207197844)

[7.10 Conclusion 68](#_Toc207197845)

[Chapter 8: Transactions and Error Handling 69](#_Toc207197846)

[8.1 Introduction to Transactions and Error Handling 69](#_Toc207197847)

[8.2 Transaction Basics 69](#_Toc207197848)

[8.3 Savepoints 71](#_Toc207197849)

[8.5 Deadlocks and Handling 73](#_Toc207197850)

[8.6 Advanced Example: Complex Transaction with Validation 74](#_Toc207197851)

[8.7 Performance Considerations 76](#_Toc207197852)

[8.8 Best Practices 76](#_Toc207197853)

[8.9 Exercises 76](#_Toc207197854)

[8.10 Conclusion 77](#_Toc207197855)

[Chapter 9: Security in SQL Server 78](#_Toc207197856)

[9.1 Introduction to SQL Server Security 78](#_Toc207197857)

[9.2 Authentication 78](#_Toc207197858)

[9.3 Authorization 79](#_Toc207197859)

[9.4 Row-Level Security (RLS) 80](#_Toc207197860)

[9.5 Encryption 81](#_Toc207197861)

[9.6 Auditing 83](#_Toc207197862)

[9.7 Advanced Example: Dynamic Access Control 84](#_Toc207197863)

[9.8 Performance Considerations 85](#_Toc207197864)

[9.9 Best Practices 85](#_Toc207197865)

[9.10 Exercises 86](#_Toc207197866)

[9.11 Conclusion 86](#_Toc207197867)

[Chapter 10: Best Practices and Advanced Topics 87](#_Toc207197868)

[10.1 Introduction to Best Practices and Advanced Topics 87](#_Toc207197869)

[10.2 Best Practices for T-SQL Programming 87](#_Toc207197870)

[10.3 Common Table Expressions (CTEs) 88](#_Toc207197871)

[10.4 Window Functions 90](#_Toc207197872)

[10.9 Best Practices for Advanced Features 95](#_Toc207197873)

[10.10 Exercises 95](#_Toc207197874)

[10.11 Conclusion 95](#_Toc207197875)

[Chapter 11: Capstone Project - Building a Comprehensive Sales Dashboard Backend with AdventureWorks2022 96](#_Toc207197876)

[11.1 Project Overview 96](#_Toc207197877)

[11.2 Setting Up the Environment 97](#_Toc207197878)

[11.3 Basic SQL Queries: Exploring and Manipulating Sales Data 98](#_Toc207197879)

[11.5 Stored Procedures: Web Interface Endpoints 101](#_Toc207197880)

[11.6 User-Defined Functions: Custom Calculations 103](#_Toc207197881)

[11.7 Triggers: Data Integrity and Auditing 104](#_Toc207197882)

[11.8 Indexes and Performance Optimization 105](#_Toc207197883)

[11.11 Advanced Topics: Sophisticated Analytics 107](#_Toc207197884)

[11.12 Conclusion and Web Integration 108](#_Toc207197885)

# Chapter 1: Introduction to SQL Server

## 1.1 Overview of SQL Server

Microsoft SQL Server is a robust relational database management system (RDBMS) designed to handle a wide range of applications, from small-scale projects to enterprise-level systems. It supports structured data storage, querying, and management using Transact-SQL (T-SQL), Microsoft’s extension of the SQL standard. T-SQL adds procedural programming capabilities like variables, loops, and error handling, making it powerful for both simple queries and complex logic.

Key Features

* **Scalability**: Supports databases from a few megabytes to terabytes, with editions like Express (free, 10GB limit) to Enterprise (high availability, advanced analytics).
* **Performance**: Features like in-memory tables, columnstore indexes, and query optimization.
* **Security**: Offers encryption, row-level security, dynamic data masking, and auditing.
* **Integration**: Seamlessly works with .NET, Azure, Power BI, and SSIS (SQL Server Integration Services).
* **Deployment Options**: On-premises, cloud (Azure SQL Database), or hybrid.

SQL Server runs on Windows and Linux, with container support via Docker. Editions include:

* **Express**: Free, limited to 10GB databases, ideal for learning.
* **Standard**: Balanced features for small to medium businesses.
* **Enterprise**: Advanced features like Always On, data warehousing.
* **Developer**: Free, full-featured for non-production use.

## 1.2 SQL Server Architecture

Understanding SQL Server’s architecture helps in writing efficient code:

* **Storage Engine**: Manages data files (.mdf for data, .ldf for logs).
* **Relational Engine**: Processes T-SQL queries, optimizes execution plans.
* **Memory Management**: Uses buffer pools for caching data, reducing disk I/O.
* **Transaction Log**: Ensures data integrity via ACID properties (Atomicity, Consistency, Isolation, Durability).

Key databases:

* **master**: Stores system configuration.
* **msdb**: Manages jobs, backups.
* **tempdb**: Temporary storage for queries, rebuilt on server restart.

## 1.3 Setting Up Your Environment

To start, install SQL Server and a client tool:

1. **Download SQL Server**: Get the free Express or Developer edition from [Microsoft’s website](https://www.microsoft.com/en-us/sql-server/sql-server-downloads).
2. **Install SSMS**: SQL Server Management Studio, available from Microsoft, is the primary GUI for querying.
3. **Alternative**: Use Azure Data Studio for a lightweight, cross-platform tool.
4. **Sample Database**: Download AdventureWorks from [Microsoft’s GitHub](https://github.com/Microsoft/sql-server-samples/releases). Restore it using:

RESTORE DATABASE AdventureWorks

FROM DISK = 'C:\Path\To\AdventureWorks.bak'

WITH MOVE 'AdventureWorks\_Data' TO 'C:\SQLData\AdventureWorks.mdf',

MOVE 'AdventureWorks\_Log' TO 'C:\SQLData\AdventureWorks.ldf';

Connecting to SQL Server

* **Windows Authentication**: Uses your Windows credentials (recommended for local setups).
* **SQL Server Authentication**: Username/password, e.g., sa account (enable during installation).
In SSMS, connect to localhost or your server name (e.g., SERVERNAME\SQLEXPRESS).

## 1.4 Creating Your First Database

Let’s create a database and a table to store employee data.

-- Create a new database

USE master;

GO

CREATE DATABASE CompanyDB;

GO

-- Switch to the new database

USE CompanyDB;

GO

-- Create a table with constraints

CREATE TABLE Employees (

 EmployeeID INT PRIMARY KEY IDENTITY(1,1),

 FirstName NVARCHAR(50) NOT NULL,

 LastName NVARCHAR(50) NOT NULL,

 Email NVARCHAR(100) UNIQUE,

 HireDate DATE NOT NULL,

 Salary DECIMAL(10,2) CHECK (Salary >= 0),

 DepartmentID INT,

 CONSTRAINT CHK\_Email CHECK (Email LIKE '%@%.%')

);

GO

**Explanation**:

* CREATE DATABASE: Initializes a new database.
* IDENTITY(1,1): Auto-increments EmployeeID starting from 1.
* NOT NULL: Ensures fields aren’t empty.
* UNIQUE: Prevents duplicate emails.
* CHECK: Enforces salary >= 0 and valid email format.
* GO: Batch separator for SSMS.

Run this in a new query window in SSMS. Verify creation:

SELECT name FROM sys.databases WHERE name = 'CompanyDB';

SELECT \* FROM sys.tables WHERE name = 'Employees';

## 1.5 Basic T-SQL Syntax

T-SQL is case-insensitive, but consistent casing improves readability. Key conventions:

* Use UPPERCASE for SQL keywords (optional but common).
* Use camelCase or PascalCase for object names.
* End statements with ; (optional in older versions, required in newer contexts).

Inserting Data

Add sample employee records:

INSERT INTO Employees (FirstName, LastName, Email, HireDate, Salary, DepartmentID)

VALUES ('John', 'Doe', 'john.doe@company.com', '2023-01-15', 75000.00, 1),

 ('Jane', 'Smith', 'jane.smith@company.com', '2022-06-20', 82000.00, 2),

 ('Bob', 'Johnson', 'bob.johnson@company.com', '2024-03-10', 65000.00, NULL);

**Explanation**:

* INSERT INTO: Specifies table and columns.
* VALUES: Provides data in matching order.
* NULL for DepartmentID if unassigned.

Verify:

SELECT \* FROM Employees;

**Output**:

| **EmployeeID** | **FirstName** | **LastName** | **Email** | **HireDate** | **Salary** | **DepartmentID** |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | John | Doe | john.doe@company.com | 2023-01-15 | 75000.00 | 1 |
| 2 | Jane | Smith | jane.smith@company.com | 2022-06-20 | 82000.00 | 2 |
| 3 | Bob | Johnson | bob.johnson@company.com | 2024-03-10 | 65000.00 | NULL |

## 1.6 Basic Data Types

SQL Server supports various data types:

* **Numeric**: INT (whole numbers), DECIMAL(p,s) (precision, scale, e.g., 10,2 for 12345678.12), FLOAT.
* **String**: NVARCHAR(n) (Unicode, variable length), CHAR(n) (fixed length).
* **Date/Time**: DATE (YYYY-MM-DD), DATETIME2 (higher precision), TIME.
* **Binary**: VARBINARY for files/images.
* **Special**: UNIQUEIDENTIFIER (GUIDs), XML.

Example: Add a binary column for employee photos.

ALTER TABLE Employees

ADD Photo VARBINARY(MAX) NULL;

**Explanation**: VARBINARY(MAX) allows large binary data (e.g., images). NULL permits empty values.

## 1.7 Constraints and Relationships

Constraints enforce data integrity:

* **PRIMARY KEY**: Uniquely identifies rows, enforces NOT NULL.
* **FOREIGN KEY**: Links tables.
* **CHECK**: Validates data.
* **UNIQUE**: Prevents duplicates.
* **DEFAULT**: Sets default values.

Example: Create a Departments table and link it.

CREATE TABLE Departments (

 DepartmentID INT PRIMARY KEY IDENTITY(1,1),

 DepartmentName NVARCHAR(50) NOT NULL,

 Location NVARCHAR(100) DEFAULT 'Unknown'

);

-- Add foreign key to Employees

ALTER TABLE Employees

ADD CONSTRAINT FK\_Employee\_Dept

FOREIGN KEY (DepartmentID) REFERENCES Departments(DepartmentID);

-- Insert departments

INSERT INTO Departments (DepartmentName, Location)

VALUES ('HR', 'New York'), ('IT', 'San Francisco');

**Explanation**:

* FOREIGN KEY: Ensures DepartmentID in Employees matches a valid DepartmentID.
* DEFAULT: Sets Location to 'Unknown' if unspecified.

Verify relationships:

SELECT e.FirstName, e.LastName, d.DepartmentName

FROM Employees e

LEFT JOIN Departments d ON e.DepartmentID = d.DepartmentID;

**Output**:

| **FirstName** | **LastName** | **DepartmentName** |
| --- | --- | --- |
| John | Doe | HR |
| Jane | Smith | IT |
| Bob | Johnson | NULL |

## 1.8 Basic Querying with SELECT

The SELECT statement retrieves data. Key clauses:

* WHERE: Filters rows.
* ORDER BY: Sorts results.
* TOP: Limits rows.

Example: Query employees with high salaries.

SELECT TOP 2 FirstName, LastName, Salary

FROM Employees

WHERE Salary > 70000

ORDER BY Salary DESC;

**Explanation**:

* TOP 2: Returns only the top 2 rows.
* WHERE: Filters salaries above 70,000.
* ORDER BY DESC: Sorts from highest to lowest.

**Output**:

| **FirstName** | **LastName** | **Salary** |
| --- | --- | --- |
| Jane | Smith | 82000.00 |
| John | Doe | 75000.00 |

## 1.9 Error Handling Basics

Use TRY...CATCH to handle errors gracefully.

Example: Handle duplicate email error.

BEGIN TRY

 INSERT INTO Employees (FirstName, LastName, Email, HireDate, Salary, DepartmentID)

 VALUES ('Alice', 'Brown', 'john.doe@company.com', '2024-05-01', 60000.00, 1);

END TRY

BEGIN CATCH

 SELECT

 ERROR\_MESSAGE() AS ErrorMessage,

 ERROR\_LINE() AS ErrorLine;

END CATCH;

**Explanation**:

* Attempts to insert a duplicate email, violating the UNIQUE constraint.
* CATCH returns the error: "Violation of UNIQUE KEY constraint..."

## 1.10 Best Practices for Beginners

* **Use Meaningful Names**: e.g., Employees not Table1.
* **Comment Code**: Add -- or /\* \*/ for clarity.
* **Backup Before Changes**: Use BACKUP DATABASE CompanyDB TO DISK = 'C:\Backups\CompanyDB.bak';.
* \*\*Avoid SELECT \*\*\* in production; specify columns.
* **Test on Small Data**: Use a dev database before production.

## 1.11 Advanced: Exploring System Views

System views like sys.tables, sys.columns provide metadata.

Example: List all tables and their columns.

SELECT

 t.name AS TableName,

 c.name AS ColumnName,

 ty.name AS DataType

FROM sys.tables t

JOIN sys.columns c ON t.object\_id = c.object\_id

JOIN sys.types ty ON c.system\_type\_id = ty.system\_type\_id

WHERE t.name IN ('Employees', 'Departments');

**Explanation**: Queries system catalog to show schema details.

**1.12 Exercises**

1. Create a database SalesDB with a table Products (ProductID, Name, Price, Stock).
2. Insert 5 products, then query products with stock > 10.
3. Add a foreign key to a new Categories table and join them.
4. Write a query to find the top 3 most expensive products.

**1.13 Conclusion**

This chapter introduced SQL Server’s core concepts, setup, and basic T-SQL programming. You’ve learned to create databases, tables, constraints, and perform basic queries with error handling. The examples are foundational, preparing you for deeper topics like stored procedures and indexing.

# Chapter 2: Basic SQL Queries

## 2.1 Introduction to CRUD Operations

In SQL Server, CRUD operations (Create, Read, Update, Delete) form the backbone of data manipulation using Transact-SQL (T-SQL). This chapter explores these operations in depth, focusing on the INSERT, SELECT, UPDATE, and DELETE statements. We’ll use the CompanyDB database created in Chapter 1, which contains the Employees and Departments tables, and add a new Projects table for more complex examples.

**Recap of CompanyDB Schema**

* **Employees**: EmployeeID (PK), FirstName, LastName, Email, HireDate, Salary, DepartmentID (FK), Photo.
* **Departments**: DepartmentID (PK), DepartmentName, Location.

Let’s add a Projects table to track employee assignments:

USE CompanyDB;

GO

CREATE TABLE Projects (

 ProjectID INT PRIMARY KEY IDENTITY(1,1),

 ProjectName NVARCHAR(100) NOT NULL,

 StartDate DATE NOT NULL,

 EndDate DATE,

 Budget DECIMAL(12,2) CHECK (Budget >= 0),

 DepartmentID INT,

 CONSTRAINT FK\_Project\_Dept FOREIGN KEY (DepartmentID) REFERENCES Departments(DepartmentID),

 CONSTRAINT CHK\_Dates CHECK (EndDate IS NULL OR EndDate >= StartDate)

);

GO

**Explanation**:

* IDENTITY(1,1): Auto-generates ProjectID.
* CHECK (Budget >= 0): Ensures non-negative budgets.
* CHECK (EndDate IS NULL OR EndDate >= StartDate): Validates date logic.
* FOREIGN KEY: Links to Departments.

Insert sample data:

INSERT INTO Projects (ProjectName, StartDate, EndDate, Budget, DepartmentID)

VALUES ('ERP Implementation', '2023-03-01', '2024-02-28', 150000.00, 2),

 ('Recruitment Portal', '2024-01-15', NULL, 50000.00, 1),

 ('Data Migration', '2023-09-01', '2024-06-30', 80000.00, 2);

## 2.2 INSERT: Creating Data

The INSERT statement adds rows to a table. It supports single-row, multi-row, and data-from-query insertions.

**Single-Row Insert**

Example: Add a new employee.

INSERT INTO Employees (FirstName, LastName, Email, HireDate, Salary, DepartmentID)

VALUES ('Alice', 'Brown', 'alice.brown@company.com', '2024-05-01', 60000.00, 1);

**Explanation**:

* Columns are specified explicitly.
* Values match column order and data types.
* EmployeeID is auto-generated by IDENTITY.

**Multi-Row Insert**

Insert multiple employees at once:

INSERT INTO Employees (FirstName, LastName, Email, HireDate, Salary, DepartmentID)

VALUES ('Tom', 'Wilson', 'tom.wilson@company.com', '2023-11-10', 72000.00, 2),

 ('Sarah', 'Davis', 'sarah.davis@company.com', '2024-02-20', 68000.00, 1);

**Explanation**:

* Multiple VALUES tuples separated by commas.
* Efficient for bulk inserts.

**Insert from Query**

Copy projects with high budgets to a new table:

CREATE TABLE MajorProjects (

 ProjectID INT PRIMARY KEY,

 ProjectName NVARCHAR(100),

 Budget DECIMAL(12,2)

);

INSERT INTO MajorProjects (ProjectID, ProjectName, Budget)

SELECT ProjectID, ProjectName, Budget

FROM Projects

WHERE Budget > 75000;

**Explanation**:

* Creates MajorProjects table.
* Inserts projects with budgets over $75,000.
* SELECT acts as the data source.

Verify:

SELECT \* FROM MajorProjects;

**Output**:

| **ProjectID** | **ProjectName** | **Budget** |
| --- | --- | --- |
| 1 | ERP Implementation | 150000.00 |
| 3 | Data Migration | 80000.00 |

**2.3 SELECT: Reading Data**

The SELECT statement retrieves data, with clauses for filtering, sorting, and limiting.

**Basic SELECT**

Retrieve all employee data:

SELECT EmployeeID, FirstName, LastName, Email, HireDate

FROM Employees;

**Explanation**:

* Explicit column names improve performance and clarity over SELECT \*.
* Returns all rows from Employees.

**Filtering with WHERE**

Filter employees by department and salary:

SELECT FirstName, LastName, Salary

FROM Employees

WHERE DepartmentID = 1 AND Salary >= 65000;

**Explanation**:

* WHERE uses AND to combine conditions.
* Returns employees in HR (DepartmentID=1) with salaries ≥ $65,000.

**Output** (assuming prior inserts):

| **FirstName** | **LastName** | **Salary** |
| --- | --- | --- |
| John | Doe | 75000.00 |

**Sorting with ORDER BY**

Sort employees by hire date, newest first:

SELECT FirstName, LastName, HireDate

FROM Employees

ORDER BY HireDate DESC;

**Explanation**:

* DESC: Descending order (newest to oldest).
* Default is ASC (ascending).

**Limiting with TOP**

Get the top 3 highest-paid employees:

SELECT TOP 3 FirstName, LastName, Salary

FROM Employees

ORDER BY Salary DESC;

**Explanation**:

* TOP 3: Limits to 3 rows.
* Sorted by salary, highest first.

**Output**:

| **FirstName** | **LastName** | **Salary** |
| --- | --- | --- |
| Jane | Smith | 82000.00 |
| John | Doe | 75000.00 |
| Tom | Wilson | 72000.00 |

**Using Expressions**

Calculate years of service:

SELECT FirstName, LastName, DATEDIFF(YEAR, HireDate, GETDATE()) AS YearsOfService

FROM Employees

WHERE DATEDIFF(YEAR, HireDate, GETDATE()) > 1;

**Explanation**:

* DATEDIFF(YEAR, HireDate, GETDATE()): Computes years between hire date and now.
* AS: Aliases the calculated column.
* Filters for employees with >1 year of service.

## 2.4 UPDATE: Modifying Data

The UPDATE statement modifies existing rows. Always use WHERE to avoid updating all rows.

Example: Increase salaries in IT (DepartmentID=2) by 10%:

UPDATE Employees

SET Salary = Salary \* 1.10

WHERE DepartmentID = 2;

**Explanation**:

* SET: Updates Salary with a 10% increase.
* WHERE: Targets IT employees.
* Verify with SELECT \* FROM Employees.

**Update with Join**

Update project budgets based on department:

UPDATE p

SET Budget = Budget + 10000

FROM Projects p

JOIN Departments d ON p.DepartmentID = d.DepartmentID

WHERE d.DepartmentName = 'HR';

**Explanation**:

* Joins Projects and Departments.
* Increases budgets for HR projects by $10,000.
* FROM and JOIN allow multi-table updates.

## 2.5 DELETE: Removing Data

The DELETE statement removes rows. Use WHERE to avoid deleting all data.

Example: Delete employees hired before 2023:

DELETE FROM Employees

WHERE HireDate < '2023-01-01';

**Explanation**:

* Removes employees like Jane Smith (hired 2022-06-20).
* Verify with SELECT \* FROM Employees.

**Delete with Join**

Delete completed projects in IT:

DELETE p

FROM Projects p

JOIN Departments d ON p.DepartmentID = d.DepartmentID

WHERE d.DepartmentName = 'IT' AND p.EndDate IS NOT NULL;

**Explanation**:

* Deletes IT projects with an EndDate (e.g., Data Migration).
* Uses join to filter by department.

## 2.6 Aggregation and Grouping

Use aggregate functions (COUNT, SUM, AVG, MIN, MAX) with GROUP BY to summarize data.

**Basic Aggregation**

Count employees per department:

SELECT d.DepartmentName, COUNT(e.EmployeeID) AS EmployeeCount

FROM Departments d

LEFT JOIN Employees e ON d.DepartmentID = e.DepartmentID

GROUP BY d.DepartmentName;

**Explanation**:

* COUNT(e.EmployeeID): Counts non-NULL employee records.
* LEFT JOIN: Includes departments with zero employees.
* GROUP BY: Groups results by department name.

**Output** (assuming current data):

| **DepartmentName** | **EmployeeCount** |
| --- | --- |
| HR | 2 |
| IT | 2 |

**Filtering Groups with HAVING**

List departments with total salaries > $100,000:

SELECT d.DepartmentName, SUM(e.Salary) AS TotalSalary

FROM Departments d

JOIN Employees e ON d.DepartmentID = e.DepartmentID

GROUP BY d.DepartmentName

HAVING SUM(e.Salary) > 100000;

**Explanation**:

* HAVING: Filters grouped results (like WHERE for aggregates).
* Only shows departments with high total salaries.

## 2.7 Handling NULLs

NULL represents missing data. Use IS NULL or IS NOT NULL.

Example: Find employees without a department:

SELECT FirstName, LastName

FROM Employees

WHERE DepartmentID IS NULL;

**Explanation**:

* Returns Bob Johnson (DepartmentID=NULL).

Use COALESCE to handle NULLs:

SELECT ProjectName, COALESCE(EndDate, 'Ongoing') AS Status

FROM Projects;

**Output**:

| **ProjectName** | **Status** |
| --- | --- |
| ERP Implementation | 2024-02-28 |
| Recruitment Portal | Ongoing |
| Data Migration | 2024-06-30 |

## 2.8 Best Practices

* **Specify Columns**: Avoid SELECT \* in production.
* **Use WHERE**: Prevent unintended updates/deletes.
* **Test Queries**: Run SELECT before UPDATE/DELETE.
* **Indexes**: Add indexes on frequently filtered columns (e.g., DepartmentID).
* **Comments**: Use -- or /\* \*/ for clarity.
* **Batch Size**: For large INSERT/UPDATE, use batches to avoid locking.

Example: Batch insert:

-- Insert 1000 employees in batches

SET NOCOUNT ON;

DECLARE @i INT = 1;

WHILE @i <= 1000

BEGIN

 INSERT INTO Employees (FirstName, LastName, Email, HireDate, Salary, DepartmentID)

 VALUES ('User' + CAST(@i AS NVARCHAR(10)), 'Test', 'user' + CAST(@i AS NVARCHAR(10)) + '@test.com', '2024-01-01', 50000.00, 1);

 SET @i = @i + 1;

END;

## 2.9 Advanced: Common Table Expressions (CTEs)

CTEs simplify complex queries.

Example: Find employees with above-average salaries:

WITH AvgSalary AS (

 SELECT AVG(Salary) AS AvgSalary

 FROM Employees

)

SELECT FirstName, LastName, Salary

FROM Employees, AvgSalary

WHERE Salary > AvgSalary.AvgSalary;

**Explanation**:

* WITH: Defines a temporary result set.
* Compares each salary to the average.

## 2.10 Exercises

1. Insert 3 new projects into Projects, including one with a NULL EndDate.
2. Write a SELECT query to find employees hired in 2024, sorted by LastName.
3. Update all projects with budgets < $60,000 to add $5,000.
4. Delete employees with salaries < $65,000 in HR.
5. Write a query to count projects per department, including departments with zero projects.

## 2.11 Conclusion

This chapter covered essential T-SQL queries for manipulating and retrieving data. You’ve learned to insert, select, update, and delete data, handle aggregations, and work with NULLs. These skills are foundational for more advanced topics like joins and stored procedures in later chapters.

# Chapter 3: Joins and Subqueries

## 3.1 Introduction to Joins and Subqueries

Joins and subqueries are essential for working with relational data in SQL Server, allowing you to combine data from multiple tables or perform nested queries. Joins link tables based on related columns, while subqueries enable queries within queries for complex filtering or calculations. This chapter covers the types of joins, subquery patterns, and their use cases, with a focus on Transact-SQL (T-SQL) syntax and performance optimization.

## Recap of CompanyDB Schema

From Chapters 1 and 2:

* **Employees**: EmployeeID (PK), FirstName, LastName, Email, HireDate, Salary, DepartmentID (FK), Photo.
* **Departments**: DepartmentID (PK), DepartmentName, Location.
* **Projects**: ProjectID (PK), ProjectName, StartDate, EndDate, Budget, DepartmentID (FK).
* **MajorProjects**: ProjectID (PK), ProjectName, Budget.

Let’s add a new table to track employee-project assignments:

USE CompanyDB;

GO

CREATE TABLE EmployeeProjects (

 EmployeeID INT,

 ProjectID INT,

 Role NVARCHAR(50) NOT NULL,

 AssignmentDate DATE NOT NULL,

 PRIMARY KEY (EmployeeID, ProjectID),

 FOREIGN KEY (EmployeeID) REFERENCES Employees(EmployeeID),

 FOREIGN KEY (ProjectID) REFERENCES Projects(ProjectID)

);

INSERT INTO EmployeeProjects (EmployeeID, ProjectID, Role, AssignmentDate)

VALUES (1, 1, 'Lead Developer', '2023-03-05'),

 (2, 1, 'Project Manager', '2023-03-01'),

 (3, 2, 'HR Coordinator', '2024-01-20'),

 (4, 3, 'Data Analyst', '2023-09-10');

**Explanation**:

* EmployeeProjects: Links employees to projects with roles and dates.
* Composite PRIMARY KEY on EmployeeID and ProjectID.
* FOREIGN KEY constraints ensure referential integrity.

## 3.2 Understanding Joins

Joins combine rows from two or more tables based on a condition, typically matching keys. SQL Server supports several join types: INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL OUTER JOIN, and CROSS JOIN.

### 3.2.1 INNER JOIN

Returns only matching rows from both tables.

Example: List employees and their departments.

SELECT e.FirstName, e.LastName, d.DepartmentName

FROM Employees e

INNER JOIN Departments d ON e.DepartmentID = d.DepartmentID;

**Explanation**:

* INNER JOIN: Matches DepartmentID between tables.
* Excludes employees with NULL DepartmentID (e.g., Bob Johnson).
* ON: Specifies the join condition.

**Output** (based on Chapter 2 data):

| **FirstName** | **LastName** | **DepartmentName** |
| --- | --- | --- |
| John | Doe | HR |
| Jane | Smith | IT |
| Alice | Brown | HR |
| Sarah | Davis | HR |
| Tom | Wilson | IT |

### 3.2.2 LEFT JOIN (LEFT OUTER JOIN)

Returns all rows from the left table and matching rows from the right. Non-matching right-table rows return NULL.

Example: Include employees without departments.

SELECT e.FirstName, e.LastName, d.DepartmentName

FROM Employees e

LEFT JOIN Departments d ON e.DepartmentID = d.DepartmentID;

**Explanation**:

* LEFT JOIN: Includes all employees, even those with NULL DepartmentID.
* Bob Johnson appears with NULL for DepartmentName.

**Output**:

| **FirstName** | **LastName** | **DepartmentName** |
| --- | --- | --- |
| John | Doe | HR |
| Jane | Smith | IT |
| Alice | Brown | HR |
| Sarah | Davis | HR |
| Tom | Wilson | IT |
| Bob | Johnson | NULL |

### 3.2.3 RIGHT JOIN (RIGHT OUTER JOIN)

Opposite of LEFT JOIN, returns all rows from the right table.

Example: List all departments, even those without employees.

SELECT d.DepartmentName, COUNT(e.EmployeeID) AS EmployeeCount

FROM Departments d

RIGHT JOIN Employees e ON d.DepartmentID = e.DepartmentID

GROUP BY d.DepartmentName;

**Explanation**:

* RIGHT JOIN: Prioritizes Employees, so unused departments may not appear unless flipped.
* Use LEFT JOIN from Departments for empty departments (see Section 3.2.5).

### 3.2.4 FULL OUTER JOIN

Returns all rows from both tables, with NULL for non-matches.

Example: All employees and departments, including non-matches.

SELECT e.FirstName, e.LastName, d.DepartmentName

FROM Employees e

FULL OUTER JOIN Departments d ON e.DepartmentID = d.DepartmentID;

**Explanation**:

* Shows employees without departments and departments without employees.
* Useful for auditing data mismatches.

### 3.2.5 Multiple Joins

Combine multiple tables.

Example: Employees, their projects, and departments.

SELECT e.FirstName, e.LastName, p.ProjectName, d.DepartmentName

FROM Employees e

INNER JOIN EmployeeProjects ep ON e.EmployeeID = ep.EmployeeID

INNER JOIN Projects p ON ep.ProjectID = p.ProjectID

LEFT JOIN Departments d ON p.DepartmentID = d.DepartmentID;

**Explanation**:

* Joins Employees to EmployeeProjects, then to Projects, and finally to Departments.
* LEFT JOIN on Departments ensures projects without departments are included.
* Returns employees’ project roles and department names.

**Output**:

| **FirstName** | **LastName** | **ProjectName** | **DepartmentName** |
| --- | --- | --- | --- |
| John | Doe | ERP Implementation | IT |
| Jane | Smith | ERP Implementation | IT |
| Bob | Johnson | Recruitment Portal | HR |
| Tom | Wilson | Data Migration | IT |

### 3.2.6 CROSS JOIN

Produces a Cartesian product (all row combinations).

Example: Pair every employee with every project (use cautiously).

SELECT e.FirstName, p.ProjectName

FROM Employees e

CROSS JOIN Projects p

WHERE e.EmployeeID = 1; -- Limit for practicality

**Explanation**:

* Generates all combinations, so 5 employees × 3 projects = 15 rows unless filtered.
* Rarely used without WHERE.

## 3.3 Subqueries

Subqueries are nested queries within a main query, used in SELECT, WHERE, or FROM.

### 3.3.1 Scalar Subqueries

Return a single value.

Example: Employees with above-average salaries.

SELECT FirstName, LastName, Salary

FROM Employees

WHERE Salary > (SELECT AVG(Salary) FROM Employees);

**Explanation**:

* Subquery (SELECT AVG(Salary)...) computes the average salary.
* Main query filters employees with higher salaries.
* Subquery runs once.

**Output** (assuming average salary ~$70,167):

| **FirstName** | **LastName** | **Salary** |
| --- | --- | --- |
| John | Doe | 75000.00 |
| Jane | Smith | 82000.00 |
| Tom | Wilson | 72000.00 |

### 3.3.2 Correlated Subqueries

Execute for each row of the main query.

Example: Employees assigned to projects with budgets > $100,000.

SELECT e.FirstName, e.LastName

FROM Employees e

WHERE EXISTS (

 SELECT 1

 FROM EmployeeProjects ep

 JOIN Projects p ON ep.ProjectID = p.ProjectID

 WHERE ep.EmployeeID = e.EmployeeID AND p.Budget > 100000

);

**Explanation**:

* EXISTS: Checks if the subquery returns rows.
* Correlates via ep.EmployeeID = e.EmployeeID.
* Returns John and Jane (on ERP Implementation, budget $150,000).

### 3.3.3 Subqueries in SELECT

Compute values per row.

Example: Show each project’s budget relative to the average.

SELECT ProjectName, Budget,

 Budget - (SELECT AVG(Budget) FROM Projects) AS BudgetVsAvg

FROM Projects;

**Output** (average budget ~$93,333):

| **ProjectName** | **Budget** | **BudgetVsAvg** |
| --- | --- | --- |
| ERP Implementation | 150000SNS0.00 | 56666.67 |
| Recruitment Portal | 50000.00 | -43333.33 |
| Data Migration | 80000.00 | -11333.33 |

### 3.3.4 IN vs EXISTS

IN and EXISTS are common in subqueries.

Example: Use IN for project IDs.

SELECT FirstName, LastName

FROM Employees

WHERE EmployeeID IN (

 SELECT EmployeeID FROM EmployeeProjects WHERE ProjectID = 1

);

**Explanation**:

* IN: Checks if EmployeeID is in the subquery’s result.
* EXISTS is often faster for correlated subqueries.

## 3.4 Performance Considerations

* **Joins vs Subqueries**: Joins are often faster than correlated subqueries due to query optimizer efficiency.
* **Indexes**: Create indexes on join/subquery columns (e.g., DepartmentID, EmployeeID).
* **Execution Plans**: Use SET STATISTICS IO ON and SSMS’s “Show Actual Execution Plan” to analyze performance.
* **Avoid Overuse of Subqueries**: Rewrite as joins where possible.

Example: Index for joins.

CREATE NONCLUSTERED INDEX IX\_EmployeeProjects\_EmployeeID

ON EmployeeProjects(EmployeeID);

CREATE NONCLUSTERED INDEX IX\_EmployeeProjects\_ProjectID

ON EmployeeProjects(ProjectID);

**Explanation**:

* Speeds up joins on EmployeeProjects.

## 3.5 Best Practices

* **Use Explicit Joins**: Prefer INNER JOIN over implicit joins (e.g., FROM A, B WHERE A.id = B.id).
* **Choose the Right Join**: Use INNER JOIN for matches, LEFT JOIN for optional matches.
* **Simplify Subqueries**: Avoid deeply nested subqueries; use CTEs or joins.
* **Test Performance**: Run queries on large datasets to identify bottlenecks.
* **Use Aliases**: Shorten table names (e.g., Employees e) for readability.

## 3.6 Advanced: Self Joins

Join a table to itself.

Example: Find employees hired after their department’s first hire.

SELECT e1.FirstName, e1.LastName, e1.HireDate

FROM Employees e1

JOIN Employees e2 ON e1.DepartmentID = e2.DepartmentID

WHERE e1.HireDate > e2.HireDate

AND e2.HireDate = (

 SELECT MIN(HireDate)

 FROM Employees e3

 WHERE e3.DepartmentID = e1.DepartmentID

);

**Explanation**:

* Self-join compares employees within the same department.
* Subquery finds the earliest hire date per department.

## 3.7 Exercises

1. Write an INNER JOIN query to list employees and their project roles.
2. Use a LEFT JOIN to find projects with no assigned employees.
3. Write a subquery to find departments with no employees.
4. Create a correlated subquery to list employees on high-budget projects (> $80,000).
5. Optimize a join query with an index and check the execution plan.

## 3.8 Conclusion

Joins and subqueries are powerful tools for combining and filtering data. Joins offer flexibility in merging tables, while subqueries enable complex logic within queries. Understanding their syntax, performance implications, and best practices is crucial for efficient SQL Server programming.

# Chapter 4: Stored Procedures

## 4.1 Introduction to Stored Procedures

Stored procedures are precompiled collections of one or more T-SQL statements stored in the database, allowing for reusable, modular, and secure code execution in SQL Server. They enhance performance by reducing network traffic (only parameters are sent, not full queries), improve security by encapsulating logic and controlling access, and promote maintainability through centralized code. Stored procedures can accept input parameters, return output parameters, return result sets, and handle errors.

This chapter explores creating, executing, modifying, and optimizing stored procedures, with advanced topics like dynamic SQL and debugging. We’ll use the CompanyDB database from previous chapters, assuming the schema with Employees, Departments, Projects, and EmployeeProjects tables.

**Benefits of Stored Procedures**

* **Performance**: Compiled once, execution plans are cached.
* **Security**: Grant EXECUTE permissions without exposing table access.
* **Modularity**: Encapsulate business logic, reducing code duplication.
* **Error Handling**: Built-in TRY/CATCH for robust operations.
* **Transaction Management**: Control commits and rollbacks.

**When to Use**

* For complex, multi-statement operations (e.g., insert with validation).
* Reusable queries with parameters to prevent SQL injection.
* Batch operations that need to be atomic.

## 4.2 Creating Basic Stored Procedures

Use CREATE PROCEDURE (or CREATE PROC) to define a procedure. It can include any valid T-SQL.

**Parameterless Procedure**

Example: A simple procedure to list all employees.

USE CompanyDB;

GO

CREATE PROCEDURE ListAllEmployees

AS

BEGIN

 SELECT EmployeeID, FirstName, LastName, HireDate, Salary

 FROM Employees

 ORDER BY LastName;

END;

GO

**Explanation**:

* AS BEGIN ... END: Defines the body.
* No parameters; just retrieves and sorts employees.
* GO: Separates batches in SSMS.

Execute it:

EXEC ListAllEmployees;

**Output** (sample data):

| **EmployeeID** | **FirstName** | **LastName** | **HireDate** | **Salary** |
| --- | --- | --- | --- | --- |
| 4 | Alice | Brown | 2024-05-01 | 60000.00 |
| 1 | John | Doe | 2023-01-15 | 75000.00 |
| 3 | Bob | Johnson | 2024-03-10 | 65000.00 |
| 2 | Jane | Smith | 2022-06-20 | 82000.00 |
| 5 | Sarah | Davis | 2024-02-20 | 68000.00 |
| 6 | Tom | Wilson | 2023-11-10 | 72000.00 |

## 4.3 Procedures with Input Parameters

Parameters are prefixed with @ and typed. They can be input (default) or output.

**Single Input Parameter**

Example: Get employee details by ID.

CREATE PROCEDURE GetEmployeeByID

 @EmployeeID INT

AS

BEGIN

 SELECT FirstName, LastName, Email, HireDate, Salary, DepartmentID

 FROM Employees

 WHERE EmployeeID = @EmployeeID;

END;

GO

**Explanation**:

* @EmployeeID INT: Input parameter.
* Used in WHERE to filter.

Execute:

EXEC GetEmployeeByID @EmployeeID = 1;

**Output**:

| **FirstName** | **LastName** | **Email** | **HireDate** | **Salary** | **DepartmentID** |
| --- | --- | --- | --- | --- | --- |
| John | Doe | john.doe@company.com | 2023-01-15 | 75000.00 | 1 |

**Multiple Input Parameters with Defaults**

Example: Search employees by department and minimum salary.

CREATE PROCEDURE SearchEmployees

 @DepartmentID INT = NULL,

 @MinSalary DECIMAL(10,2) = 0

AS

BEGIN

 SELECT e.FirstName, e.LastName, d.DepartmentName, e.Salary

 FROM Employees e

 LEFT JOIN Departments d ON e.DepartmentID = d.DepartmentID

 WHERE (@DepartmentID IS NULL OR e.DepartmentID = @DepartmentID)

 AND e.Salary >= @MinSalary

 ORDER BY e.Salary DESC;

END;

GO

**Explanation**:

* Defaults: @DepartmentID = NULL (all departments), @MinSalary = 0 (no minimum).
* WHERE handles optional parameters with OR and IS NULL.
* Joins for department name.

Execute examples:

* All employees: EXEC SearchEmployees;
* IT employees with salary >= $70,000: EXEC SearchEmployees @DepartmentID = 2, @MinSalary = 70000;

## 4.4 Procedures with Output Parameters

Output parameters return values to the caller.

Example: Calculate average salary and return it.

CREATE PROCEDURE GetAverageSalary

 @AvgSalary DECIMAL(10,2) OUTPUT

AS

BEGIN

 SELECT @AvgSalary = AVG(Salary)

 FROM Employees;

END;

GO

**Explanation**:

* @AvgSalary OUTPUT: Returns the computed average.

Execute and capture:

DECLARE @Average DECIMAL(10,2);

EXEC GetAverageSalary @AvgSalary = @Average OUTPUT;

PRINT 'Average Salary: ' + CAST(@Average AS NVARCHAR(20));

**Output**: Average Salary: 70333.33 (based on sample data).

**Combining Input and Output**

Example: Get employee count by department.

CREATE PROCEDURE GetEmployeeCountByDept

 @DepartmentID INT,

 @EmployeeCount INT OUTPUT

AS

BEGIN

 SELECT @EmployeeCount = COUNT(\*)

 FROM Employees

 WHERE DepartmentID = @DepartmentID;

END;

GO

Execute:

DECLARE @Count INT;

EXEC GetEmployeeCountByDept @DepartmentID = 1, @EmployeeCount = @Count OUTPUT;

PRINT 'HR Employees: ' + CAST(@Count AS NVARCHAR(10));

## 4.5 Returning Result Sets and Multiple Results

Procedures can return multiple result sets.

Example: Get employees and departments.

CREATE PROCEDURE GetCompanyData

AS

BEGIN

 SELECT \* FROM Employees ORDER BY LastName;

 SELECT \* FROM Departments ORDER BY DepartmentName;

END;

GO

**Explanation**:

* Two SELECT statements return separate result sets.

Execute:

EXEC GetCompanyData; – SSMS shows two grids.

## 4.6 Error Handling in Stored Procedures

Use TRY...CATCH for robust error management.

Example: Insert employee with error handling.

CREATE PROCEDURE InsertEmployee

 @FirstName NVARCHAR(50),

 @LastName NVARCHAR(50),

 @Email NVARCHAR(100),

 @HireDate DATE,

 @Salary DECIMAL(10,2),

 @DepartmentID INT

AS

BEGIN

 BEGIN TRY

 BEGIN TRANSACTION;

 INSERT INTO Employees (FirstName, LastName, Email, HireDate, Salary, DepartmentID)

 VALUES (@FirstName, @LastName, @Email, @HireDate, @Salary, @DepartmentID);

 COMMIT TRANSACTION;

 END TRY

 BEGIN CATCH

 IF @@TRANCOUNT > 0

 ROLLBACK TRANSACTION;

 SELECT

 ERROR\_NUMBER() AS ErrorNumber,

 ERROR\_MESSAGE() AS ErrorMessage;

 END CATCH;

END;

GO

**Explanation**:

* BEGIN TRANSACTION: Starts a transaction.
* TRY: Attempts insert.
* CATCH: Rolls back on error (e.g., duplicate email) and returns error details.
* Prevents partial updates.

Execute (success):

EXEC InsertEmployee 'Michael', 'Scott', 'michael.scott@company.com', '2024-08-01', 90000.00, 1;

Execute (error, duplicate email):

EXEC InsertEmployee 'John', 'Doe', 'john.doe@company.com', '2023-01-15', 75000.00, 1;

**Output on Error**:

| **ErrorNumber** | **ErrorMessage** |
| --- | --- |
| 2627 | Violation of UNIQUE KEY constraint... |

## 4.7 Modifying and Dropping Procedures

Alter existing procedures with ALTER PROCEDURE.

Example: Add a column to ListAllEmployees.

ALTER PROCEDURE ListAllEmployees

AS

BEGIN

 SELECT EmployeeID, FirstName, LastName, HireDate, Salary, Email

 FROM Employees

 ORDER BY LastName;

END;

GO

Drop:

DROP PROCEDURE IF EXISTS ListAllEmployees;

**Explanation**:

* IF EXISTS: Prevents errors if not present.

**4.8 Dynamic SQL in Procedures**

Dynamic SQL builds and executes strings at runtime, useful for flexible queries.

Example: Dynamic search.

CREATE PROCEDURE DynamicEmployeeSearch

 @SearchColumn NVARCHAR(50),

 @SearchValue NVARCHAR(100)

AS

BEGIN

 DECLARE @SQL NVARCHAR(MAX);

 SET @SQL = N'SELECT \* FROM Employees WHERE ' + QUOTENAME(@SearchColumn) + N' = @Value';

 EXEC sp\_executesql @SQL, N'@Value NVARCHAR(100)', @Value = @SearchValue;

END;

GO

**Explanation**:

* Builds @SQL dynamically.
* QUOTENAME: Prevents SQL injection on column names.
* sp\_executesql: Executes with parameters, safer than EXEC.

Execute:

EXEC DynamicEmployeeSearch @SearchColumn = 'LastName', @SearchValue = 'Doe';

## 4.9 Debugging and Profiling

* **SSMS Debugger**: Set breakpoints, step through code (View > SQL Server Object Explorer > Procedures > Right-click > Debug Procedure).
* **PRINT**: Insert for variable values.
* **PROFILER**: Use SQL Server Profiler to trace execution.
* **Extended Events**: For advanced monitoring.

Example: Add PRINT for debugging.

ALTER PROCEDURE InsertEmployee

 -- Parameters...

AS

BEGIN

 PRINT 'Starting Insert...';

 -- Rest of code

END;

## 4.10 Security and Permissions

Grant EXECUTE:

GRANT EXECUTE ON InsertEmployee TO SomeUser;

**Explanation**:

* Users can run the proc without table INSERT permissions.

## 4.11 Best Practices

* **Parameterize Everything**: Avoid hardcoding to prevent injection.
* **Use SCHEMABINDING**: For performance (CREATE PROC WITH SCHEMABINDING).
* **Limit Dynamic SQL**: Use only when necessary; prefer static.
* **Document**: Add comments with purpose, parameters, history.
* **Version Control**: Use ALTER, keep backups.
* **Test Thoroughly**: Cover edge cases, errors, performance.
* **Naming**: Prefix like usp\_ (user stored proc) for clarity.

## 4.12 Advanced: Nested Procedures and Recursion

Procedures can call others.

Example: Nested call.

CREATE PROCEDURE MainProc

AS

BEGIN

 EXEC ListAllEmployees;

 EXEC GetAverageSalary @AvgSalary = @SomeVar OUTPUT; -- Declare @SomeVar first

END;

Recursion: Limited to 32 levels.

CREATE PROCEDURE RecursiveProc @Level INT

AS

BEGIN

 IF @Level > 0

 BEGIN

 PRINT @Level;

 EXEC RecursiveProc @Level - 1;

 END;

END;

GO

EXEC RecursiveProc 5;

## 4.13 Exercises

1. Create a procedure to update an employee's salary by ID, with input parameters for ID and new salary.
2. Build a procedure with output parameter to return the total budget for a department.
3. Implement error handling in a procedure that deletes an employee, ensuring no open projects.
4. Write a dynamic SQL procedure to filter projects by any column.
5. Debug a procedure using PRINT and test with invalid data.

## 4.14 Conclusion

Stored procedures are a cornerstone of SQL Server programming, offering efficiency, security, and modularity. You've learned to create, parameterize, handle errors, and optimize them. These concepts prepare you for functions and triggers in upcoming chapters.

# Chapter 5: User-Defined Functions

## 5.1 Introduction to User-Defined Functions

User-defined functions (UDFs) in SQL Server allow you to encapsulate reusable T-SQL logic, returning scalar values or tables. Unlike stored procedures (Chapter 4), UDFs can be embedded in queries, computed columns, or constraints, making them versatile for calculations and data transformations. They are ideal for modularizing complex logic, improving readability, and ensuring consistency across queries.

**Types of UDFs**

* **Scalar Functions**: Return a single value (e.g., INT, DECIMAL).
* **Inline Table-Valued Functions (TVFs)**: Return a table result set, defined by a single SELECT statement.
* **Multi-Statement Table-Valued Functions (MSTVFs)**: Return a table, built using multiple T-SQL statements.

**Benefits**

* **Reusability**: Call functions in SELECT, WHERE, or JOIN clauses.
* **Consistency**: Centralize logic (e.g., formatting rules).
* **Modularity**: Simplify complex queries.

**Limitations**

* Cannot modify data (e.g., INSERT, UPDATE).
* Limited error handling compared to stored procedures.
* Performance overhead for scalar UDFs in large datasets (mitigated with SCHEMABINDING).

We’ll use the CompanyDB schema:

* **Employees**: EmployeeID (PK), FirstName, LastName, Email, HireDate, Salary, DepartmentID (FK), Photo.
* **Departments**: DepartmentID (PK), DepartmentName, Location.
* **Projects**: ProjectID (PK), ProjectName, StartDate, EndDate, Budget, DepartmentID (FK).
* **EmployeeProjects**: EmployeeID, ProjectID, Role, AssignmentDate.

## 5.2 Scalar Functions

Scalar functions return a single value and are used in SELECT, WHERE, or computed columns.

**Basic Scalar Function**

Example: Calculate years of service for an employee.

USE CompanyDB;

GO

CREATE FUNCTION dbo.CalculateYearsOfService (@HireDate DATE)

RETURNS INT

AS

BEGIN

 RETURN DATEDIFF(YEAR, @HireDate, GETDATE());

END;

GO

**Explanation**:

* @HireDate: Input parameter.
* RETURNS INT: Specifies return type.
* DATEDIFF: Computes years between hire date and current date.

Use in a query:

SELECT FirstName, LastName, dbo.CalculateYearsOfService(HireDate) AS YearsOfService

FROM Employees

WHERE dbo.CalculateYearsOfService(HireDate) >= 2;

**Output** (assuming today is 2025-08-27):

| **FirstName** | **LastName** | **YearsOfService** |
| --- | --- | --- |
| Jane | Smith | 3 |
| John | Doe | 2 |

**Advanced Scalar Function: Salary Bonus Calculation**

Example: Calculate a performance-based bonus with tiered rates.

CREATE FUNCTION dbo.CalculateBonus (

 @Salary DECIMAL(10,2),

 @PerformanceRating INT

)

RETURNS DECIMAL(10,2)

WITH SCHEMABINDING

AS

BEGIN

 DECLARE @BonusRate DECIMAL(4,2);

 SET @BonusRate = CASE

 WHEN @PerformanceRating >= 8 THEN 0.15

 WHEN @PerformanceRating >= 5 THEN 0.10

 ELSE 0.05

 END;

 RETURN @Salary \* @BonusRate;

END;

GO

**Explanation**:

* @PerformanceRating: 1–10 scale.
* CASE: Sets bonus rate (15% for 8+, 10% for 5–7, 5% otherwise).
* WITH SCHEMABINDING: Improves performance by binding to schema, preventing table changes.
* Returns bonus amount.

Use:

SELECT FirstName, LastName, Salary,

 dbo.CalculateBonus(Salary, 8) AS Bonus

FROM Employees

WHERE DepartmentID = 1;

**Output** (HR employees, rating 8):

| **FirstName** | **LastName** | **Salary** | **Bonus** |
| --- | --- | --- | --- |
| John | Doe | 75000.00 | 11250.00 |
| Alice | Brown | 60000.00 | 9000.00 |
| Sarah | Davis | 68000.00 | 10200.00 |

**5.3 Inline Table-Valued Functions**

Inline TVFs return a table result via a single SELECT statement, optimized like views.

**Basic Inline TVF**

Example: Get employees by department.

CREATE FUNCTION dbo.GetEmployeesByDept (@DepartmentID INT)

RETURNS TABLE

AS

RETURN

 SELECT e.EmployeeID, e.FirstName, e.LastName, e.Salary, d.DepartmentName

 FROM Employees e

 INNER JOIN Departments d ON e.DepartmentID = d.DepartmentID

 WHERE e.DepartmentID = @DepartmentID;

GO

**Explanation**:

* RETURNS TABLE: Specifies table output.
* RETURN: Single SELECT statement.
* Joins Employees and Departments.

Use:

SELECT \* FROM dbo.GetEmployeesByDept(2);

**Output** (IT employees):

| **EmployeeID** | **FirstName** | **LastName** | **Salary** | **DepartmentName** |
| --- | --- | --- | --- | --- |
| 2 | Jane | Smith | 82000.00 | IT |
| 6 | Tom | Wilson | 72000.00 | IT |

**Advanced Inline TVF: Project Team Details**

Example: Get project team members with roles and project details.

CREATE FUNCTION dbo.GetProjectTeam (@ProjectID INT)

RETURNS TABLE

AS

RETURN

 SELECT

 e.FirstName,

 e.LastName,

 ep.Role,

 ep.AssignmentDate,

 p.ProjectName,

 p.Budget

 FROM EmployeeProjects ep

 INNER JOIN Employees e ON ep.EmployeeID = e.EmployeeID

 INNER JOIN Projects p ON ep.ProjectID = p.ProjectID

 WHERE p.ProjectID = @ProjectID

 AND p.StartDate <= GETDATE()

 AND (p.EndDate IS NULL OR p.EndDate >= GETDATE());

GO

**Explanation**:

* Filters for active projects (started, not ended).
* Joins three tables to get team details.
* Efficient due to inline execution (no temporary tables).

Use:

SELECT \* FROM dbo.GetProjectTeam(2);

**Output** (Recruitment Portal team):

| **FirstName** | **LastName** | **Role** | **AssignmentDate** | **ProjectName** | **Budget** |
| --- | --- | --- | --- | --- | --- |
| Bob | Johnson | HR Coordinator | 2024-01-20 | Recruitment Portal | 50000.00 |

## 5.4 Multi-Statement Table-Valued Functions

MSTVFs define a table variable, populated with multiple statements.

**Basic MSTVF**

Example: Return employees with salary ranges.

CREATE FUNCTION dbo.GetEmployeesBySalaryRange (

 @MinSalary DECIMAL(10,2),

 @MaxSalary DECIMAL(10,2)

)

RETURNS @Result TABLE (

 EmployeeID INT,

 FullName NVARCHAR(101),

 Salary DECIMAL(10,2)

)

AS

BEGIN

 INSERT INTO @Result (EmployeeID, FullName, Salary)

 SELECT EmployeeID,

 FirstName + ' ' + LastName AS FullName,

 Salary

 FROM Employees

 WHERE Salary BETWEEN @MinSalary AND @MaxSalary;

 RETURN;

END;

GO

**Explanation**:

* @Result: Table variable with defined columns.
* Populates with filtered employees.
* Concatenates names for FullName.

Use:

SELECT \* FROM dbo.GetEmployeesBySalaryRange(60000, 75000);

**Output**:

| **EmployeeID** | **FullName** | **Salary** |
| --- | --- | --- |
| 1 | John Doe | 75000.00 |
| 3 | Bob Johnson | 65000.00 |
| 4 | Alice Brown | 60000.00 |
| 5 | Sarah Davis | 68000.00 |

**Advanced MSTVF: Employee Project Summary**

Example: Summarize employee project involvement with hours worked.

CREATE FUNCTION dbo.GetEmployeeProjectSummary (@EmployeeID INT)

RETURNS @Summary TABLE (

 EmployeeID INT,

 FullName NVARCHAR(101),

 ProjectCount INT,

 TotalEstimatedHours INT,

 HighestBudgetProject NVARCHAR(100)

)

AS

BEGIN

 DECLARE @TotalHours INT;

 -- Calculate estimated hours based on project duration

 INSERT INTO @Summary (EmployeeID, FullName, ProjectCount, TotalEstimatedHours, HighestBudgetProject)

 SELECT

 e.EmployeeID,

 e.FirstName + ' ' + e.LastName,

 COUNT(ep.ProjectID) AS ProjectCount,

 SUM(DATEDIFF(DAY, p.StartDate, ISNULL(p.EndDate, GETDATE())) \* 8) AS TotalEstimatedHours,

 MAX(p.ProjectName) AS HighestBudgetProject

 FROM Employees e

 LEFT JOIN EmployeeProjects ep ON e.EmployeeID = ep.EmployeeID

 LEFT JOIN Projects p ON ep.ProjectID = p.ProjectID

 WHERE e.EmployeeID = @EmployeeID

 GROUP BY e.EmployeeID, e.FirstName, e.LastName

 HAVING COUNT(ep.ProjectID) > 0;

 RETURN;

END;

GO

**Explanation**:

* Estimates hours (days × 8 hours/day).
* Aggregates project count and max project name.
* HAVING: Ensures only employees with projects are included.

Use:

SELECT \* FROM dbo.GetEmployeeProjectSummary(1);

**Output** (John Doe on ERP Implementation):

| **EmployeeID** | **FullName** | **ProjectCount** | **TotalEstimatedHours** | **HighestBudgetProject** |
| --- | --- | --- | --- | --- |
| 1 | John Doe | 1 | 7296 | ERP Implementation |

## 5.5 Using UDFs in Other Constructs

**In Computed Columns**

Add a computed column for years of service.

ALTER TABLE Employees ADD YearsOfService AS dbo.CalculateYearsOfService(HireDate);

Query:

SELECT FirstName, LastName, YearsOfService FROM Employees;

**In CHECK Constraints**

Ensure new employees have valid hire dates.

ALTER TABLE Employees

ADD CONSTRAINT CHK\_HireDate CHECK (dbo.CalculateYearsOfService(HireDate) >= 0);

**In Joins**

Join with TVF:

SELECT t.FullName, t.Salary, d.DepartmentName

FROM dbo.GetEmployeesBySalaryRange(60000, 80000) t

JOIN Departments d ON t.EmployeeID IN (

 SELECT EmployeeID FROM Employees WHERE DepartmentID = d.DepartmentID

);

**5.6 Performance Considerations**

* **Scalar UDFs**: Avoid in large datasets; they execute row-by-row, causing overhead. Use WITH SCHEMABINDING or rewrite as inline TVFs.
* **Inline TVFs**: Optimized like views, generally faster than MSTVFs.
* **MSTVFs**: Use sparingly; table variables don’t use statistics, impacting performance.
* **Indexes**: Cannot index UDF results directly, but consider indexing underlying tables.
* **Execution Plans**: Use SET STATISTICS IO ON and SSMS’s execution plan to optimize.

Example: Optimize join performance.

CREATE NONCLUSTERED INDEX IX\_Employees\_Salary ON Employees(Salary);

**5.7 Debugging and Error Handling**

UDFs have limited error handling (no TRY/CATCH). Use RETURN for early exit.

Example: Safe scalar function.

CREATE FUNCTION dbo.SafeDivide (@Numerator INT, @Denominator INT)

RETURNS DECIMAL(10,2)

AS

BEGIN

 IF @Denominator = 0

 RETURN NULL;

 RETURN CAST(@Numerator AS DECIMAL(10,2)) / @Denominator;

END;

GO

Use:

SELECT dbo.SafeDivide(10, 0) AS Result; -- Returns NULL

**5.8 Best Practices**

* **Use SCHEMABINDING**: For performance and schema consistency.
* **Prefix with Schema**: dbo.FunctionName for clarity.
* **Avoid Side Effects**: UDFs should not modify data.
* **Keep Simple**: Complex logic may be better in stored procedures.
* **Document**: Comment parameters, purpose, and return types.
* **Test Edge Cases**: NULLs, empty tables, extreme values.

**5.9 Advanced Example: Dynamic Aggregation Function**

Create an MSTVF to aggregate project budgets by custom ranges.

CREATE FUNCTION dbo.GetProjectBudgetRanges (@RangeSize DECIMAL(10,2))

RETURNS @Result TABLE (

 BudgetRange NVARCHAR(50),

 ProjectCount INT,

 TotalBudget DECIMAL(12,2)

)

AS

BEGIN

 INSERT INTO @Result (BudgetRange, ProjectCount, TotalBudget)

 SELECT

 CAST(FLOOR(Budget / @RangeSize) \* @RangeSize AS NVARCHAR(50)) + '-' +

 CAST((FLOOR(Budget / @RangeSize) + 1) \* @RangeSize AS NVARCHAR(50)) AS BudgetRange,

 COUNT(\*) AS ProjectCount,

 SUM(Budget) AS TotalBudget

 FROM Projects

 GROUP BY FLOOR(Budget / @RangeSize);

 RETURN;

END;

GO

Use:

SELECT \* FROM dbo.GetProjectBudgetRanges(50000);

**Output**:

| **BudgetRange** | **ProjectCount** | **TotalBudget** |
| --- | --- | --- |
| 0-50000 | 1 | 50000.00 |
| 50000-100000 | 1 | 80000.00 |
| 100000-150000 | 1 | 150000.00 |

**Explanation**:

* Groups projects by budget ranges (e.g., $0–50,000).
* Dynamic FLOOR calculation for flexible ranges.

## 5.10 Exercises

1. Create a scalar function to format employee names (e.g., “LastName, FirstName”).
2. Build an inline TVF to list projects by department with active status.
3. Create an MSTVF to return employees with their total project hours and roles.
4. Add a computed column using a scalar function for employee tenure in months.
5. Optimize a TVF with an index and compare execution plans.

## 5.11 Conclusion

User-defined functions provide powerful ways to encapsulate logic, from simple calculations to complex table results. Scalar functions are great for reusable calculations, while inline TVFs offer performance for table results. MSTVFs provide flexibility for complex logic. Mastering UDFs enhances query modularity and prepares you for triggers in Chapter 6.

# Chapter 6: Triggers

## 6.1 Introduction to Triggers

Triggers in SQL Server are special stored procedures that automatically execute in response to specific database events, such as data modifications (INSERT, UPDATE, DELETE) or database structure changes (CREATE, ALTER, DROP). They are powerful for enforcing business rules, maintaining audit logs, or preventing invalid data changes, but they must be used carefully to avoid performance issues or unintended side effects.

**Types of Triggers**

* **DML Triggers**: Respond to data manipulation language (DML) events (INSERT, UPDATE, DELETE).
	+ **AFTER**: Execute after the event completes.
	+ **INSTEAD OF**: Execute instead of the event, allowing custom logic.
* **DDL Triggers**: Respond to data definition language (DDL) events (e.g., table creation).
* **Logon Triggers**: Respond to server logon events (e.g., for auditing).

**Benefits**

* **Automation**: Enforce rules or log changes without application intervention.
* **Data Integrity**: Validate or transform data before saving.
* **Audit Trails**: Track changes for compliance or debugging.

**Limitations**

* Can impact performance if complex or poorly designed.
* Limited error handling compared to stored procedures.
* Recursive triggers can cause infinite loops if not managed.

We’ll use the CompanyDB schema:

* **Employees**: EmployeeID (PK), FirstName, LastName, Email, HireDate, Salary, DepartmentID (FK), Photo, YearsOfService.
* **Departments**: DepartmentID (PK), DepartmentName, Location.
* **Projects**: ProjectID (PK), ProjectName, StartDate, EndDate, Budget, DepartmentID (FK).
* **EmployeeProjects**: EmployeeID, ProjectID, Role, AssignmentDate.

## 6.2 DML Triggers: AFTER Triggers

AFTER triggers execute after the triggering event (INSERT, UPDATE, DELETE) completes but before the transaction commits. They access special tables: inserted (new data) and deleted (old data).

**Basic AFTER Trigger**

Example: Log employee inserts to an audit table.

USE CompanyDB;

GO

CREATE TABLE EmployeeAudit (

 AuditID INT IDENTITY(1,1) PRIMARY KEY,

 EmployeeID INT,

 Action NVARCHAR(50),

 ActionDate DATETIME,

 OldData NVARCHAR(MAX),

 NewData NVARCHAR(MAX)

);

CREATE TRIGGER trg\_AfterInsertEmployee

ON Employees

AFTER INSERT

AS

BEGIN

 INSERT INTO EmployeeAudit (EmployeeID, Action, ActionDate, NewData)

 SELECT

 i.EmployeeID,

 'INSERT',

 GETDATE(),

 (SELECT \* FROM inserted FOR XML RAW)

 FROM inserted i;

END;

GO

**Explanation**:

* EmployeeAudit: Stores audit data.
* trg\_AfterInsertEmployee: Fires after INSERT on Employees.
* inserted: Magic table with new rows.
* FOR XML RAW: Converts row to XML for flexible logging.

Test:

INSERT INTO Employees (FirstName, LastName, Email, HireDate, Salary, DepartmentID)

VALUES ('Pam', 'Beesly', 'pam.beesly@company.com', '2024-09-01', 62000.00, 1);

SELECT \* FROM EmployeeAudit;

**Output**:

| **AuditID** | **EmployeeID** | **Action** | **ActionDate** | **OldData** | **NewData** |
| --- | --- | --- | --- | --- | --- |
| 1 | 7 | INSERT | 2025-08-27 09:28:00 | NULL | <row EmployeeID="7" FirstName="Pam" LastName="Beesly" ... /> |

**Advanced AFTER Trigger: Salary Validation**

Example: Prevent salary decreases and log updates.

CREATE TRIGGER trg\_AfterUpdateEmployee

ON Employees

AFTER UPDATE

AS

BEGIN

 SET NOCOUNT ON;

 IF EXISTS (

 SELECT 1

 FROM inserted i

 JOIN deleted d ON i.EmployeeID = d.EmployeeID

 WHERE i.Salary < d.Salary

 )

 BEGIN

 RAISERROR ('Salary cannot be decreased.', 16, 1);

 ROLLBACK TRANSACTION;

 RETURN;

 END;

 INSERT INTO EmployeeAudit (EmployeeID, Action, ActionDate, OldData, NewData)

 SELECT

 i.EmployeeID,

 'UPDATE',

 GETDATE(),

 (SELECT \* FROM deleted WHERE EmployeeID = i.EmployeeID FOR XML RAW),

 (SELECT \* FROM inserted WHERE EmployeeID = i.EmployeeID FOR XML RAW)

 FROM inserted i;

END;

GO

**Explanation**:

* Checks if any new salary (inserted) is less than the old salary (deleted).
* RAISERROR: Raises an error and rolls back if invalid.
* Logs successful updates to EmployeeAudit.

Test:

sql

*-- Valid update*

UPDATE Employees SET Salary = 78000.00 WHERE EmployeeID = 1;

*-- Invalid update (will fail)*

UPDATE Employees SET Salary = 70000.00 WHERE EmployeeID = 1;

**Error Output**:

text

Msg 50000, Level 16, State 1

Salary cannot be decreased.

**6.3 DML Triggers: INSTEAD OF Triggers**

INSTEAD OF triggers override the triggering event, allowing custom logic.

**Basic INSTEAD OF Trigger**

Example: Prevent deletion of employees on active projects.

CREATE TRIGGER trg\_InsteadOfDeleteEmployee

ON Employees

INSTEAD OF DELETE

AS

BEGIN

 SET NOCOUNT ON;

 IF EXISTS (

 SELECT 1

 FROM deleted d

 JOIN EmployeeProjects ep ON d.EmployeeID = ep.EmployeeID

 JOIN Projects p ON ep.ProjectID = p.ProjectID

 WHERE p.EndDate IS NULL OR p.EndDate > GETDATE()

 )

 BEGIN

 RAISERROR ('Cannot delete employees assigned to active projects.', 16, 1);

 RETURN;

 END;

 DELETE FROM Employees WHERE EmployeeID IN (SELECT EmployeeID FROM deleted);

END;

GO

**Explanation**:

* Checks if deleted employees are on active projects.
* If so, raises an error; otherwise, performs the delete.

Test:

-- Should fail (John is on ERP Implementation)

DELETE FROM Employees WHERE EmployeeID = 1;

-- Should succeed (assuming Bob’s project is inactive)

DELETE FROM Employees WHERE EmployeeID = 3;

## 6.4 DDL Triggers

DDL triggers respond to schema changes (e.g., CREATE TABLE).

**Basic DDL Trigger**

Example: Log table creations.

CREATE TABLE SchemaAudit (

 AuditID INT IDENTITY(1,1) PRIMARY KEY,

 EventType NVARCHAR(100),

 ObjectName NVARCHAR(128),

 EventDate DATETIME,

 UserName NVARCHAR(128)

);

CREATE TRIGGER trg\_DDLTableCreate

ON DATABASE

FOR CREATE\_TABLE

AS

BEGIN

 INSERT INTO SchemaAudit (EventType, ObjectName, EventDate, UserName)

 SELECT

 EVENTDATA().value('(/EVENT\_INSTANCE/EventType)[1]', 'NVARCHAR(100)'),

 EVENTDATA().value('(/EVENT\_INSTANCE/ObjectName)[1]', 'NVARCHAR(128)'),

 GETDATE(),

 SUSER\_SNAME();

END;

GO

**Explanation**:

* EVENTDATA(): XML containing event details.
* Logs table creation events with user and timestamp.

Test:

CREATE TABLE TestTable (ID INT);

SELECT \* FROM SchemaAudit;

**Output**:

| **AuditID** | **EventType** | **ObjectName** | **EventDate** | **UserName** |
| --- | --- | --- | --- | --- |
| 1 | CREATE\_TABLE | TestTable | 2025-08-27 09:28:00 | YourUser |

**Advanced DDL Trigger: Prevent Table Drops**

Example: Block dropping critical tables.

sql

CREATE TRIGGER trg\_PreventTableDrop

ON DATABASE

FOR DROP\_TABLE

AS

BEGIN

 DECLARE @TableName NVARCHAR(128) = EVENTDATA().value('(/EVENT\_INSTANCE/ObjectName)[1]', 'NVARCHAR(128)');

 IF @TableName IN ('Employees', 'Departments', 'Projects')

 BEGIN

 RAISERROR ('Cannot drop critical tables.', 16, 1);

 ROLLBACK;

 END;

END;

GO

**Explanation**:

* Checks if the dropped table is critical.
* Rolls back if so.

Test:

DROP TABLE Employees; *-- Fails*

DROP TABLE TestTable; *-- Succeeds*

**6.5 Advanced Example: Cascading Updates**

Example: Update project assignments when an employee’s department changes.

CREATE TRIGGER trg\_AfterUpdateEmployeeDept

ON Employees

AFTER UPDATE

AS

BEGIN

 SET NOCOUNT ON;

 IF UPDATE(DepartmentID)

 BEGIN

 UPDATE ep

 SET ep.Role = 'Reassigned - ' + ep.Role

 FROM EmployeeProjects ep

 JOIN inserted i ON ep.EmployeeID = i.EmployeeID

 JOIN deleted d ON i.EmployeeID = d.EmployeeID

 WHERE i.DepartmentID <> d.DepartmentID

 AND EXISTS (

 SELECT 1

 FROM Projects p

 WHERE p.ProjectID = ep.ProjectID

 AND p.DepartmentID = i.DepartmentID

 );

 INSERT INTO EmployeeAudit (EmployeeID, Action, ActionDate, OldData, NewData)

 SELECT

 i.EmployeeID,

 'DEPT\_CHANGE',

 GETDATE(),

 (SELECT \* FROM deleted WHERE EmployeeID = i.EmployeeID FOR XML RAW),

 (SELECT \* FROM inserted WHERE EmployeeID = i.EmployeeID FOR XML RAW)

 FROM inserted i

 JOIN deleted d ON i.EmployeeID = d.EmployeeID

 WHERE i.DepartmentID <> d.DepartmentID;

 END;

END;

GO

**Explanation**:

* Fires on UPDATE of DepartmentID.
* Updates Role in EmployeeProjects if the project’s department matches the new department.
* Logs the change.

Test:

UPDATE Employees SET DepartmentID = 2 WHERE EmployeeID = 4;

SELECT \* FROM EmployeeProjects WHERE EmployeeID = 4;

SELECT \* FROM EmployeeAudit WHERE EmployeeID = 4;

**6.6 Performance Considerations**

* **Minimize Logic**: Keep triggers lightweight to avoid locking.
* **Indexes**: Ensure indexes on joined columns (e.g., EmployeeID).
* **Avoid Recursion**: Use ALTER DATABASE CompanyDB SET RECURSIVE\_TRIGGERS OFF; if needed.
* **Test Impact**: Measure with SET STATISTICS IO ON and execution plans.
* **Batch Operations**: Triggers fire per statement, so batch updates can slow performance.

Example: Index for performance.

sql

CREATE NONCLUSTERED INDEX IX\_EmployeeProjects\_EmployeeID ON EmployeeProjects(EmployeeID);

## 6.7 Debugging Triggers

* **Log to Table**: Use audit tables (e.g., EmployeeAudit).
* **PRINT**: Limited use in triggers; prefer logging.
* **Extended Events**: Trace trigger execution.

Example: Add debug logging “‘PRINT’ style”.

ALTER TRIGGER trg\_AfterInsertEmployee

ON Employees

AFTER INSERT

AS

BEGIN

 INSERT INTO EmployeeAudit (EmployeeID, Action, ActionDate, NewData)

 SELECT

 i.EmployeeID,

 'INSERT (Debug: ' + CAST(@@ROWCOUNT AS NVARCHAR(10)) + ' rows)',

 GETDATE(),

 (SELECT \* FROM inserted FOR XML RAW)

 FROM inserted i;

END;

GO

6.8 Best Practices

* **Keep Simple**: Avoid complex logic; use stored procedures for heavy processing.
* **Use SET NOCOUNT ON**: Prevent rowcount messages from interfering.
* **Name Clearly**: Prefix like trg\_ (e.g., trg\_AfterInsertEmployee).
* **Document**: Comment purpose and logic.
* **Test Extensively**: Cover edge cases, multi-row operations.
* **Disable When Needed**: ALTER TABLE Employees DISABLE TRIGGER trg\_AfterInsertEmployee;.

## 6.9 Exercises

1. Create an AFTER INSERT trigger to log project creations.
2. Build an INSTEAD OF UPDATE trigger to prevent budget decreases in Projects.
3. Create a DDL trigger to log all index creations.
4. Implement a trigger to update EmployeeProjects roles when a project’s department changes.
5. Test a trigger with a bulk insert and optimize with an index.

## 6.10 Conclusion

Triggers automate data integrity, auditing, and business rules, but require careful design to avoid performance issues. You’ve learned to create AFTER and INSTEAD OF DML triggers, DDL triggers, and advanced scenarios, setting the stage for indexing and optimization in Chapter 7.

# Chapter 7: Indexes and Performance Optimization

## 7.1 Introduction to Indexes and Performance

Indexes in SQL Server are database structures that improve query performance by allowing faster data retrieval, similar to an index in a book. However, they come with trade-offs, such as increased storage and slower write operations (INSERT, UPDATE, DELETE). Performance optimization involves creating effective indexes, writing efficient queries, and using tools to analyze and tune performance. This chapter explores index types, their maintenance, and query optimization strategies to ensure your SQL Server database performs efficiently.

**Why Indexes Matter**

* **Speed Up Reads**: Indexes reduce the number of data pages scanned for SELECT queries.
* **Trade-Offs**: Slow down writes due to index updates.
* **Use Cases**: Frequent WHERE, JOIN, ORDER BY, or GROUP BY columns benefit most.

**Why Optimize Queries**

* Poorly written queries can cause full table scans, excessive I/O, or locking.
* Optimized queries reduce resource usage and improve response times.

We’ll use the CompanyDB schema:

* **Employees**: EmployeeID (PK), FirstName, LastName, Email, HireDate, Salary, DepartmentID (FK), Photo, YearsOfService.
* **Departments**: DepartmentID (PK), DepartmentName, Location.
* **Projects**: ProjectID (PK), ProjectName, StartDate, EndDate, Budget, DepartmentID (FK).
* **EmployeeProjects**: EmployeeID, ProjectID, Role, AssignmentDate.
* **EmployeeAudit**: AuditID (PK), EmployeeID, Action, ActionDate, OldData, NewData.

## 7.2 Index Types in SQL Server

SQL Server supports several index types, each suited for specific scenarios.

### 7.2.1 Clustered Index

* Defines the physical order of data in a table (one per table).
* Typically created on the primary key automatically.

Example: The Employees table has a clustered index on EmployeeID (from PRIMARY KEY).

-- Verify clustered index

SELECT name, type\_desc

FROM sys.indexes

WHERE object\_id = OBJECT\_ID('Employees') AND is\_primary\_key = 1;

**Output**:

| **name** | **type\_desc** |
| --- | --- |
| PK\_\_Employees\_\_... | CLUSTERED |

### 7.2.2 Nonclustered Index

* Separate structure with pointers to table data, like a book index.
* Can have multiple per table.

Example: Create a nonclustered index on Salary for range queries.

USE CompanyDB;

GO

CREATE NONCLUSTERED INDEX IX\_Employees\_Salary

ON Employees(Salary);

**Explanation**:

* Speeds up queries like SELECT \* FROM Employees WHERE Salary > 70000.
* Stores Salary values and pointers to rows.

### 7.2.3 Included Columns

Include non-key columns to cover queries without accessing the table.

Example: Index for employee searches by department and salary.

CREATE NONCLUSTERED INDEX IX\_Employees\_Dept\_Salary

ON Employees(DepartmentID, Salary)

INCLUDE (FirstName, LastName);

**Explanation**:

* DepartmentID, Salary: Key columns for sorting/filtering.
* INCLUDE (FirstName, LastName): Stores these columns in the index, avoiding table lookups.
* Covers queries like SELECT FirstName, LastName FROM Employees WHERE DepartmentID = 1 AND Salary > 60000.

### 7.2.4 Filtered Index

Indexes a subset of rows, reducing size and maintenance.

Example: Index active projects only.

CREATE NONCLUSTERED INDEX IX\_Projects\_Active

ON Projects(ProjectName)

WHERE EndDate IS NULL OR EndDate > GETDATE();

**Explanation**:

* Only indexes projects that are active, reducing index size.
* Useful for queries like SELECT ProjectName FROM Projects WHERE EndDate IS NULL.

### 7.2.5 Composite Index

Indexes multiple columns for complex queries.

Example: Index for employee-project lookups.

CREATE NONCLUSTERED INDEX IX\_EmployeeProjects\_Composite

ON EmployeeProjects(EmployeeID, ProjectID, AssignmentDate);

**Explanation**:

* Supports queries filtering on EmployeeID, ProjectID, or sorting by AssignmentDate.

## 7.3 Analyzing Query Performance

Use tools to identify bottlenecks:

* **Execution Plans**: Graphical or XML plans showing how SQL Server executes queries.
* **SET STATISTICS IO ON**: Reports I/O usage.
* **Dynamic Management Views (DMVs)**: Track performance metrics.

**Execution Plan Example**

Query without index:

SET STATISTICS IO ON;

SELECT FirstName, LastName FROM Employees WHERE Salary > 70000;

**IO Output** (before index):

Table 'Employees'. Scan count 1, logical reads 50, ...

Add index:

CREATE NONCLUSTERED INDEX IX\_Employees\_Salary ON Employees(Salary);

Run again:
**IO Output** (after index):

Table 'Employees'. Scan count 1, logical reads 10, ...

**Explanation**:

* Reduced reads indicate index seek instead of table scan.
* View plan in SSMS (Query > Display Estimated Execution Plan).

## 7.4 Query Optimization Techniques

### 7.4.1 Write Sargable Queries

Sargable (Search ARGument ABLE) queries leverage indexes.

Non-sargable:

SELECT \* FROM Employees WHERE YEAR(HireDate) = 2023;

Sargable:

SELECT \* FROM Employees WHERE HireDate >= '2023-01-01' AND HireDate < '2024-01-01';

**Explanation**:

* Functions like YEAR prevent index usage.
* Range conditions use indexes effectively.

### 7.4.2 Avoid SELECT \*

Specify columns to reduce I/O and leverage covering indexes.

Example:

-- Instead of SELECT \*

SELECT EmployeeID, FirstName, LastName FROM Employees WHERE DepartmentID = 1;

### 7.4.3 Use Joins Efficiently

Example: Optimize a join-heavy query.

SELECT e.FirstName, e.LastName, p.ProjectName, d.DepartmentName

FROM Employees e

INNER JOIN EmployeeProjects ep ON e.EmployeeID = ep.EmployeeID

INNER JOIN Projects p ON ep.ProjectID = p.ProjectID

INNER JOIN Departments d ON e.DepartmentID = d.DepartmentID

WHERE e.Salary > 70000

AND p.EndDate IS NULL;

**Optimization**:

* Ensure indexes on EmployeeID, ProjectID, DepartmentID, Salary.
* Use INNER JOIN for strict matches to reduce rows early.

## 7.5 Advanced Example: Index for Complex Reporting

Create a report for project budgets by department with employee counts.

Query:

SELECT

 d.DepartmentName,

 COUNT(DISTINCT e.EmployeeID) AS EmployeeCount,

 SUM(p.Budget) AS TotalBudget

FROM Departments d

LEFT JOIN Projects p ON d.DepartmentID = p.DepartmentID

LEFT JOIN EmployeeProjects ep ON p.ProjectID = ep.ProjectID

LEFT JOIN Employees e ON ep.EmployeeID = e.EmployeeID

GROUP BY d.DepartmentName;

**Optimization**:

1. Create indexes:

CREATE NONCLUSTERED INDEX IX\_Projects\_Dept ON Projects(DepartmentID) INCLUDE (Budget, ProjectID);

CREATE NONCLUSTERED INDEX IX\_EmployeeProjects\_ProjEmp ON EmployeeProjects(ProjectID, EmployeeID);

1. Check execution plan in SSMS.
2. Use DMV to find missing indexes:

SELECT

 migs.avg\_total\_user\_cost \* migs.avg\_user\_impact \* (migs.user\_seeks + migs.user\_scans) AS improvement\_measure,

 mid.statement,

 mid.equality\_columns,

 mid.inequality\_columns,

 mid.included\_columns

FROM sys.dm\_db\_missing\_index\_groups mig

JOIN sys.dm\_db\_missing\_index\_group\_stats migs ON mig.index\_group\_handle = migs.group\_handle

JOIN sys.dm\_db\_missing\_index\_details mid ON mig.index\_handle = mid.index\_handle

WHERE mid.database\_id = DB\_ID('CompanyDB');

**Explanation**:

* Indexes cover join and GROUP BY columns.
* DMV suggests additional indexes if needed.

## 7.6 Index Maintenance

Indexes require maintenance to stay effective.

**Rebuild Indexes**

Fragmentation slows queries.

ALTER INDEX IX\_Employees\_Salary ON Employees REBUILD;

**Reorganize Indexes**

Less resource-intensive than rebuild.

ALTER INDEX IX\_Employees\_Salary ON Employees REORGANIZE;

**Check Fragmentation**

SELECT

 index\_name = i.name,

 avg\_fragmentation\_in\_percent

FROM sys.dm\_db\_index\_physical\_stats(DB\_ID('CompanyDB'), OBJECT\_ID('Employees'), NULL, NULL, 'LIMITED') s

JOIN sys.indexes i ON s.object\_id = i.object\_id AND s.index\_id = i.index\_id;

**Explanation**:

* Rebuild if fragmentation > 30%; reorganize if 5–30%.

## 7.7 Advanced Example: Partitioned Index

For large tables, partition indexes to improve manageability.

Example: Partition EmployeeAudit by year.

CREATE PARTITION FUNCTION PF\_AuditYear (DATETIME)

AS RANGE RIGHT FOR VALUES ('2024-01-01', '2025-01-01');

CREATE PARTITION SCHEME PS\_AuditYear

AS PARTITION PF\_AuditYear TO ([PRIMARY], [PRIMARY], [PRIMARY]);

ALTER TABLE EmployeeAudit DROP CONSTRAINT PK\_EmployeeAudit;

ALTER TABLE EmployeeAudit ADD CONSTRAINT PK\_EmployeeAudit PRIMARY KEY NONCLUSTERED (AuditID);

CREATE CLUSTERED INDEX CIX\_EmployeeAudit\_ActionDate

ON EmployeeAudit(ActionDate)

ON PS\_AuditYear(ActionDate);

**Explanation**:

* Partitions EmployeeAudit by ActionDate year.
* Moves clustered index to partition scheme.
* Improves query performance for date-based audits.

## 7.8 Best Practices

* **Index Selectively**: Only on frequently queried columns.
* **Monitor Usage**: Use sys.dm\_db\_index\_usage\_stats to find unused indexes.
* **Avoid Over-Indexing**: Increases write overhead.
* **Test Changes**: Measure before/after with execution plans.
* **Use Covering Indexes**: Include frequently selected columns.
* **Regular Maintenance**: Schedule rebuild/reorganize jobs.

## 7.9 Exercises

1. Create a nonclustered index for HireDate on Employees and test a range query.
2. Write a query with a missing index and use DMV to suggest improvements.
3. Rebuild all indexes on Projects and check fragmentation.
4. Create a filtered index for high-budget projects (> $100,000).
5. Optimize a join query across all tables and verify with execution plan.

## 7.10 Conclusion

Indexes and query optimization are critical for efficient SQL Server performance. You’ve learned to create and manage clustered, nonclustered, and specialized indexes, optimize queries, and use tools like execution plans and DMVs. These skills prepare you for transactions and error handling in Chapter 8.

# Chapter 8: Transactions and Error Handling

## 8.1 Introduction to Transactions and Error Handling

Transactions in SQL Server ensure data integrity by grouping operations into a single, atomic unit of work. They follow the **ACID** properties: **Atomicity** (all or nothing), **Consistency** (valid state), **Isolation** (concurrent transactions don’t interfere), and **Durability** (committed changes are permanent). Error handling, using constructs like TRY...CATCH, ensures robust responses to failures, such as constraint violations or deadlocks.

This chapter explores transaction management, error handling techniques, isolation levels, and advanced scenarios like savepoints and deadlock resolution, building on the CompanyDB schema:

* **Employees**: EmployeeID (PK), FirstName, LastName, Email, HireDate, Salary, DepartmentID (FK), Photo, YearsOfService.
* **Departments**: DepartmentID (PK), DepartmentName, Location.
* **Projects**: ProjectID (PK), ProjectName, StartDate, EndDate, Budget, DepartmentID (FK).
* **EmployeeProjects**: EmployeeID, ProjectID, Role, AssignmentDate.
* **EmployeeAudit**: AuditID (PK), EmployeeID, Action, ActionDate, OldData, NewData.

## 8.2 Transaction Basics

A transaction groups T-SQL statements, ensuring they either all succeed or all fail.

**Explicit Transactions**

Use BEGIN TRANSACTION, COMMIT TRANSACTION, and ROLLBACK TRANSACTION.

Example: Transfer an employee to a new department with audit logging.

USE CompanyDB;

GO

BEGIN TRANSACTION;

BEGIN TRY

 -- Update employee department

 UPDATE Employees

 SET DepartmentID = 2

 WHERE EmployeeID = 4;

 -- Log to audit table

 INSERT INTO EmployeeAudit (EmployeeID, Action, ActionDate, OldData, NewData)

 VALUES (4, 'DEPT\_CHANGE', GETDATE(),

 (SELECT \* FROM Employees WHERE EmployeeID = 4 FOR XML RAW),

 (SELECT \* FROM Employees WHERE EmployeeID = 4 FOR XML RAW));

 COMMIT TRANSACTION;

END TRY

BEGIN CATCH

 IF @@TRANCOUNT > 0

 ROLLBACK TRANSACTION;

 SELECT

 ERROR\_NUMBER() AS ErrorNumber,

 ERROR\_MESSAGE() AS ErrorMessage;

END CATCH;

**Explanation**:

* BEGIN TRANSACTION: Starts the transaction.
* TRY...CATCH: Wraps operations for error handling.
* COMMIT: Saves changes if successful.
* ROLLBACK: Reverts changes on error.
* @@TRANCOUNT: Tracks open transactions.

Test with an error (invalid DepartmentID):

BEGIN TRANSACTION;

BEGIN TRY

 UPDATE Employees

 SET DepartmentID = 999 -- Non-existent

 WHERE EmployeeID = 4;

 COMMIT TRANSACTION;

END TRY

BEGIN CATCH

 IF @@TRANCOUNT > 0

 ROLLBACK TRANSACTION;

 SELECT

 ERROR\_NUMBER() AS ErrorNumber,

 ERROR\_MESSAGE() AS ErrorMessage;

END CATCH;

**Output**:

| **ErrorNumber** | **ErrorMessage** |
| --- | --- |
| 547 | The UPDATE statement conflicted with the FOREIGN KEY constraint... |

## 8.3 Savepoints

Savepoints allow partial rollbacks within a transaction.

Example: Assign an employee to a project with a savepoint.

BEGIN TRANSACTION;

BEGIN TRY

 SAVE TRANSACTION AssignStart;

 INSERT INTO EmployeeProjects (EmployeeID, ProjectID, Role, AssignmentDate)

 VALUES (5, 2, 'Tester', '2025-09-01');

 -- Simulate an error

 INSERT INTO EmployeeProjects (EmployeeID, ProjectID, Role, AssignmentDate)

 VALUES (5, 999, 'Invalid', '2025-09-01'); -- Non-existent ProjectID

 COMMIT TRANSACTION;

END TRY

BEGIN CATCH

 IF @@TRANCOUNT > 0

 BEGIN

 ROLLBACK TRANSACTION AssignStart;

 -- Log error but allow partial commit

 INSERT INTO EmployeeAudit (EmployeeID, Action, ActionDate, OldData)

 VALUES (5, 'FAILED\_ASSIGNMENT', GETDATE(), ERROR\_MESSAGE());

 COMMIT TRANSACTION;

 END;

END CATCH;

SELECT \* FROM EmployeeProjects WHERE EmployeeID = 5;

SELECT \* FROM EmployeeAudit WHERE EmployeeID = 5;

**Explanation**:

* SAVE TRANSACTION: Marks a point to roll back to.
* First insert succeeds; second fails.
* Rolls back to AssignStart, keeping the first insert.
* Logs error and commits.

**8.4 Transaction Isolation Levels**

Isolation levels control how transactions see uncommitted changes, balancing consistency and concurrency.

**Common Isolation Levels**

* **READ UNCOMMITTED**: Allows dirty reads (uncommitted data).
* **READ COMMITTED** (default): Prevents dirty reads but allows non-repeatable reads.
* **REPEATABLE READ**: Prevents non-repeatable reads but allows phantom reads.
* **SERIALIZABLE**: Strictest, prevents all concurrency issues but reduces performance.
* **SNAPSHOT**: Uses row versioning for consistent reads without locking.

Example: Set isolation level to avoid dirty reads.

SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

BEGIN TRANSACTION;

SELECT FirstName, LastName, Salary

FROM Employees WHERE EmployeeID = 1;

COMMIT;

**Advanced: Snapshot Isolation**

Enable snapshot isolation for consistent reads.

ALTER DATABASE CompanyDB SET ALLOW\_SNAPSHOT\_ISOLATION ON;

SET TRANSACTION ISOLATION LEVEL SNAPSHOT;

BEGIN TRANSACTION;

SELECT \* FROM Projects WHERE Budget > 100000;

-- Other sessions can update without blocking

COMMIT;

**Explanation**:

* Snapshot isolation uses row versioning, reducing locks.
* Enable at database level first.

## 8.5 Deadlocks and Handling

Deadlocks occur when transactions block each other, causing SQL Server to terminate one.

**Simulate a Deadlock**

Session 1:

BEGIN TRANSACTION;

UPDATE Employees SET Salary = 80000 WHERE EmployeeID = 1;

WAITFOR DELAY '00:00:05';

UPDATE Projects SET Budget = 160000 WHERE ProjectID = 1;

COMMIT;

Session 2 (run simultaneously):

BEGIN TRANSACTION;

UPDATE Projects SET Budget = 170000 WHERE ProjectID = 1;

WAITFOR DELAY '00:00:05';

UPDATE Employees SET Salary = 81000 WHERE EmployeeID = 1;

COMMIT;

**Explanation**:

* Session 1 locks Employees, waits for Projects.
* Session 2 locks Projects, waits for Employees.
* SQL Server kills one transaction (victim) with error 1205.

**Handle Deadlocks**

Example: Retry on deadlock.

DECLARE @Retry INT = 3;

DECLARE @Attempt INT = 1;

WHILE @Attempt <= @Retry

BEGIN

 BEGIN TRY

 BEGIN TRANSACTION;

 UPDATE Employees SET Salary = 80000 WHERE EmployeeID = 1;

 WAITFOR DELAY '00:00:02';

 UPDATE Projects SET Budget = 160000 WHERE ProjectID = 1;

 COMMIT TRANSACTION;

 BREAK;

 END TRY

 BEGIN CATCH

 IF ERROR\_NUMBER() = 1205 -- Deadlock

 BEGIN

 SET @Attempt = @Attempt + 1;

 IF @Attempt <= @Retry

 BEGIN

 WAITFOR DELAY '00:00:01';

 CONTINUE;

 END

 ELSE

 THROW;

 END

 ELSE

 THROW;

 END CATCH;

END;

**Explanation**:

* Retries up to 3 times on deadlock (error 1205).
* THROW: Re-raises non-deadlock errors.

## 8.6 Advanced Example: Complex Transaction with Validation

Example: Transfer budget between projects with validation and logging.

CREATE PROCEDURE TransferProjectBudget

 @FromProjectID INT,

 @ToProjectID INT,

 @Amount DECIMAL(12,2)

AS

BEGIN

 SET NOCOUNT ON;

 BEGIN TRY

 BEGIN TRANSACTION;

 -- Validate inputs

 IF NOT EXISTS (SELECT 1 FROM Projects WHERE ProjectID = @FromProjectID AND Budget >= @Amount)

 THROW 50001, 'Insufficient budget in source project.', 1;

 IF NOT EXISTS (SELECT 1 FROM Projects WHERE ProjectID = @ToProjectID)

 THROW 50002, 'Target project does not exist.', 1;

 -- Update budgets

 UPDATE Projects

 SET Budget = Budget - @Amount

 WHERE ProjectID = @FromProjectID;

 UPDATE Projects

 SET Budget = Budget + @Amount

 WHERE ProjectID = @ToProjectID;

 -- Log transaction

 INSERT INTO EmployeeAudit (EmployeeID, Action, ActionDate, OldData)

 VALUES (NULL, 'BUDGET\_TRANSFER', GETDATE(),

 'From ProjectID: ' + CAST(@FromProjectID AS NVARCHAR(10)) +

 ', To ProjectID: ' + CAST(@ToProjectID AS NVARCHAR(10)) +

 ', Amount: ' + CAST(@Amount AS NVARCHAR(20)));

 COMMIT TRANSACTION;

 END TRY

 BEGIN CATCH

 IF @@TRANCOUNT > 0

 ROLLBACK TRANSACTION;

 INSERT INTO EmployeeAudit (EmployeeID, Action, ActionDate, OldData)

 VALUES (NULL, 'TRANSFER\_FAILED', GETDATE(), ERROR\_MESSAGE());

 THROW;

 END CATCH;

END;

GO

Test:

EXEC TransferProjectBudget @FromProjectID = 1, @ToProjectID = 2, @Amount = 20000;

SELECT \* FROM Projects WHERE ProjectID IN (1, 2);

SELECT \* FROM EmployeeAudit WHERE Action LIKE 'BUDGET%';

**Explanation**:

* Validates source project has enough budget.
* Updates both projects atomically.
* Logs success or failure.

## 8.7 Performance Considerations

* **Minimize Transaction Scope**: Keep transactions short to reduce locking.
* **Use Appropriate Isolation**: Avoid SERIALIZABLE unless necessary.
* **Monitor Deadlocks**: Use sys.dm\_tran\_locks or Extended Events.
* **Batch Operations**: Process large updates in chunks to avoid escalation to table locks.

Example: Batch update.

DECLARE @BatchSize INT = 1000;

WHILE EXISTS (SELECT 1 FROM Employees WHERE Salary < 65000)

BEGIN

 BEGIN TRANSACTION;

 UPDATE TOP (@BatchSize) Employees

 SET Salary = Salary \* 1.05

 WHERE Salary < 65000;

 COMMIT;

END;

GO

## 8.8 Best Practices

* **Always Use TRY/CATCH**: Handle errors gracefully.
* **Check @@TRANCOUNT**: Ensure proper rollback.
* **Log Errors**: Use audit tables for debugging.
* **Test Deadlocks**: Simulate concurrent transactions.
* **Document**: Comment transaction logic and error cases.
* **Use Savepoints**: For partial rollbacks in complex transactions.

## 8.9 Exercises

1. Write a transaction to insert an employee and assign them to a project atomically.
2. Create a procedure with a savepoint to update employee salary and log, rolling back only the salary change on error.
3. Test a deadlock scenario and implement retry logic.
4. Use SNAPSHOT isolation for a read-heavy query and compare performance.
5. Build a procedure to transfer employees between departments with validation and logging.

## 8.10 Conclusion

Transactions and error handling ensure data integrity and robustness in SQL Server. You’ve learned to manage transactions, use save points, handle errors, and mitigate deadlocks. These skills prepare you for advanced security topics in Chapter 9.

# Chapter 9: Security in SQL Server

## 9.1 Introduction to SQL Server Security

Security in SQL Server is critical to protect sensitive data, ensure compliance, and prevent unauthorized access. SQL Server provides a robust security model encompassing **authentication** (who can access the system), **authorization** (what they can do), **encryption** (protecting data), and **auditing** (tracking actions). This chapter explores these components, focusing on practical implementation and advanced techniques like row-level security and transparent data encryption, using the CompanyDB schema:

* **Employees**: EmployeeID (PK), FirstName, LastName, Email, HireDate, Salary, DepartmentID (FK), Photo, YearsOfService.
* **Departments**: DepartmentID (PK), DepartmentName, Location.
* **Projects**: ProjectID (PK), ProjectName, StartDate, EndDate, Budget, DepartmentID (FK).
* **EmployeeProjects**: EmployeeID, ProjectID, Role, AssignmentDate.
* **EmployeeAudit**: AuditID (PK), EmployeeID, Action, ActionDate, OldData, NewData.

**Security Goals**

* **Confidentiality**: Protect sensitive data (e.g., salaries).
* **Integrity**: Prevent unauthorized changes.
* **Availability**: Ensure access for legitimate users.
* **Auditability**: Track actions for compliance.

## 9.2 Authentication

Authentication verifies user identity. SQL Server supports two modes:

* **Windows Authentication**: Uses Windows credentials (preferred for integrated security).
* **SQL Server Authentication**: Uses username/password stored in SQL Server.

**Creating a SQL Server Login**

Example: Create a login for an application user.

USE master;

GO

CREATE LOGIN AppUser WITH PASSWORD = 'Str0ngP@ssw0rd!',

 CHECK\_EXPIRATION = ON, CHECK\_POLICY = ON;

**Explanation**:

* CHECK\_EXPIRATION: Enforces password expiration.
* CHECK\_POLICY: Applies Windows password policies (e.g., complexity).

**Creating a Database User**

Map the login to a database user.

USE CompanyDB;

GO

CREATE USER AppUser FOR LOGIN AppUser;

**Explanation**:

* Links AppUser login to CompanyDB.

## 9.3 Authorization

Authorization defines what users can do, using roles and permissions.

**Granting Permissions**

Example: Grant SELECT on Employees to AppUser.

GRANT SELECT ON Employees TO AppUser;

**Explanation**:

* Allows AppUser to read Employees data.

**Database Roles**

Use fixed or user-defined roles for grouped permissions.

Example: Create a role for HR staff.

CREATE ROLE HRRole;

GRANT SELECT, UPDATE ON Employees TO HRRole;

GRANT SELECT ON Departments TO HRRole;

ALTER ROLE HRRole ADD MEMBER AppUser;

**Explanation**:

* HRRole: Groups permissions for HR tasks.
* ADD MEMBER: Assigns AppUser to the role.

**Deny and Revoke**

* **DENY**: Explicitly blocks access.
* **REVOKE**: Removes granted permissions.

Example: Deny access to sensitive columns.

DENY SELECT ON Employees(Salary, Photo) TO AppUser;

**Explanation**:

* Prevents AppUser from viewing Salary or Photo.

## 9.4 Row-Level Security (RLS)

RLS restricts data access at the row level based on user attributes.

**Implementing RLS**

Example: Restrict employees to see only their department’s data.

-- Create a function to filter rows

CREATE FUNCTION dbo.fn\_RestrictDepartmentAccess (@DepartmentID INT)

RETURNS TABLE

WITH SCHEMABINDING

AS

RETURN

 SELECT 1 AS Allowed

 WHERE @DepartmentID = (

 SELECT DepartmentID

 FROM Employees

 WHERE Email = ORIGINAL\_LOGIN()

 ) OR ORIGINAL\_LOGIN() = 'sa';

GO

-- Create security policy

CREATE SECURITY POLICY DepartmentFilter

ADD FILTER PREDICATE dbo.fn\_RestrictDepartmentAccess(DepartmentID)

ON Employees

WITH (STATE = ON);

GO

**Explanation**:

* fn\_RestrictDepartmentAccess: Checks if the user’s email (from ORIGINAL\_LOGIN) matches a department.
* SECURITY POLICY: Applies the filter to Employees.
* sa bypasses for admin access.

Test as AppUser:

-- Simulate AppUser (ensure AppUser's email is in Employees)

EXECUTE AS USER = 'AppUser';

SELECT \* FROM Employees; -- Only sees own department

REVERT;

## 9.5 Encryption

Encryption protects data at rest and in transit.

**Transparent Data Encryption (TDE)**

Encrypts the database files.

Example: Enable TDE on CompanyDB.

USE master;

GO

-- Create master key

CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'M@sterK3yP@ss!';

GO

-- Create certificate

CREATE CERTIFICATE CompanyDBCert WITH SUBJECT = 'CompanyDB Encryption';

GO

USE CompanyDB;

GO

-- Create database encryption key

CREATE DATABASE ENCRYPTION KEY

WITH ALGORITHM = AES\_256

ENCRYPTION BY SERVER CERTIFICATE CompanyDBCert;

GO

-- Enable TDE

ALTER DATABASE CompanyDB SET ENCRYPTION ON;

GO

Explanation:

Encrypts data files and backups.

Requires master key and certificate.

Column-Level Encryption

Encrypt specific columns (e.g., Photo).

Example:

-- Create a symmetric key

CREATE SYMMETRIC KEY EmployeePhotoKey

WITH ALGORITHM = AES\_256

ENCRYPTION BY CERTIFICATE CompanyDBCert;

GO

-- Add encrypted column

ALTER TABLE Employees

ADD EncryptedPhoto VARBINARY(MAX);

GO

-- Encrypt data

OPEN SYMMETRIC KEY EmployeePhotoKey

DECRYPTION BY CERTIFICATE CompanyDBCert;

UPDATE Employees

SET EncryptedPhoto = ENCRYPTBYKEY(KEY\_GUID('EmployeePhotoKey'), Photo);

-- Decrypt for viewing

SELECT

 EmployeeID,

 CAST(DECRYPTBYKEY(EncryptedPhoto) AS VARBINARY(MAX)) AS DecryptedPhoto

FROM Employees

WHERE EmployeeID = 1;

CLOSE SYMMETRIC KEY EmployeePhotoKey;

**Explanation**:

* Uses symmetric key for performance.
* ENCRYPTBYKEY/DECRYPTBYKEY: Encrypts/decrypts data.

## 9.6 Auditing

Auditing tracks user actions for compliance.

**SQL Server Audit**

Example: Audit SELECT operations on Employees.

USE master;

GO

-- Create server audit

CREATE SERVER AUDIT EmployeeAccessAudit

TO FILE (FILEPATH = 'C:\SQLAudits\')

WITH (ON\_FAILURE = CONTINUE);

GO

-- Enable audit

ALTER SERVER AUDIT EmployeeAccessAudit WITH (STATE = ON);

GO

USE CompanyDB;

GO

-- Create database audit specification

CREATE DATABASE AUDIT SPECIFICATION EmployeeAccessSpec

FOR SERVER AUDIT EmployeeAccessAudit

ADD (SELECT ON Employees BY PUBLIC)

WITH (STATE = ON);

GO

**Explanation**:

* Logs SELECT operations to a file.
* PUBLIC: Applies to all users.

View audit logs:

SELECT event\_time, action\_id, statement

FROM sys.fn\_get\_audit\_file('C:\SQLAudits\\*.sqlaudit', default, default);

**Trigger-Based Auditing**

Already implemented in EmployeeAudit (Chapter 6). Example: Extend for SELECT.

CREATE TABLE SelectAudit (

 AuditID INT IDENTITY(1,1) PRIMARY KEY,

 TableName NVARCHAR(128),

 UserName NVARCHAR(128),

 ActionDate DATETIME

);

CREATE TRIGGER trg\_AuditEmployeeSelect

ON Employees

AFTER SELECT

AS

BEGIN

 INSERT INTO SelectAudit (TableName, UserName, ActionDate)

 VALUES ('Employees', SUSER\_SNAME(), GETDATE());

END;

GO

Test:

SELECT \* FROM Employees;

SELECT \* FROM SelectAudit;

## 9.7 Advanced Example: Dynamic Access Control

Example: Create a procedure to grant permissions dynamically based on user roles.

CREATE PROCEDURE GrantDepartmentAccess

 @UserName NVARCHAR(128),

 @DepartmentID INT

AS

BEGIN

 SET NOCOUNT ON;

 BEGIN TRY

 DECLARE @SQL NVARCHAR(MAX);

 SET @SQL = N'GRANT SELECT ON Employees TO ' + QUOTENAME(@UserName) +

 N' WHERE DepartmentID = ' + CAST(@DepartmentID AS NVARCHAR(10));

 EXEC sp\_executesql @SQL;

 INSERT INTO EmployeeAudit (EmployeeID, Action, ActionDate, OldData)

 VALUES (NULL, 'GRANT\_ACCESS', GETDATE(),

 'User: ' + @UserName + ', Dept: ' + CAST(@DepartmentID AS NVARCHAR(10)));

 END TRY

 BEGIN CATCH

 INSERT INTO EmployeeAudit (EmployeeID, Action, ActionDate, OldData)

 VALUES (NULL, 'GRANT\_FAILED', GETDATE(), ERROR\_MESSAGE());

 THROW;

 END CATCH;

END;

GO

Test:

EXEC GrantDepartmentAccess 'AppUser', 1;

**Explanation**:

* Dynamically grants SELECT with a WHERE clause.
* Logs success or failure.

## 9.8 Performance Considerations

* **RLS**: Filter predicates can impact query performance; index filtered columns.
* **Encryption**: Adds CPU overhead; use selectively for sensitive data.
* **Auditing**: File-based audits can grow large; manage storage.
* **Permissions**: Minimize GRANTs to reduce security management overhead.

Example: Index for RLS.

CREATE NONCLUSTERED INDEX IX\_Employees\_Email ON Employees(Email);

## 9.9 Best Practices

* **Use Windows Authentication**: More secure than SQL logins.
* **Principle of Least Privilege**: Grant only necessary permissions.
* **Encrypt Sensitive Data**: Use TDE or column encryption for compliance.
* **Regularly Review Permissions**: Use sys.database\_permissions.
* **Backup Keys/Certificates**: Store securely outside the database.
* **Monitor Audits**: Rotate log files to manage size.

## 9.10 Exercises

1. Create a login and user, granting SELECT on Projects.
2. Implement RLS to restrict project access by department.
3. Set up TDE on a test database and verify encryption.
4. Create an audit to track UPDATE operations on Employees.
5. Write a procedure to revoke permissions dynamically.

## 9.11 Conclusion

SQL Server’s security features protect data through authentication, authorization, encryption, and auditing. You’ve learned to manage users, roles, RLS, encryption, and auditing, preparing you for advanced topics like CTEs and window functions in Chapter 10.

# Chapter 10: Best Practices and Advanced Topics

## 10.1 Introduction to Best Practices and Advanced Topics

This chapter consolidates best practices for writing robust, efficient, and maintainable T-SQL code and introduces advanced SQL Server features to enhance your programming capabilities. Best practices ensure code is readable, performant, and secure, while advanced topics like Common Table Expressions (CTEs), window functions, temporal tables, and JSON support enable complex data manipulation and analysis. We’ll use the CompanyDB schema:

* **Employees**: EmployeeID (PK), FirstName, LastName, Email, HireDate, Salary, DepartmentID (FK), Photo, YearsOfService.
* **Departments**: DepartmentID (PK), DepartmentName, Location.
* **Projects**: ProjectID (PK), ProjectName, StartDate, EndDate, Budget, DepartmentID (FK).
* **EmployeeProjects**: EmployeeID, ProjectID, Role, AssignmentDate.
* **EmployeeAudit**: AuditID (PK), EmployeeID, Action, ActionDate, OldData, NewData.

## 10.2 Best Practices for T-SQL Programming

Adhering to best practices ensures your code is maintainable, performant, and secure.

**10.2.1 Code Readability**

* **Consistent Naming**: Use prefixes (e.g., usp\_ for stored procedures, fn\_ for functions, trg\_ for triggers).
* **Comments**: Document purpose, parameters, and logic.
* **Formatting**: Use consistent indentation and casing (e.g., uppercase keywords).

Example: Well-documented procedure.

-- Purpose: Updates employee salary and logs change

-- Parameters: @EmployeeID - Employee to update, @NewSalary - New salary value

-- Author: [Your Name], 2025-08-27

CREATE PROCEDURE usp\_UpdateEmployeeSalary

 @EmployeeID INT,

 @NewSalary DECIMAL(10,2)

AS

BEGIN

 SET NOCOUNT ON;

 BEGIN TRY

 UPDATE Employees

 SET Salary = @NewSalary

 WHERE EmployeeID = @EmployeeID;

 END TRY

 BEGIN CATCH

 THROW;

 END CATCH;

END;

GO

**10.2.2 Performance**

* \*\*Avoid SELECT \*\*\* in production; specify columns.
* **Use Sargable Queries**: Avoid functions on indexed columns (e.g., WHERE YEAR(HireDate) = 2023).
* **Minimize Transactions**: Keep transactions short to reduce locking.
* **Index Strategically**: Cover frequently queried columns (Chapter 7).

**10.2.3 Security**

* **Parameterize Queries**: Prevent SQL injection.
* **Least Privilege**: Grant minimal permissions (Chapter 9).
* **Encrypt Sensitive Data**: Use TDE or column encryption.

**10.2.4 Error Handling**

* Always use TRY...CATCH for robust error management.
* Log errors to audit tables.

**10.2.5 Version Control**

* Store scripts in a repository (e.g., Git).
* Use ALTER instead of CREATE/DROP for updates.
* Backup before schema changes:

BACKUP DATABASE CompanyDB TO DISK = 'C:\Backups\CompanyDB\_20250827.bak';

## 10.3 Common Table Expressions (CTEs)

CTEs provide a readable way to define temporary result sets, useful for recursive or hierarchical queries.

**Basic CTE**

Example: List employees with above-average salaries.

WITH HighEarners AS (

 SELECT EmployeeID, FirstName, LastName, Salary

 FROM Employees

 WHERE Salary > (SELECT AVG(Salary) FROM Employees)

)

SELECT FirstName, LastName, Salary

FROM HighEarners

ORDER BY Salary DESC;

**Explanation**:

* WITH: Defines the CTE.
* Replaces subquery for clarity.
* Reusable within the query.

**Advanced CTE: Recursive Hierarchy**

Assume a new table for employee reporting structure.

CREATE TABLE EmployeeHierarchy (

 EmployeeID INT PRIMARY KEY,

 ManagerID INT,

 FOREIGN KEY (EmployeeID) REFERENCES Employees(EmployeeID),

 FOREIGN KEY (ManagerID) REFERENCES Employees(EmployeeID)

);

INSERT INTO EmployeeHierarchy (EmployeeID, ManagerID)

VALUES (1, NULL), (2, 1), (4, 2), (5, 2), (6, 1), (7, 4);

Recursive CTE to list reporting hierarchy:

WITH EmployeeTree AS (

 -- Anchor: Top-level managers

 SELECT EmployeeID, FirstName, LastName, ManagerID, 0 AS Level

 FROM Employees e

 LEFT JOIN EmployeeHierarchy h ON e.EmployeeID = h.EmployeeID

 WHERE h.ManagerID IS NULL

 UNION ALL

 -- Recursive: Subordinates

 SELECT e.EmployeeID, e.FirstName, e.LastName, h.ManagerID, Level + 1

 FROM Employees e

 JOIN EmployeeHierarchy h ON e.EmployeeID = h.EmployeeID

 JOIN EmployeeTree et ON h.ManagerID = et.EmployeeID

)

SELECT

 REPLICATE(' ', Level) + FirstName + ' ' + LastName AS EmployeeName,

 Level,

 (SELECT FirstName + ' ' + LastName FROM Employees WHERE EmployeeID = EmployeeTree.ManagerID) AS ManagerName

FROM EmployeeTree

ORDER BY Level, LastName;

**Output**:

| **EmployeeName** | **Level** | **ManagerName** |
| --- | --- | --- |
| John Doe | 0 | NULL |
| Jane Smith | 1 | John Doe |
| Tom Wilson | 1 | John Doe |
| Alice Brown | 2 | Jane Smith |
| Sarah Davis | 2 | Jane Smith |
| Pam Beesly | 3 | Alice Brown |

**Explanation**:

* Anchor: Starts with top-level manager (no ManagerID).
* Recursive: Joins subordinates iteratively.
* REPLICATE: Indents names for visual hierarchy.

## 10.4 Window Functions

Window functions perform calculations across a set of rows (a “window”) without grouping.

**Basic Window Function**

Example: Rank employees by salary within departments.

SELECT

 e.FirstName,

 e.LastName,

 d.DepartmentName,

 e.Salary,

 RANK() OVER (PARTITION BY e.DepartmentID ORDER BY e.Salary DESC) AS SalaryRank

FROM Employees e

JOIN Departments d ON e.DepartmentID = d.DepartmentID;

**Output**:

| **FirstName** | **LastName** | **DepartmentName** | **Salary** | **SalaryRank** |
| --- | --- | --- | --- | --- |
| Jane | Smith | IT | 82000.00 | 1 |
| Tom | Wilson | IT | 72000.00 | 2 |
| John | Doe | HR | 75000.00 | 1 |
| Sarah | Davis | HR | 68000.00 | 2 |
| Alice | Brown | HR | 60000.00 | 3 |
| Pam | Beesly | HR | 62000.00 | 4 |

**Explanation**:

* PARTITION BY: Groups by DepartmentID.
* RANK(): Assigns rank within each department.
* ORDER BY: Ranks by salary descending.

**Advanced Window Function: Running Total**

Example: Calculate cumulative project budget by department.

SELECT

 d.DepartmentName,

 p.ProjectName,

 p.Budget,

 SUM(p.Budget) OVER (PARTITION BY p.DepartmentID ORDER BY p.StartDate) AS RunningBudget

FROM Projects p

JOIN Departments d ON p.DepartmentID = d.DepartmentID;

**Output**:

| **DepartmentName** | **ProjectName** | **Budget** | **RunningBudget** |
| --- | --- | --- | --- |
| HR | Recruitment Portal | 50000.00 | 50000.00 |
| IT | ERP Implementation | 150000.00 | 150000.00 |
| IT | Data Migration | 80000.00 | 230000.00 |

**Explanation**:

* SUM...OVER: Calculates running total within department, ordered by StartDate.

**10.5 Temporal Tables**

Temporal tables track historical data automatically.

**Creating a Temporal Table**

Example: Make Employees a temporal table.

ALTER TABLE Employees

ADD

 SysStartTime DATETIME2 GENERATED ALWAYS AS ROW START NOT NULL,

 SysEndTime DATETIME2 GENERATED ALWAYS AS ROW END NOT NULL,

 PERIOD FOR SYSTEM\_TIME (SysStartTime, SysEndTime);

ALTER TABLE Employees

SET (SYSTEM\_VERSIONING = ON (HISTORY\_TABLE = dbo.EmployeesHistory));

**Explanation**:

* Adds system-time columns for versioning.
* EmployeesHistory: Stores historical data.

Test:

UPDATE Employees SET Salary = 85000 WHERE EmployeeID = 1;

SELECT \* FROM EmployeesHistory WHERE EmployeeID = 1;

**Output (History)**:

| **EmployeeID** | **FirstName** | **LastName** | **...** | **Salary** | **SysStartTime** | **SysEndTime** |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | John | Doe | ... | 75000.00 | 2023-01-15 00:00:00 | 2025-08-27 09:36:00 |

**10.6 JSON Support**

SQL Server supports JSON for semi-structured data.

**Querying JSON**

Example: Store and query employee metadata.

ALTER TABLE Employees

ADD Metadata NVARCHAR(MAX) CHECK (ISJSON(Metadata) = 1);

UPDATE Employees

SET Metadata = '{"Skills": ["SQL", "Management"], "Certification": "PMP"}'

WHERE EmployeeID = 2;

SELECT

 EmployeeID,

 FirstName,

 JSON\_VALUE(Metadata, '$.Certification') AS Certification,

 JSON\_QUERY(Metadata, '$.Skills') AS Skills

FROM Employees

WHERE JSON\_VALUE(Metadata, '$.Certification') = 'PMP';

**Output**:

| **EmployeeID** | **FirstName** | **Certification** | **Skills** |
| --- | --- | --- | --- |
| 2 | Jane | PMP | ["SQL", "Management"] |

**Explanation**:

* JSON\_VALUE: Extracts scalar values.
* JSON\_QUERY: Extracts JSON arrays/objects.
* ISJSON: Validates JSON.

**10.7 Advanced Example: Combining CTEs and Window Functions**

Example: Analyze employee project contributions.

WITH ProjectAssignments AS (

 SELECT

 e.EmployeeID,

 e.FirstName + ' ' + e.LastName AS FullName,

 p.ProjectName,

 p.Budget,

 ROW\_NUMBER() OVER (PARTITION BY e.EmployeeID ORDER BY p.Budget DESC) AS ProjectRank

 FROM Employees e

 JOIN EmployeeProjects ep ON e.EmployeeID = ep.EmployeeID

 JOIN Projects p ON ep.ProjectID = p.ProjectID

)

SELECT

 FullName,

 ProjectName,

 Budget,

 SUM(Budget) OVER (PARTITION BY FullName) AS TotalBudgetContribution

FROM ProjectAssignments

WHERE ProjectRank = 1;

**Output**:

| **FullName** | **ProjectName** | **Budget** | **TotalBudgetContribution** |
| --- | --- | --- | --- |
| John Doe | ERP Implementation | 150000.00 | 150000.00 |
| Jane Smith | ERP Implementation | 150000.00 | 150000.00 |
| Pam Beesly | Recruitment Portal | 50000.00 | 50000.00 |
| Tom Wilson | Data Migration | 80000.00 | 80000.00 |

**Explanation**:

* CTE assigns row numbers to projects per employee.
* Filters for top project per employee.
* Calculates total budget contribution.

**10.8 Performance Considerations**

* **CTEs**: Optimize inner SELECT statements; consider temporary tables for large datasets.
* **Window Functions**: Index columns in PARTITION BY and ORDER BY.
* **Temporal Tables**: Monitor history table size; index SysStartTime, SysEndTime.
* **JSON**: Use sparingly for large datasets; prefer relational storage.

Example: Index for window function.

CREATE NONCLUSTERED INDEX IX\_Employees\_Dept\_Salary

ON Employees(DepartmentID, Salary);

## 10.9 Best Practices for Advanced Features

* **CTEs**: Use for readability; avoid deep nesting.
* **Window Functions**: Test performance on large datasets.
* **Temporal Tables**: Enable only for tables needing history.
* **JSON**: Validate with ISJSON; index computed columns for frequent queries.
* **Test and Monitor**: Use execution plans and DMVs (sys.dm\_exec\_query\_stats).

## 10.10 Exercises

1. Write a recursive CTE to list all projects under a department hierarchy (assume a DepartmentHierarchy table).
2. Create a window function to calculate running total of salaries by hire date.
3. Set up a temporal table for Projects and query historical budgets.
4. Store and query JSON data for employee skills.
5. Combine a CTE and window function to rank employees by project count.

## 10.11 Conclusion

This chapter solidified best practices for writing robust T-SQL and introduced advanced features like CTEs, window functions, temporal tables, and JSON support. These tools enhance your ability to handle complex data scenarios, preparing you for real-world SQL Server development.

# Chapter 11: Capstone Project - Building a Comprehensive Sales Dashboard Backend with AdventureWorks2022

## 11.1 Project Overview

This capstone chapter integrates all the concepts from the booklet into a practical, in-depth software project: developing a robust backend for a sales dashboard web application using the AdventureWorks2022 sample database. AdventureWorks2022 models a fictional bicycle manufacturing company, with rich schemas for production, sales, purchasing, and human resources. We'll focus on the **Sales** schema for dashboard features, while incorporating elements from **Production** (products), **Person** (customers/employees), and **HumanResources** (employees) to create a comprehensive system.

The dashboard backend will support a hypothetical web application (e.g., built with ASP.NET Core or similar) that calls SQL Server stored procedures and functions via APIs to perform CRUD operations, generate reports, and ensure data integrity. Key features include:

* **Reporting Metrics**: Total sales by customer/product/territory, year-over-year growth, top performers.
* **CRUD Interfaces**: Add/update/delete sales orders, customers, and products securely.
* **Advanced Analytics**: Trend analysis, rankings, historical tracking.
* **Security and Auditing**: Role-based access, row-level security, encryption for sensitive data.
* **Performance Optimization**: Indexes, efficient queries to handle large datasets.

We'll build this step-by-step, emphasizing depth:

* Stored procedures as "API endpoints" for web calls (e.g., via ADO.NET or Entity Framework).
* UDFs for reusable calculations.
* Triggers for automatic validation/auditing.
* Transactions for safe operations.
* Security to protect data.
* Advanced T-SQL for sophisticated insights.

This project assumes the web app connects via connection strings and executes procs/functions. For example, in C#:

using (SqlConnection conn = new SqlConnection("your\_connection\_string"))

{

 conn.Open();

 using (SqlCommand cmd = new SqlCommand("usp\_GetSalesByCustomer", conn))

 {

 cmd.CommandType = CommandType.StoredProcedure;

 cmd.Parameters.AddWithValue("@CustomerID", 123);

 using (SqlDataReader reader = cmd.ExecuteReader())

 {

 // Process results

 }

 }

}

## 11.2 Setting Up the Environment

To ensure a solid foundation, we'll go beyond basic restoration with scripting, verification, and initial customizations.

**Restoring the Database**

1. Download AdventureWorks2022.bak from [Microsoft's GitHub](https://github.com/Microsoft/sql-server-samples/releases/download/adventureworks/AdventureWorks2022.bak).
2. Script the restore for repeatability:

RESTORE DATABASE AdventureWorks2022

FROM DISK = 'C:\Backups\AdventureWorks2022.bak'

WITH MOVE 'AdventureWorks2022' TO 'C:\SQLData\AdventureWorks2022.mdf',

MOVE 'AdventureWorks2022\_log' TO 'C:\SQLData\AdventureWorks2022\_log.ldf',

REPLACE;

1. Verify key tables and add custom extensions (e.g., for auditing):

USE AdventureWorks2022;

GO

-- Verify schema

SELECT \* FROM sys.schemas WHERE name IN ('Sales', 'Production', 'HumanResources');

-- Create audit table (integrates with triggers later)

CREATE TABLE dbo.SalesAudit (

 AuditID INT IDENTITY PRIMARY KEY,

 SalesOrderID INT,

 Action NVARCHAR(50),

 ActionDate DATETIME DEFAULT GETDATE(),

 UserName NVARCHAR(128) DEFAULT SUSER\_SNAME(),

 OldData NVARCHAR(MAX),

 NewData NVARCHAR(MAX)

);

GO

-- Add temporal support to SalesOrderHeader for history (Chapter 10)

ALTER TABLE Sales.SalesOrderHeader

ADD

 SysStartTime DATETIME2 GENERATED ALWAYS AS ROW START NOT NULL DEFAULT GETDATE(),

 SysEndTime DATETIME2 GENERATED ALWAYS AS ROW END NOT NULL DEFAULT CONVERT(DATETIME2, '9999-12-31 23:59:59.9999999'),

 PERIOD FOR SYSTEM\_TIME (SysStartTime, SysEndTime);

ALTER TABLE Sales.SalesOrderHeader

SET (SYSTEM\_VERSIONING = ON (HISTORY\_TABLE = Sales.SalesOrderHeaderHistory));

GO

**Explanation**: This setup script restores the DB, verifies schemas, adds an audit table (for triggers), and enables temporal tables on SalesOrderHeader for tracking order history changes over time. Temporal tables automatically maintain versions, useful for dashboard audits.

**Performance Baseline**

Create initial indexes for common joins:

CREATE NONCLUSTERED INDEX IX\_SalesOrderDetail\_ProductID ON Sales.SalesOrderDetail(ProductID);

CREATE NONCLUSTERED INDEX IX\_SalesOrderHeader\_CustomerID\_OrderDate ON Sales.SalesOrderHeader(CustomerID, OrderDate);

**Explanation**: These indexes (Chapter 7) optimize frequent sales queries, reducing I/O for large datasets.

## 11.3 Basic SQL Queries: Exploring and Manipulating Sales Data

Dive deep into CRUD with examples tied to dashboard use cases, including filtering, aggregation, and basic error handling.

**SELECT: Dashboard Metrics**

Example: Detailed customer sales summary.

SELECT

 c.CustomerID,

 p.FirstName + ' ' + p.LastName AS CustomerName,

 st.Name AS Territory,

 COUNT(soh.SalesOrderID) AS OrderCount,

 SUM(soh.TotalDue) AS TotalSales,

 AVG(soh.TotalDue) AS AvgOrderValue

FROM Sales.Customer c

JOIN Person.Person p ON c.PersonID = p.BusinessEntityID

JOIN Sales.SalesOrderHeader soh ON c.CustomerID = soh.CustomerID

JOIN Sales.SalesTerritory st ON soh.TerritoryID = st.TerritoryID

WHERE soh.OrderDate BETWEEN '2022-01-01' AND '2022-12-31'

GROUP BY c.CustomerID, p.FirstName, p.LastName, st.Name

HAVING SUM(soh.TotalDue) > 10000

ORDER BY TotalSales DESC;

**Explanation**: Aggregates orders per customer (Chapter 2), filters by date, groups with HAVING for high-value customers. Useful for dashboard charts.

**INSERT: Adding a New Order**

Example: Insert with validation.

BEGIN TRY

 INSERT INTO Sales.SalesOrderHeader (CustomerID, OrderDate, DueDate, ShipDate, Status, SubTotal, TaxAmt, Freight, TotalDue, TerritoryID, BillToAddressID, ShipToAddressID)

 VALUES (1, GETDATE(), DATEADD(DAY, 7, GETDATE()), DATEADD(DAY, 5, GETDATE()), 5, 150.00, 12.00, 15.00, 177.00, 1, 1, 1);

 DECLARE @NewOrderID INT = SCOPE\_IDENTITY();

 INSERT INTO Sales.SalesOrderDetail (SalesOrderID, ProductID, OrderQty, UnitPrice, UnitPriceDiscount, LineTotal)

 VALUES (@NewOrderID, 707, 2, 75.00, 0.00, 150.00);

END TRY

BEGIN CATCH

 SELECT ERROR\_MESSAGE();

END CATCH;

**Explanation**: Adds header and detail (Chapter 2), uses SCOPE\_IDENTITY() for ID retrieval. Integrates basic error handling (Chapter 8).

**UPDATE: Modifying an Order**

Example: Update status with check.

UPDATE Sales.SalesOrderHeader

SET Status = 5, -- Shipped

 ModifiedDate = GETDATE()

WHERE SalesOrderID = 43659

AND Status = 2; -- In process

**Explanation**: Conditional update to prevent invalid changes.

**DELETE: Removing a Detail Line**

Example: Delete with transaction.

BEGIN TRANSACTION;

BEGIN TRY

 DELETE FROM Sales.SalesOrderDetail

 WHERE SalesOrderID = 43659 AND ProductID = 707;

 COMMIT;

END TRY

BEGIN CATCH

 ROLLBACK;

 SELECT ERROR\_MESSAGE();

END CATCH;

**Explanation**: Ensures atomicity (Chapter 8).

11.4 Joins and Subqueries: Complex Dashboard Reports

Expand with multi-table joins and correlated subqueries for in-depth analysis.

**Join Example: Product Sales with Inventory**

SELECT

 p.ProductID,

 p.Name,

 SUM(sod.OrderQty) AS TotalQuantitySold,

 AVG(sod.UnitPrice) AS AvgPrice,

 (SELECT SUM(Quantity) FROM Production.ProductInventory pi WHERE pi.ProductID = p.ProductID) AS CurrentInventory

FROM Production.Product p

INNER JOIN Sales.SalesOrderDetail sod ON p.ProductID = sod.ProductID

INNER JOIN Sales.SalesOrderHeader soh ON sod.SalesOrderID = soh.SalesOrderID

WHERE soh.OrderDate >= '2022-01-01'

GROUP BY p.ProductID, p.Name

HAVING SUM(sod.OrderQty) > 100

ORDER BY TotalQuantitySold DESC;

**Explanation**: Joins three tables, uses subquery for inventory (Chapter 3). Filters high-volume products for dashboard stock alerts.

**Subquery Example: Customers with Above-Average Orders**

SELECT

 c.CustomerID,

 p.FirstName + ' ' + p.LastName AS CustomerName,

 (SELECT COUNT(\*) FROM Sales.SalesOrderHeader WHERE CustomerID = c.CustomerID) AS OrderCount

FROM Sales.Customer c

JOIN Person.Person p ON c.PersonID = p.BusinessEntityID

WHERE (SELECT AVG(TotalDue) FROM Sales.SalesOrderHeader WHERE CustomerID = c.CustomerID) >

 (SELECT AVG(TotalDue) FROM Sales.SalesOrderHeader)

ORDER BY OrderCount DESC;

**Explanation**: Correlated subqueries for counts and averages (Chapter 3), identifying loyal customers.

## 11.5 Stored Procedures: Web Interface Endpoints

Create procs as callable endpoints for web CRUD and reports, with params, transactions, and error handling.

**Reporting Proc: Sales Summary**

CREATE PROCEDURE usp\_GetSalesSummary

 @TerritoryID INT = NULL,

 @StartDate DATE = NULL,

 @EndDate DATE = NULL

AS

BEGIN

 SET NOCOUNT ON;

 BEGIN TRY

 SELECT

 st.Name AS Territory,

 SUM(soh.TotalDue) AS TotalSales,

 COUNT(soh.SalesOrderID) AS OrderCount,

 AVG(soh.TotalDue) AS AvgOrderValue

 FROM Sales.SalesOrderHeader soh

 JOIN Sales.SalesTerritory st ON soh.TerritoryID = st.TerritoryID

 WHERE (@TerritoryID IS NULL OR soh.TerritoryID = @TerritoryID)

 AND (@StartDate IS NULL OR soh.OrderDate >= @StartDate)

 AND (@EndDate IS NULL OR soh.OrderDate <= @EndDate)

 GROUP BY st.Name

 ORDER BY TotalSales DESC;

 END TRY

 BEGIN CATCH

 THROW;

 END CATCH;

END;

GO

Execute:

EXEC usp\_GetSalesSummary @StartDate = '2022-01-01', @EndDate = '2022-12-31';

**Explanation**: Flexible filtering, aggregates for dashboard KPIs (Chapter 4). Web app can call this for charts.

**CRUD Proc: Add Sales Order**

CREATE PROCEDURE usp\_AddSalesOrder

 @CustomerID INT,

 @TerritoryID INT,

 @SubTotal DECIMAL(19,4),

 @TaxAmt DECIMAL(19,4),

 @Freight DECIMAL(19,4),

 @ProductID INT,

 @OrderQty INT,

 @UnitPrice DECIMAL(19,4),

 @NewOrderID INT OUTPUT

AS

BEGIN

 SET NOCOUNT ON;

 BEGIN TRANSACTION;

 BEGIN TRY

 -- Insert header

 INSERT INTO Sales.SalesOrderHeader (CustomerID, TerritoryID, OrderDate, DueDate, ShipDate, Status, SubTotal, TaxAmt, Freight, TotalDue, BillToAddressID, ShipToAddressID)

 VALUES (@CustomerID, @TerritoryID, GETDATE(), DATEADD(DAY, 7, GETDATE()), DATEADD(DAY, 5, GETDATE()), 5, @SubTotal, @TaxAmt, @Freight, @SubTotal + @TaxAmt + @Freight, 1, 1);

 SET @NewOrderID = SCOPE\_IDENTITY();

 -- Insert detail

 INSERT INTO Sales.SalesOrderDetail (SalesOrderID, ProductID, OrderQty, UnitPrice, UnitPriceDiscount, LineTotal)

 VALUES (@NewOrderID, @ProductID, @OrderQty, @UnitPrice, 0.00, @OrderQty \* @UnitPrice);

 COMMIT TRANSACTION;

 END TRY

 BEGIN CATCH

 IF @@TRANCOUNT > 0 ROLLBACK TRANSACTION;

 THROW;

 END CATCH;

END;

GO

**Explanation**: Atomic insert for header/detail, outputs ID for web confirmation (Chapters 4, 8).

Similar procs for Update/Delete, with validation.

## 11.6 User-Defined Functions: Custom Calculations

UDFs for metrics, integrated into procs/queries.

**Scalar UDF: Profit Margin**

CREATE FUNCTION dbo.fn\_ProductProfitMargin (@ProductID INT)

RETURNS DECIMAL(5,2)

AS

BEGIN

 DECLARE @ListPrice DECIMAL(19,4), @StandardCost DECIMAL(19,4);

 SELECT @ListPrice = ListPrice, @StandardCost = StandardCost

 FROM Production.Product WHERE ProductID = @ProductID;

 RETURN IIF(@ListPrice = 0, 0, ((@ListPrice - @StandardCost) / @ListPrice) \* 100);

END;

GO

Use:

SELECT p.ProductID, p.Name, dbo.fn\_ProductProfitMargin(p.ProductID) AS Margin

FROM Production.Product p;

**Explanation**: Handles zero division (Chapter 5).

**Table-Valued UDF: Customer Orders**

CREATE FUNCTION dbo.fn\_GetCustomerOrders (@CustomerID INT)

RETURNS TABLE

AS

RETURN

 SELECT soh.SalesOrderID, soh.OrderDate, SUM(sod.LineTotal) AS Total

 FROM Sales.SalesOrderHeader soh

 JOIN Sales.SalesOrderDetail sod ON soh.SalesOrderID = sod.SalesOrderID

 WHERE soh.CustomerID = @CustomerID

 GROUP BY soh.SalesOrderID, soh.OrderDate;

GO

Use in proc:

SELECT \* FROM dbo.fn\_GetCustomerOrders(1);

**Explanation**: Returns table for web order history (Chapter 5).

## 11.7 Triggers: Data Integrity and Auditing

Multiple triggers for validation and logging.

**AFTER INSERT/UPDATE Trigger: Validate Inventory**

CREATE TRIGGER trg\_ValidateInventoryOnOrder

ON Sales.SalesOrderDetail

AFTER INSERT, UPDATE

AS

BEGIN

 IF EXISTS (

 SELECT 1

 FROM inserted i

 JOIN Production.ProductInventory pi ON i.ProductID = pi.ProductID

 WHERE pi.Quantity < i.OrderQty

 )

 BEGIN

 RAISERROR ('Insufficient inventory for product.', 16, 1);

 ROLLBACK TRANSACTION;

 END;

 INSERT INTO dbo.SalesAudit (SalesOrderID, Action, OldData, NewData)

 SELECT i.SalesOrderID, 'ORDER\_DETAIL\_CHANGE', (SELECT \* FROM deleted FOR XML RAW), (SELECT \* FROM inserted FOR XML RAW)

 FROM inserted i;

END;

GO

**Explanation**: Checks stock, logs changes (Chapter 6).

**DDL Trigger: Audit Schema Changes**

CREATE TRIGGER trg\_AuditSchemaChanges

ON DATABASE

FOR CREATE\_TABLE, ALTER\_TABLE, DROP\_TABLE

AS

BEGIN

 INSERT INTO dbo.SalesAudit (Action, OldData)

 VALUES ('SCHEMA\_CHANGE', EVENTDATA().value('(/EVENT\_INSTANCE/TSQLCommand/CommandText)[1]', 'NVARCHAR(MAX)'));

END;

GO

**Explanation**: Logs DDL for compliance (Chapter 6).

## 11.8 Indexes and Performance Optimization

Analyze and optimize with DMVs and plans.

**Missing Index Suggestion**

SELECT \* FROM sys.dm\_db\_missing\_index\_details WHERE database\_id = DB\_ID();

**Create Indexes Based on Analysis**

Example: For order date queries.

CREATE NONCLUSTERED INDEX IX\_SalesOrderHeader\_OrderDate

ON Sales.SalesOrderHeader(OrderDate)

INCLUDE (TotalDue, CustomerID);

Before/After Test:

SET STATISTICS IO ON;

-- Run query from 11.3

SET STATISTICS IO OFF;

**Explanation**: Shows I/O reduction (Chapter 7). For large AdventureWorks data, this cuts logical reads significantly.

**11.9 Transactions and Error Handling: Safe Web Operations**

Embed in procs (as in 11.5), with isolation.

Example: Set isolation in proc.

ALTER PROCEDURE usp\_AddSalesOrder

-- Params...

AS

BEGIN

 SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

 -- Rest of code

END;

**Explanation**: Prevents dirty reads in concurrent web requests (Chapter 8).

**11.10 Security: Securing Web Access**

Roles, RLS, encryption.

**Role Creation**

CREATE ROLE DashboardUser;

GRANT EXECUTE ON usp\_GetSalesSummary TO DashboardUser;

GRANT EXECUTE ON usp\_AddSalesOrder TO DashboardUser;

-- Add web app user to role

**RLS: Territory-Based**

CREATE FUNCTION fn\_TerritoryAccess (@TerritoryID INT)

RETURNS TABLE WITH SCHEMABINDING

AS RETURN SELECT 1 WHERE @TerritoryID = CAST(SESSION\_CONTEXT(N'TerritoryID') AS INT);

CREATE SECURITY POLICY TerritoryFilter

ADD FILTER PREDICATE dbo.fn\_TerritoryAccess(TerritoryID) ON Sales.SalesOrderHeader

WITH (STATE = ON);

In web app, set SESSION\_CONTEXT post-login.

**Encryption: Sensitive Data**

CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'StrongPass!';

CREATE CERTIFICATE SalesCert WITH SUBJECT = 'Sales Encryption';

CREATE SYMMETRIC KEY SalesKey WITH ALGORITHM = AES\_256 ENCRYPTION BY CERTIFICATE SalesCert;

ALTER TABLE Sales.Customer ADD EncryptedEmail VARBINARY(MAX);

OPEN SYMMETRIC KEY SalesKey DECRYPTION BY CERTIFICATE SalesCert;

UPDATE Sales.Customer

SET EncryptedEmail = ENCRYPTBYKEY(KEY\_GUID('SalesKey'), EmailAddress)

FROM Person.EmailAddress ea WHERE Customer.PersonID = ea.BusinessEntityID;

CLOSE SYMMETRIC KEY SalesKey;

**Explanation**: Encrypts emails (Chapter 9).

## 11.11 Advanced Topics: Sophisticated Analytics

CTEs, windows, temporal, JSON.

**CTE with Window: Sales Trends**

WITH MonthlySales AS (

 SELECT

 YEAR(soh.OrderDate) \* 100 + MONTH(soh.OrderDate) AS MonthKey,

 SUM(soh.TotalDue) AS MonthlyTotal

 FROM Sales.SalesOrderHeader soh

 GROUP BY YEAR(soh.OrderDate) \* 100 + MONTH(soh.OrderDate)

)

SELECT

 MonthKey,

 MonthlyTotal,

 AVG(MonthlyTotal) OVER (ORDER BY MonthKey ROWS BETWEEN 3 PRECEDING AND CURRENT ROW) AS MovingAvg

FROM MonthlySales;

**Explanation**: Moving average for trends (Chapter 10).

**Temporal Query: Historical Orders**

SELECT SalesOrderID, TotalDue, SysStartTime, SysEndTime

FROM Sales.SalesOrderHeader FOR SYSTEM\_TIME ALL

WHERE SalesOrderID = 43659

ORDER BY SysStartTime;

**Explanation**: Views changes over time.

**JSON: Export Metrics**

SELECT

 CustomerID,

 (SELECT soh.SalesOrderID, soh.TotalDue FOR JSON PATH) AS Orders

FROM Sales.Customer c

JOIN Sales.SalesOrderHeader soh ON c.CustomerID = soh.CustomerID

FOR JSON PATH;

**Explanation**: JSON output for web API.

## 11.12 Conclusion and Web Integration

This expanded project demonstrates a full backend, callable from web apps via stored procs/UDFs. For integration, use Entity Framework to map procs to methods. Next steps: Add SSIS for ETL, Power BI for visuals, or Azure migration.