

Prepared by Randall Fadler

May 19, 2025

***Mastering SQL Server: From Fundamentals to Performance Power***

**Part I: The Foundations**

1. **Introduction to SQL Server and Editions**
2. **Installation and Configuration Best Practices**
3. **SQL Server Management Studio (SSMS) Overview**
4. **Understanding Databases, Tables, and Indexes**
5. **Data Types and Constraints**

**Part II: T-SQL Essentials**

1. **Querying with SELECT, INSERT, UPDATE, DELETE**
2. **Joins, Subqueries, and Set Operators**
3. **Functions, Expressions, and CASE Logic**
4. **Stored Procedures, Views, and Triggers**
5. **Transactions and Error Handling**

**Part III: Advanced SQL Features**

1. **System-Versioned Temporal Tables**
2. **Partitioning Strategies and Implementation**
3. **Dynamic SQL and Execution Plans**
4. **SQL Server Sequences and Identity Columns**
5. **Change Data Capture vs. Change Tracking**

**Part IV: Performance and Optimization**

1. **Indexing Deep Dive: Clustered, Non-Clustered, and Columnstore**
2. **Execution Plans and Query Tuning**
3. **Statistics, Histograms, and Cardinality Estimation**
4. **Best Practices for Writing High-Performance Queries**
5. **Using DMVs and DMFs for Monitoring**

**Part V: Administration and Security**

1. **Backup, Restore, and Recovery Models**
2. **Security and Role-Based Access**
3. **Automating Jobs with SQL Server Agent**
4. **Database Maintenance Plans**
5. **High Availability and Disaster Recovery (HADR)**

**Part VI: Case Studies & Projects**

1. **Building a Performance Dashboard in T-SQL**
2. **Implementing a System-Versioned Table in Practice**
3. **Partitioning a Historical Sales Table**
4. **Building a Rust-Powered CLI for Querying SQL Server**
5. **Troubleshooting Real-World Query Bottlenecks**

# 📖 Chapter 1: Introduction to SQL Server and Editions

**What is SQL Server?** Microsoft SQL Server is a relational database management system (RDBMS) designed to store, retrieve, and manage data efficiently. It powers everything from small desktop apps to massive enterprise systems with billions of records.

## Why SQL Server?

* Mature and stable platform
* Rich T-SQL language support
* Strong integration with Microsoft tools (e.g., .NET, Azure)
* Advanced features like temporal tables, in-memory OLTP, and partitioning

## Editions Overview

| **Edition** | **Best For** | **Key Features** |
| --- | --- | --- |
| **Express** | Lightweight/local apps, learning | Free, limited to 10 GB per DB, 1 GB RAM per instance |
| **Developer** | Development and testing | Full Enterprise features, but not licensed for production |
| **Standard** | Small to medium production workloads | Core RDBMS features, 128 GB RAM limit |
| **Enterprise** | Mission-critical enterprise systems | Full feature set: Always On, compression, in-memory, etc. |
| **Azure SQL** | Cloud-native deployments | Fully managed PaaS, built-in high availability |

> ✨ *Tip:* SQL Server Developer Edition is perfect for learning and personal projects—it’s free and feature-complete.

## Licensing Models

* **Core-based licensing** for larger systems
* **Server + CAL** (Client Access License) model for smaller environments

Next, we’ll flow into **Chapter 2: Installation and Configuration Best Practices**—where we can cover setup choices, tempdb optimization, and basic security hardening.

# Chapter 2: Installation and Configuration Best Practices

**Why This Matters:** A well-configured SQL Server installation can mean the difference between “set it and forget it” and “why is this crawling at 2 a.m.?”

**🔧** Pre-Installation Planning

* **Hardware considerations:** Prioritize fast disks (SSD or NVMe), ample memory, and multiple CPU cores.
* **Capacity planning:** Estimate current and future storage and workload demands.
* **Windows configuration:** Use a dedicated service account, disable unnecessary services, and ensure up-to-date security patches.

**💿** Installing SQL Server

* Choose appropriate edition (see Chapter 1).
* Opt for **custom installation** to select features like:
  + Database Engine Services
  + SQL Server Agent
  + Full-Text Search (if needed)
* Set collation appropriately (default is usually fine unless multilingual or legacy systems are involved).

**⚙️** Post-Installation Configuration

* **Set max server memory:** Prevent SQL Server from hogging all available RAM.
* **Configure** tempdb**:**
  + Use multiple data files (1 per core up to 8 is a good starting point).
  + Place tempdb on fast storage.
* **Enable Instant File Initialization** (for faster file allocation during growth).
* **Configure backup compression** and default backup locations.
* **Enable TCP/IP protocol** via SQL Server Configuration Manager.

**🛡️** Security Best Practices

* Use **Windows Authentication** where possible.
* Disable the sa account or rename it.
* Limit sysadmin role to essential accounts only.
* Regularly audit logins and permissions.

**🧪** Optional Tools and Enhancements

* **SQL Server Management Studio (SSMS)** – your day-to-day GUI.
* **SQL Server Data Tools (SSDT)** – for database projects and deployments.
* **SQLCMD** – for scripting and automating installs.
* **dbatools (PowerShell module)** – for automation, migrations, and advanced configuration.

In Chapter 3, we’ll explore **SQL Server Management Studio (SSMS)** and how to navigate it efficiently—perfect for someone building tools like your Rust-powered CLI or optimizing SQL dev workflows.

# Chapter 3: Navigating SQL Server Management Studio (SSMS)

What is SSMS? SQL Server Management Studio (SSMS) is Microsoft’s flagship GUI tool for managing, developing, and administrating SQL Server. Think of it as your command center for anything SQL-related—query writing, object exploration, performance tuning, and beyond.

**🧭 Key Components of the Interface**

* **Object Explorer**: View and manage databases, tables, views, stored procedures, and more. It’s your filesystem for SQL.
* **Query Editor**: Where you write and run T-SQL scripts. Includes syntax highlighting, IntelliSense, and execution history.
* **Activity Monitor**: Real-time dashboard for server performance—CPU, I/O, expensive queries, and blocking sessions.
* **Registered Servers**: Organize connections to different instances—perfect for managing dev, test, and prod environments.

> 💡 *Pro Tip:* You can split Object Explorer and Query Editor into separate windows or monitors—perfect for multitasking.

**⚙️ Workflow Tips and Customizations**

* **Use keyboard shortcuts** (F5 to execute, Ctrl+R to toggle Results pane).
* **Drag column names** from Object Explorer directly into your query window.
* **Change Results to Text** (Ctrl+T) for cleaner output formatting.
* **Use Templates** from the View menu to quickly generate common T-SQL statements.

**📊 Built-in Tools You Should Know**

* **Execution Plan Viewer**: Visualize how SQL Server runs a query—useful for performance tuning.
* **SQL Server Profiler** (optional install): Capture and analyze real-time SQL activity.
* **Database Diagrams**: Visually map out table relationships (ER-style modeling).
* **Generate Scripts Wizard**: Script out entire databases or specific objects.

**☁️ SSMS and Azure**

If you're managing cloud-based databases like Azure SQL:

* SSMS connects seamlessly via Authentication: Azure Active Directory
* Cloud-specific options like **Geo-Replication**, **Intelligent Insights**, and **Scaling Tiers** are just clicks away

In **Chapter 4: Understanding Databases, Tables, and Indexes**, we’ll dig into schema architecture, table design, and the importance of smart indexing—especially relevant to your performance tuning expertise.

# Chapter 4: Understanding Databases, Tables, and Indexes

**Why this chapter matters: Grasping how databases, tables, and indexes work together is essential for designing scalable, performant, and well-organized systems.**

**🧱** What is a Database?

**A database is a container that holds all your data objects—tables, views, stored procedures, and more. In SQL Server, a single instance can host many databases.**

**Key elements:**

* **Data files (.mdf/.ndf): Store data and objects**
* **Log file (.ldf): Records all transactions for recovery**
* **Schemas: Logical containers that group related objects**

**📋** What are Tables?

**Tables are the core data structures—rows and columns like a spreadsheet.**

**Each table has:**

* **Columns: Define data types and constraints**
* **Rows: Represent individual records**
* **Keys: Enforce uniqueness and relationships**

**> Example:**

**CREATE TABLE Customers (**

**CustomerID INT PRIMARY KEY,**

**Name NVARCHAR(100),**

**Email VARCHAR(255) UNIQUE,**

**JoinDate DATE DEFAULT GETDATE()**

**);**

**🗝️** Indexes: Accelerators for Your Queries

**An index is like a book’s table of contents—it helps SQL Server find data faster without scanning every row.**

**Types of Indexes:**

* **Clustered: Sorts table data by the index key. Only one per table.**
* **Non-clustered: Separate structure that points to the data rows.**
* **Columnstore: Ideal for analytics and large read-heavy tables.**
* **Filtered: Indexes only a subset of rows (e.g., active records).**

**> 🔍 *Tip:* Always examine execution plans before adding indexes—you don’t want to over-index and hurt insert/update performance.**

# Chapter 5: Data Types and Constraints

**Why this chapter matters:** Choosing the right data type and enforcing the right constraints ensures your database is efficient, reliable, and easier to maintain. It's like laying a strong foundation before building your house.

**🔤** Commonly Used Data Types

| **Category** | **Examples** | **Use Case** |
| --- | --- | --- |
| **Numeric** | INT, BIGINT, DECIMAL, FLOAT | For IDs, amounts, calculations |
| **String/Text** | VARCHAR, NVARCHAR, CHAR, TEXT | Names, emails, descriptions |
| **Date/Time** | DATE, DATETIME2, TIME | Timestamps, logging, scheduling |
| **Binary** | VARBINARY, IMAGE | File storage, images |
| **Other** | BIT, UNIQUEIDENTIFIER, XML, JSON | Flags, UUIDs, semi-structured data |

> ✨ *Pro Tip:* Always prefer VARCHAR(n) over TEXT and DATETIME2 over DATETIME for precision and flexibility.

**🧩 Constraints for Data Integrity**

Constraints define the rules SQL Server uses to enforce valid, consistent data.

* **PRIMARY KEY**: Ensures each row is uniquely identifiable.
* **FOREIGN KEY**: Maintains referential integrity between tables.
* **UNIQUE**: Guarantees all values in a column are different.
* **NOT NULL**: Prevents blank values in a column.
* **DEFAULT**: Sets a standard value when none is provided.
* **CHECK**: Enforces custom rules, like CHECK (Age >= 0)

> 🧠 *Remember:* Constraints are your best allies in reducing application bugs and keeping your data honest.

**🧠 Choosing the Right Data Type: Tips**

* Use the **smallest type** that meets your needs (e.g., SMALLINT instead of INT for small ranges).
* Match data types in **joins and predicates** to avoid implicit conversions.
* When storing currency, use DECIMAL(19,4) for accuracy—not FLOAT.

In the next chapter—**T-SQL Essentials**—we’ll start writing queries that bring your database to life. From SELECT basics to joins and subqueries, we’ll gear up for hands-on querying.

**Chapter 6: Querying with SELECT, INSERT, UPDATE, DELETE**

**Why This Matters:** CRUD operations—Create, Read, Update, Delete—are the heartbeat of all SQL interactions. Mastering these basics equips you to retrieve, manipulate, and maintain your data with precision and confidence.

**🔍 SELECT: Reading Data**

The SELECT statement retrieves data from one or more tables.

-- Get all rows and columns

SELECT \* FROM Employees;

-- Select specific columns

SELECT FirstName, LastName, HireDate FROM Employees;

-- Filtering rows

SELECT \* FROM Employees WHERE Department = 'Sales';

-- Sorting and limiting results

SELECT TOP 10 \* FROM Employees ORDER BY HireDate DESC;

> 🎯 *Tip:* Avoid SELECT \* in production—only retrieve the columns you need.

**🆕 INSERT: Adding New Records**

-- Insert a single row

INSERT INTO Employees (FirstName, LastName, Department)

VALUES ('Randy', 'Fadler', 'Engineering');

-- Insert multiple rows

INSERT INTO Departments (DeptName, ManagerID)

VALUES

('IT', 101),

('HR', 102);

**🔁 UPDATE: Changing Existing Records**

-- Change department for an employee

UPDATE Employees

SET Department = 'Operations'

WHERE EmployeeID = 3;

> 🧠 *Always* include a WHERE clause unless you want to update all rows!

**🗑️ DELETE: Removing Records**

-- Delete a specific employee

DELETE FROM Employees

WHERE EmployeeID = 5;

For safety, test deletes using SELECT first:

-- Preview rows to delete

SELECT \* FROM Employees WHERE EmployeeID = 5;

**🚧 Safe Practices**

* Wrap changes in a **transaction** if you’re making multiple updates:

BEGIN TRAN;

UPDATE ...;

DELETE ...;

COMMIT;

* Always back up critical data before running destructive commands.

# Chapter 7: Joins, Subqueries, and Set Operators

**Why this matters:** Relational databases are designed to relate! This chapter is about connecting the dots—efficiently and meaningfully.

**🤝** Joins: Combining Rows Across Tables

Joins let you pull data from two or more tables using a related column (often a foreign key).

**🔄** Types of Joins

| **Join Type** | **Description** |
| --- | --- |
| **INNER JOIN** | Returns matching rows from both tables |
| **LEFT JOIN** | All rows from the left table, and matches (or NULLs) from the right |
| **RIGHT JOIN** | All rows from the right table, and matches (or NULLs) from the left |
| **FULL JOIN** | All rows from both tables—matched and unmatched |
| **CROSS JOIN** | Cartesian product (all combinations!) |

> 💡 *Use* INNER JOIN *when you only want matching pairs. Switch to* LEFT JOIN *when you want to preserve unmatched rows from the left table.*

**Example:**

SELECT e.FirstName, e.LastName, d.DeptName

FROM Employees e

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

**🔎 Subqueries: Queries Inside Queries**

Subqueries let you compute values dynamically—either in the SELECT, WHERE, or FROM clauses.

**Types:**

* **Scalar Subquery**: Returns a single value
* **Correlated Subquery**: Depends on values from the outer query
* **Derived Table**: A subquery used in the FROM clause

**Example (Subquery in WHERE clause):**

SELECT FirstName, LastName

FROM Employees

WHERE Salary > (

SELECT AVG(Salary) FROM Employees

);

> 🧠 *Use subqueries for "give me the value from there that I need to compare here" scenarios.*

**⚖️ Set Operators: Merging Results**

Set operators combine the results of multiple queries.

| **Operator** | **Description** |
| --- | --- |
| **UNION** | Combines results, removes duplicates |
| **UNION ALL** | Combines all results (including duplicates) |
| **INTERSECT** | Only rows common to both queries |
| **EXCEPT** | Rows in the first query but not the second |

**Example:**

SELECT CustomerName FROM Customers\_US

UNION

SELECT CustomerName FROM Customers\_Canada;

In **Chapter 8: Functions, Expressions, and CASE Logic**, we’ll shape and transform data with built-in tools—perfect for creating calculated fields and conditional outputs.

# Chapter 9: Stored Procedures, Views, and Triggers

**🔁** Stored Procedures (SPs)

A **stored procedure** is a precompiled block of T-SQL code that you can execute with a single call.

**Benefits:**

* Encapsulation of logic
* Improved performance via execution plan reuse
* Centralized code maintenance

CREATE PROCEDURE GetHighValueOrders

@MinAmount DECIMAL

AS

BEGIN

SELECT OrderID, CustomerID, Amount

FROM Orders

WHERE Amount >= @MinAmount;

END;

-- Execute it

EXEC GetHighValueOrders @MinAmount = 500.00;

> ✨ *Tip:* Use parameters for flexibility, and wrap complex business logic in SPs to keep client code clean.

**👁️ Views**

A **view** is a virtual table based on the result of a SELECT query.

**Benefits:**

* Simplifies complex joins or filtering
* Improves security by limiting exposed columns
* Useful for abstraction and consistent querying

CREATE VIEW vw\_ActiveCustomers AS

SELECT CustomerID, Name, Email

FROM Customers

WHERE IsActive = 1;

**You can then query it like a table:**

SELECT \* FROM vw\_ActiveCustomers;

> 🔍 *Note:* Views don't store data—they’re re-evaluated on each query. Use **indexed views** if persistent performance is required.

**⚡ Triggers**

A **trigger** is special stored procedure that automatically runs in response to an event (INSERT, UPDATE, or DELETE).

**Common Use Cases:**

* Enforcing audit trails
* Validating business rules
* Logging or notification logic

sql

CREATE TRIGGER trg\_LogOrderInsert

ON Orders

AFTER INSERT

AS

BEGIN

INSERT INTO OrderAudit (OrderID, InsertedAt)

SELECT OrderID, GETDATE()

FROM inserted;

END;

> 🚧 *Use triggers with care!* They can introduce hidden performance costs and unintended side effects.

Next up in **Chapter 10**, we’ll wrap this section with **Transactions and Error Handling**—perfect for ensuring data integrity and catching problems before they spiral.

# Chapter 11: System-Versioned Temporal Tables

**📌** What Are Temporal Tables?

A **system-versioned temporal table** automatically keeps full history of all changes to a table's data. Instead of manually tracking changes with triggers or audit tables, SQL Server does the heavy lifting.

Each versioned row includes a **valid time window**:

* ValidFrom (start time)
* ValidTo (end time)

> When a row is updated or deleted, SQL Server moves the old version to a **history table**—automatically!

**🧱** Creating a Temporal Table

CREATE TABLE Department (

DeptID INT PRIMARY KEY,

DeptName VARCHAR(100),

ValidFrom DATETIME2 GENERATED ALWAYS AS ROW START NOT NULL,

ValidTo DATETIME2 GENERATED ALWAYS AS ROW END NOT NULL,

PERIOD FOR SYSTEM\_TIME (ValidFrom, ValidTo)

)

WITH (SYSTEM\_VERSIONING = ON);

SQL Server will auto-create a matching history table like MSSQL\_TemporalHistoryFor\_<object\_id> unless you specify your own.

**🔄 Modifying an Existing Table**

Already have a table? Just alter it to become temporal:

ALTER TABLE Employee

ADD ValidFrom DATETIME2 GENERATED ALWAYS AS ROW START NOT NULL

CONSTRAINT DF\_ValidFrom DEFAULT SYSUTCDATETIME(),

ValidTo DATETIME2 GENERATED ALWAYS AS ROW END NOT NULL

CONSTRAINT DF\_ValidTo DEFAULT '9999-12-31 23:59:59.9999999',

PERIOD FOR SYSTEM\_TIME (ValidFrom, ValidTo);

ALTER TABLE Employee

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

**🔍 Querying with Time Travel**

You can query temporal tables with FOR SYSTEM\_TIME to explore changes:

-- See how a row looked on June 1st

SELECT \* FROM Department

FOR SYSTEM\_TIME AS OF '2025-06-01'

WHERE DeptID = 2;

-- Find all past versions

SELECT \* FROM Department

FOR SYSTEM\_TIME ALL

WHERE DeptID = 2;

Other options:

* FROM/TO – get rows active in a date range
* BETWEEN/AND – alias for FROM/TO
* CONTAINED IN – get rows fully contained within a range

**🧠 Why Use Temporal Tables?**

* **Auditing**: Track who changed what and when
* **Data recovery**: Revert accidental deletes/updates
* **Slowly changing dimensions**: Ideal for data warehouses
* **Simplified logic**: No need for custom history triggers

**⚠️ Best Practices**

* Name your history table explicitly
* Add indexes to history tables for large datasets
* Avoid manually modifying the history table—it’s managed by the system
* Monitor growth and consider archiving old history

# Chapter 12: Partitioning Strategies and Implementation

**Why Partitioning Matters:** As tables grow into millions—or billions—of rows, partitioning helps distribute data across manageable chunks called *partitions*, improving query performance, maintenance, and scalability.

**🧱** What Is Table Partitioning?

Partitioning splits a large table horizontally based on the value of a column (e.g., OrderDate, RegionID, TransactionYear). Each "slice" is stored separately, yet you can still query the table as a whole.

> ⚡ Use cases: archival of old data, reporting over recent years, or sharding large transactional tables.

**🗺️** Key Components of Partitioning

1. **Partition Function** Defines how the table is logically divided.

CREATE PARTITION FUNCTION pf\_YearRange (INT)

AS RANGE LEFT FOR VALUES (2022, 2023, 2024);

1. **Partition Scheme** Maps partitions to physical filegroups (optional for basic usage).

CREATE PARTITION SCHEME ps\_YearRange

AS PARTITION pf\_YearRange

TO ([PRIMARY], [FG\_2023], [FG\_2024], [FG\_2025]);

1. **Partitioned Table** Table is created on the scheme, automatically routing rows based on partitioning column.

CREATE TABLE Sales (

SaleID INT PRIMARY KEY,

SaleYear INT,

Amount MONEY

) ON ps\_YearRange (SaleYear);

**🔍** Query Behavior

SQL Server applies **partition elimination** to skip scanning irrelevant partitions. Example:

SELECT \* FROM Sales WHERE SaleYear = 2024;

> SQL Server touches only the 2024 partition—faster reads, smaller scans.

**🧹 Maintenance Benefits**

* **Switch partitions out** for quick archival
* **Truncate by partition** instead of deleting millions of rows
* Perform **index rebuilds** on a single partition

**🧠** Best Practices and Considerations

* Choose a partition key that's **frequently filtered in queries**
* Monitor skew—**don’t let one partition dominate the data**
* Keep your partition boundaries aligned with business logic (e.g., monthly or yearly)

**📚** Real-World Reference: Wide World Importers

The Wide World Importers sample includes partitioned tables that support both OLTP and real-time analytics workloads. Great sandbox to test out partitioning and query tuning patterns.

In Chapter 13, we’ll turn our focus to **Dynamic SQL and Execution Plans**—where performance meets flexibility!

# Chapter 13: Dynamic SQL and Execution Plans

**🔄 What Is Dynamic SQL?**

**Dynamic SQL** is T-SQL code that’s constructed and executed at runtime—often as a string—allowing flexible filtering, object targeting, or automation.

**✅ Use Cases:**

* Building search filters based on optional inputs
* Running commands across multiple tables or databases
* Automating maintenance or code generation

**Example:**

DECLARE @sql NVARCHAR(MAX);

DECLARE @tableName SYSNAME = 'Orders';

SET @sql = 'SELECT \* FROM ' + QUOTENAME(@tableName);

EXEC sp\_executesql @sql;

> ✨ *Tip:* Always validate and sanitize parameters to avoid SQL injection—especially when working with user inputs.

**⚙️** sp\_executesql vs. EXEC

| **Method** | **Benefits** | **Limitations** |
| --- | --- | --- |
| sp\_executesql | Supports parameters, plan reuse | Slightly more verbose |
| EXEC | Simpler syntax | No plan reuse, no parameters |

**Example with parameters:**

DECLARE @sql NVARCHAR(MAX), @MinAmount MONEY = 500;

SET @sql = 'SELECT \* FROM Orders WHERE Amount > @amt';

EXEC sp\_executesql @sql, N'@amt MONEY', @MinAmount;

**🔍** Execution Plans: Reading the Roadmap

An **execution plan** shows how SQL Server processes a query—step by step.

**Two types:**

* **Estimated Plan**: No execution—just the blueprint.
* **Actual Plan**: Shows what actually happened (row counts, operators).

**How to View:**

* In SSMS: Click **“Include Actual Execution Plan”** or use Ctrl + M
* Run your query, then switch to the **Execution Plan** tab

**🧩 Understanding Plan Components**

| **Icon** | **Operator** | **Meaning** |
| --- | --- | --- |
| 🔍 | Index Seek | Efficient lookup via index |
| 📚 | Table Scan | Full scan of table (may be slow!) |
| 🔁 | Nested Loops | Simple row-by-row joining |
| ♻️ | Hash Match | Good for large sets, but memory intensive |
| 🔄 | Merge Join | Efficient for sorted inputs |

> 💡 *Hover over arrows* to see row counts and costs. Pay attention to estimates vs. actual—big mismatches suggest outdated statistics.

**🧠 Pro Tips**

* Use **parameterized dynamic SQL** for better performance and plan reuse
* Review **“Missing Index” suggestions** in plans—but don’t add blindly
* **Compare plans** before and after tuning queries or adding indexes
* Use SET STATISTICS IO ON and SET STATISTICS TIME ON for deeper insight

In **Chapter 14**, we’ll explore **Sequences and Identity Columns**—auto-generating keys and handling gaps or concurrency.

# Chapter 14: SQL Server Sequences and Identity Columns

**🆔 Identity Columns**

**Identity columns** auto-generate numeric values when rows are inserted—commonly used for primary keys.

**Basic Syntax:**CREATE TABLE Orders (

OrderID INT IDENTITY(1,1) PRIMARY KEY,

CustomerID INT,

OrderDate DATE

);

* IDENTITY(1,1) starts at 1 and increments by 1.
* Values are automatically generated—no need to specify them during inserts.

**Identity Notes:**

* Use DBCC CHECKIDENT('Orders') to check or reseed.
* Gaps can occur (e.g., from rolled-back transactions).
* Only one identity column per table.

**🔄 Sequences**

Introduced in SQL Server 2012, **sequences** are independent objects that generate a series of numbers—you control when and how they're used.

**Benefits Over Identity:**

* Can be shared across tables
* No gaps from rollbacks (if used carefully)
* More flexible for batching and parallelism

**Creating and Using a Sequence:**

CREATE SEQUENCE OrderSeq

START WITH 1000

INCREMENT BY 5;

-- Get the next value

SELECT NEXT VALUE FOR OrderSeq;

-- Insert using the sequence

INSERT INTO Orders (OrderID, CustomerID, OrderDate)

VALUES (NEXT VALUE FOR OrderSeq, 101, '2025-06-19');

**Resetting or Altering:**

ALTER SEQUENCE OrderSeq RESTART WITH 500;

**🤝 Choosing Between Identity and Sequence**

| **Feature** | **Identity Column** | **Sequence Object** |
| --- | --- | --- |
| Tied to table | Yes | No (separate object) |
| Gaps on rollback | Likely | Avoidable |
| Parallel inserts | Limited control | Better support |
| Used in multiple tables | No | Yes |

> ✨ *Use Identity for simple cases; Sequence when you need control across tables or parallel inserts.*

In **Chapter 15**, we’ll compare **Change Data Capture and Change Tracking**—two different but equally powerful ways to monitor changes in your data over time.

# Chapter 15: Change Data Capture vs. Change Tracking

**🧭** Why Capture Changes?

Whether you're syncing with external systems, replicating changes, or auditing updates, it's crucial to know **what changed and when**—without rolling your own triggers or poll-based logic.

**📸** Change Data Capture (CDC)

**CDC** records row-level changes to tracked tables in associated change tables using **transaction log mining**.

**Key Features:**

* Captures **INSERT**, **UPDATE**, and **DELETE**
* Shows **before-and-after values** for updated rows
* Stores changes in SQL-managed tables (e.g., cdc.dbo\_TableName\_CT)
* Uses **asynchronous log scanning**
* Ideal for **ETL pipelines**, **auditing**, and **data replication**

**Enabling CDC:**

EXEC sys.sp\_cdc\_enable\_db; -- One-time for the database

EXEC sys.sp\_cdc\_enable\_table

@source\_schema = 'dbo',

@source\_name = 'Orders',

@role\_name = NULL;

**Querying Changes:**

SELECT \*

FROM cdc.fn\_cdc\_get\_all\_changes\_dbo\_Orders(

@from\_lsn, @to\_lsn, 'all');

> 🧠 *LSNs (Log Sequence Numbers)* are used to track changes over time.

**🧵 Change Tracking (CT)**

**CT** offers a lightweight way to track **which rows changed**, but not how.

**Key Features:**

* Tracks **INSERT, UPDATE, DELETE**, but only row versions
* **Doesn't store previous values**
* Minimal overhead
* Requires consumer to track a **version number (CHANGE\_VERSION)**
* Ideal for **apps syncing changes**, like mobile clients or caching layers

**Enabling CT:**

ALTER DATABASE YourDB SET CHANGE\_TRACKING = ON

(CHANGE\_RETENTION = 2 DAYS, AUTO\_CLEANUP = ON);

ALTER TABLE Orders ENABLE CHANGE\_TRACKING

WITH (TRACK\_COLUMNS\_UPDATED = ON);

**Querying Changes:**

SELECT \*

FROM CHANGETABLE(CHANGES Orders, @last\_sync\_version) AS CT;

**⚖️ Comparison Table**

| **Feature** | **CDC** | **Change Tracking** |
| --- | --- | --- |
| Captures data | Yes, full before/after | No, only row-level change |
| Query complexity | Higher (via log + LSNs) | Simpler |
| Performance | Medium | High (lightweight) |
| Use case | Auditing, ETL, DW loads | Syncing apps, lightweight change check |
| Retention | Based on cleanup jobs | Based on retention setting |

**✨ Choosing the Right Tool**

* Use **CDC** when you need full change history, including old and new values.
* Use **Change Tracking** when you only need to know *what* changed, not *how*.

In **Chapter 16**, we’ll zoom into the nitty-gritty of **Indexing**, including when and how to use clustered, non-clustered, and columnstore indexes for performance tuning.

# Chapter 16: Indexing Deep Dive—Clustered, Non-Clustered, and Columnstore

**🔍** Why Indexing Matters

Indexes allow SQL Server to locate data efficiently—without scanning every row in a table. They’re essential for performance, especially in read-heavy workloads.

> 📌 *Think of indexes like the index in a book: you can quickly jump to what you need instead of flipping every page.*

**🧱** Clustered Index

* **Defines the physical order of data in the table**
* There can be **only one** per table
* Often set on the primary key

CREATE CLUSTERED INDEX IX\_Employees\_EmployeeID

ON Employees(EmployeeID);

> 🧠 *The table* ***is*** *the index. Efficient for range queries and sorting.*

**🧾 Non-Clustered Index**

* **Separate structure** from the table with pointers to rows
* You can have **many** non-clustered indexes per table
* Can include **included columns** for covering queries

CREATE NONCLUSTERED INDEX IX\_Employees\_DeptID

ON Employees(DeptID)

INCLUDE (FirstName, LastName);

> 💡 *Use INCLUDE to avoid lookups and return all needed data from the index alone.*

**🧱 vs. 🧾 Comparison**

| **Feature** | **Clustered Index** | **Non-Clustered Index** |
| --- | --- | --- |
| Physical data order | Yes | No |
| Quantity per table | 1 | Many |
| Great for | Range scans, ordering | Filtered lookups, covering |

📦 Columnstore Index

Designed for **analytics and warehousing**, Columnstore indexes store data **column by column** instead of row by row.

**Benefits:**

* Amazing compression
* Blazing speed for aggregations over millions of rows
* Ideal for OLAP, fact tables, large scans

CREATE CLUSTERED COLUMNSTORE INDEX IX\_Sales\_Columnstore

ON Sales;

> 🚀 *Massive performance boost for read-heavy analytics—less ideal for heavy OLTP.*

🔧 Filtered Indexes

Only index a **subset of rows**, reducing size and overhead.

CREATE NONCLUSTERED INDEX IX\_Employees\_ActiveOnly

ON Employees(IsActive)

WHERE IsActive = 1;

> 🧠 *Great for indexes on* IsDeleted*,* Status = 'Active'*, etc.*

**📉 When Indexes Hurt**

* Too many = higher storage, slower INSERT/UPDATE/DELETE
* Wrong indexes = unused by optimizer
* Missing stats = poor query plans

Use:

SELECT \* FROM sys.dm\_db\_missing\_index\_details;

to identify opportunities—but always verify.

In **Chapter 17**, we’ll analyze **Execution Plans and Query Tuning**—perfect timing after this indexing foundation.

# Chapter 17: Execution Plans and Query Tuning

**🧭** What Are Execution Plans?

An **execution plan** is a roadmap of how SQL Server processes your query. It shows **which indexes were used**, **join strategies**, and **estimated vs. actual row counts**.

**Types of Plans:**

* **Estimated Plan**: Generated before running the query
* **Actual Plan**: Collected during execution, includes real metrics

> 📌 *Use* Ctrl + M *in SSMS before running a query to include the actual execution plan.*

**🔍** Reading Execution Plans

Key elements to look for:

| **Icon** | **Operator** | **What It Tells You** |
| --- | --- | --- |
| 🔍 | Index Seek | Efficient lookup—ideal |
| 📚 | Table Scan | Slow full scan—missing index or filter |
| 🔁 | Nested Loops | Best for small row sets |
| ♻️ | Hash Match | Efficient for large joins |
| 🔄 | Merge Join | Great for pre-sorted data |

> 🧠 *Hover over arrows to compare “Estimated Rows” vs. “Actual Rows”. Big mismatches = outdated stats or bad estimates.*

🧩 Common Tuning Techniques

1. **Add Missing Indexes** Use sys.dm\_db\_missing\_index\_details to discover helpful suggestions—but **verify usefulness** first!
2. **Use Covering Indexes** Add INCLUDE columns so the query can be satisfied entirely from the index.
3. **Avoid Functions on Indexed Columns**

WHERE YEAR(OrderDate) = 2025 -- ❌ kills index

WHERE OrderDate >= '2025-01-01' AND OrderDate < '2026-01-01' -- ✅ index-friendly

1. **Update Statistics** Keeps row estimates accurate:

UPDATE STATISTICS dbo.Orders;

1. **Use SARGable Predicates** “Search Argument Able” = SQL can use indexes
2. **Simplify Joins and Subqueries** Rewrite complex joins and avoid correlated subqueries in favor of joins or temp tables.

**🛠️ Tools for Tuning**

* **SSMS Execution Plan Viewer** Interactive, shows row flow and operator costs.
* **Query Store** Captures query history, plans, and performance stats over time.
* **DMVs (Dynamic Management Views)** Analyze cache, indexes, and memory grants.

Example:

SELECT TOP 5 \*

FROM sys.dm\_exec\_query\_stats

ORDER BY total\_worker\_time DESC;

**Built-In SQL Server Tools (Already Included with Installation)**

| **Tool** | **Purpose** |
| --- | --- |
| **SQL Server Management Studio (SSMS)** | Central hub for query tuning, execution plans, and index management. |
| **Execution Plans** | Visualize query performance bottlenecks (use *Actual Execution Plan* in SSMS). |
| **Dynamic Management Views (DMVs)** | Query system views like sys.dm\_exec\_query\_stats, sys.dm\_db\_index\_usage\_stats for performance insights. |
| **Database Engine Tuning Advisor (DTA)** | Analyzes workloads and recommends indexes, partitions, and stats updates. |
| **Query Store** | Tracks query performance over time, helps identify regressions and force plans. |
| **Extended Events** | Lightweight tracing for deep diagnostics (replaces SQL Profiler). |
| **Live Query Statistics** | See query execution in real time—great for long-running queries. |
| **Activity Monitor** | Quick overview of CPU, I/O, expensive queries, and blocking sessions. |
| **DBCC Commands** | Use DBCC SHOW\_STATISTICS, DBCC FREEPROCCACHE, etc., for tuning and diagnostics. |

**🧰 Optional Tools You Might Want to Install**

| **Tool** | **Description** |
| --- | --- |
| **SQL Server Profiler** | Legacy tool for tracing events—still useful for some scenarios. |
| **Performance Monitor (PerfMon)** | Windows tool to track SQL Server counters like buffer cache hit ratio, page life expectancy. |
| **Azure Data Studio** | Lightweight alternative to SSMS with extensions for performance insights. |
| **Third-Party Tools** | Redgate SQL Monitor, SolarWinds DPA, and ApexSQL Plan offer advanced diagnostics and visualizations. |

In **Chapter 18**, we’ll dig deeper into **Statistics, Histograms, and Cardinality Estimation**—the math behind execution plans.

# Chapter 18: Statistics, Histograms, and Cardinality Estimation

**🧠** Why Statistics Matter

SQL Server uses statistics to **estimate row counts** during query planning. These estimates impact everything from join strategies to index selection. If the stats are wrong, the plan will be too.

**🔍** What Are Statistics?

Statistics are metadata that describe the **distribution of values** in one or more columns of a table or index.

They’re built using:

* **Histograms**: Bucketed value frequency maps
* **Density vectors**: Measure how unique the values are
* **Cardinality**: The expected number of rows a predicate returns

> 📌 *You can view a table’s statistics with:*

DBCC SHOW\_STATISTICS ('Products', 'IX\_Products\_CategoryID');

**📈 Histograms**

A histogram summarizes the distribution of values in a single column—like a bar chart behind the scenes.

* Up to **200 steps** (bins)
* Shows **range\_high\_key**, **equal\_rows**, and **distinct\_range\_rows**
* Built from a **sample** of rows (unless FULLSCAN is specified)

> 🧠 *Useful for estimating filters like* WHERE Price = 9.99 *or* WHERE OrderDate BETWEEN '2024-01-01' AND '2024-12-31'

🔮 Cardinality Estimation (CE)

The **Cardinality Estimator** is the engine that predicts how many rows a query will return—based on available statistics.

**Why it matters:**

* **Bad estimates = bad plans**
* **Good estimates = efficient joins, correct index choices**

**Example:**

SELECT \* FROM Orders WHERE CustomerID = 42;

If SQL estimates 1 row and gets 1,000, it might use a Nested Loop join instead of a more efficient Hash Match.

**🔄 Updating Statistics**

Stats can become stale due to data changes.

Options:

-- Rebuild one stat

UPDATE STATISTICS dbo.Customers;

-- Rebuild all

EXEC sp\_updatestats;

You can also enable **AUTO\_UPDATE\_STATISTICS** (default) and control thresholds.

**🔎 Viewing Statistics Usage**

Use this DMV to find recently used stats:

SELECT \*

FROM sys.dm\_db\_stats\_properties(object\_id('Orders'), stats\_id)

Or explore column-level usage:

SELECT \* FROM sys.stats WHERE object\_id = OBJECT\_ID('Orders');

**⚠️ When Statistics Go Wrong**

* Skewed distributions (e.g. 90% NULL, 10% active)
* Multi-column filtering with only single-column stats
* Filtered or computed columns lacking coverage

> ✨ *Tip:* For complex queries, consider creating **filtered stats** or **manual stats on computed expressions**\*.

In **Chapter 19**, we’ll take all this knowledge and sharpen it into **best practices for writing high-performance queries**.

# Chapter 19: Best Practices for Writing High-Performance Queries

**🧠 1.** Be Selective in SELECT

Only return the columns you need—avoid SELECT \*.

-- ✅ Better

SELECT FirstName, LastName FROM Employees;

-- ❌ Avoid

SELECT \* FROM Employees;

> Reduces I/O, improves index usage, and shrinks result sets.

**🧩 2.** Filter Early, Filter Often

Always apply WHERE clauses when possible—and push predicates down.

-- Good: filters early

SELECT \* FROM Orders WHERE OrderDate >= '2024-01-01';

-- Bad: filters after joining a huge set

SELECT \* FROM Orders JOIN Customers ON ... WHERE OrderDate >= ...

**🧮 3.** Mind Your Joins

Use the right join for the job:

* INNER JOIN for matching rows
* LEFT JOIN only when needed
* Avoid CROSS JOIN unless you *want* a Cartesian product

Check indexing on join keys and match data types.

**📚 4.** Watch for Table Scans

If your query is scanning large tables, check for:

* Missing indexes
* Non-sargable filters
* Outdated stats

Use **execution plans** to catch these silently expensive operations.

**📉 5.** Avoid Row-by-Row (RBAR) Processing

SQL Server thrives on set-based logic—not loops.

Instead of:

DECLARE cursor CURSOR FOR SELECT ...;

OPEN cursor;

FETCH NEXT FROM cursor...

Try:

UPDATE Customers

SET Status = 'Inactive'

WHERE LastLogin < '2024-01-01';

**🏗️ 6.** Break Down Complex Queries

Long queries with nested logic can confuse the optimizer. Consider:

* Using temp tables or CTEs
* Materializing steps instead of deeply nested subqueries

**🔍 7.** Leverage Covering Indexes

If you’re always filtering and selecting the same columns, build a covering index using INCLUDE.

CREATE INDEX IX\_Orders\_Covering

ON Orders (CustomerID)

INCLUDE (OrderDate, Amount);

**🧰 8.** Use Proper Data Types

Match types in joins and filters, and choose the smallest data type that meets your needs.

-- Implicit conversion slows query:

WHERE PhoneNumber = 123456789

-- Better:

WHERE PhoneNumber = '123456789'

**🔄 9.** Keep Stats and Indexes Fresh

Schedule regular updates:

EXEC sp\_updatestats;

ALTER INDEX ALL ON Customers REBUILD;

Monitor fragmentation, stale plans, and I/O.

**🧪 10.** Always Test Against Real Data

Small dev tables may hide performance bottlenecks. Use representative volumes and patterns when benchmarking.

In **Chapter 20**, we’ll pull out the power tools—**DMVs and DMFs**—for advanced performance monitoring and diagnostics.

🔍 Chapter 20: Using DMVs and DMFs for Monitoring

**DMV = Dynamic Management View** **DMF = Dynamic Management Function** Together, they expose **real-time server internals** to help DBAs and developers monitor everything from query stats to memory usage.

**🧭 Key DMV Categories**

| **DMV Category** | **Description** |
| --- | --- |
| sys.dm\_exec\_\* | Query execution, SQL text, and plans |
| sys.dm\_db\_\* | Per-database info—indexes, stats, storage |
| sys.dm\_os\_\* | Operating system-level info—CPU, memory |
| sys.dm\_tran\_\* | Transactions and locking |
| sys.dm\_io\_virtual\_file\_stats | I/O patterns per file |

**📈 Useful DMVs in Action**

**1.** Top 10 Most Expensive Queries

SELECT TOP 10

qs.total\_elapsed\_time / qs.execution\_count AS avg\_time\_ms,

qs.execution\_count,

qt.text

FROM sys.dm\_exec\_query\_stats qs

CROSS APPLY sys.dm\_exec\_sql\_text(qs.sql\_handle) qt

ORDER BY avg\_time\_ms DESC;

> Find slow queries to target for tuning.

**2.** Missing Indexes

SELECT

migs.avg\_user\_impact,

mid.equality\_columns,

mid.inequality\_columns,

mid.included\_columns

FROM sys.dm\_db\_missing\_index\_group\_stats migs

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

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

ORDER BY migs.avg\_user\_impact DESC;

> Find potentially useful indexes—verify before implementing!

**3.** Wait Statistics

SELECT TOP 10 \*

FROM sys.dm\_os\_wait\_stats

ORDER BY wait\_time\_ms DESC;

> Understand where SQL Server is “waiting” (e.g., I/O, locks, CPU).

**4.** Active Sessions and Blocking

SELECT

session\_id, status, blocking\_session\_id, wait\_type, last\_request\_start\_time

FROM sys.dm\_exec\_requests

WHERE blocking\_session\_id <> 0;

> Pinpoint performance bottlenecks and blocking chains.

**5.** Disk I/O by File

SELECT DB\_NAME(database\_id) AS db,

file\_id,

io\_stall,

num\_of\_reads,

num\_of\_writes

FROM sys.dm\_io\_virtual\_file\_stats(NULL, NULL);

> Analyze hot files and storage lag.

**🔧 Pro Tips**

* **Filter** by database using DB\_ID() or DB\_NAME() functions
* DMVs reset on **SQL Server restart**—log results if long-term trends are needed
* Wrap DMV queries into **stored procedures or dashboards** for reuse

In Chapter 21, we explored **backups and recovery**. Coming up in **Chapter 22**, we’ll lock down your environment with **Role-Based Access and Security Best Practices**.

# Chapter 21: Backup, Restore, and Recovery Models

**🔐 Why Backups Matter**

Backups protect your data from corruption, user error, hardware failure, or catastrophic events. But not all backups (or recovery plans) are created equal.

**🔄 Recovery Models Overview**

The **recovery model** determines how much data can be recovered and what happens to the transaction log.

| **Model** | **Log Behavior** | **Use Case** |
| --- | --- | --- |
| **Full** | Logs every transaction | Point-in-time recovery |
| **Bulk-Logged** | Like Full, but logs minimally for bulk ops | Efficient for big imports |
| **Simple** | Truncates log automatically | Dev/test, non-critical |

> 💡 *Use* SELECT name, recovery\_model\_desc FROM sys.databases; *to check current settings.*

**💽 Backup Types**

**1.** Full Backup

Captures the entire database.

BACKUP DATABASE WideWorldImporters

TO DISK = 'C:\Backups\WWI\_Full.bak';

**2.** Differential Backup

Captures only changes since the last full backup.

BACKUP DATABASE WideWorldImporters

TO DISK = 'C:\Backups\WWI\_Diff.bak'

WITH DIFFERENTIAL;

**3.** Transaction Log Backup

Allows point-in-time recovery in Full or Bulk-Logged models.

BACKUP LOG WideWorldImporters

TO DISK = 'C:\Backups\WWI\_Log.trn';

**♻️** Restore Scenarios

**Full Restore:**

RESTORE DATABASE WideWorldImporters

FROM DISK = 'C:\Backups\WWI\_Full.bak'

WITH NORECOVERY;

Followed by Differential**:**

RESTORE DATABASE WideWorldImporters

FROM DISK = 'C:\Backups\WWI\_Diff.bak'

WITH NORECOVERY;

**Then Transaction Log:**

RESTORE LOG WideWorldImporters

FROM DISK = 'C:\Backups\WWI\_Log.trn'

WITH RECOVERY;

> 🛠️ *Use* WITH FILE = n *if restoring from a media set with multiple backups.*

**🧪 Best Practices**

* Automate backups using **SQL Server Agent jobs**.
* Store backups offsite or in the cloud (e.g., Azure Blob).
* Regularly test restores—**a backup is only as good as its recovery**.
* Monitor backup job success and alert on failures.

In **Chapter 22**, we’ll secure the database engine with **Role-Based Access, Logins, and Permission Management**.

# Chapter 22: Security and Role-Based Access

**🧱** Authentication Modes

SQL Server supports two main authentication models:

* **Windows Authentication** (integrated security): Trusted, uses domain credentials
* **SQL Server Authentication**: Uses separate login/password—use sparingly and always with strong password policies

-- Enable mixed mode (if needed) via SQL Server Configuration Manager

**👥** Logins vs. Users

* **Login**: Authenticates you to the server (instance-level)
* **User**: Grants access to a specific database

-- Create login

CREATE LOGIN FadlerUser WITH PASSWORD = 'StrongP@ssword2025!';

-- Map login to a user in your database

USE YourDatabase;

CREATE USER FadlerUser FOR LOGIN FadlerUser;

**🛡️** Fixed Server Roles (Instance-Level)

| **Role** | **Privilege Scope** |
| --- | --- |
| sysadmin | Full control |
| securityadmin | Manage logins, roles |
| serveradmin | Configure server-wide settings |
| diskadmin, setupadmin, etc. | Specialized tasks |

> 🧠 *Use* sysadmin *sparingly—assign minimum required privileges!*

**🧩** Database Roles

SQL Server provides fixed **database roles** to control common access levels:

| **Role** | **Permissions** |
| --- | --- |
| db\_datareader | Can SELECT from all tables and views |
| db\_datawriter | Can INSERT, UPDATE, DELETE |
| db\_ddladmin | Can run DDL commands (CREATE/ALTER objects) |
| db\_owner | Full control within that database |

-- Add user to datareader role

ALTER ROLE db\_datareader ADD MEMBER FadlerUser;

**🧬** Custom Roles and Fine-Grained Permissions

You can create **application roles** or custom database roles to group specific privileges.

CREATE ROLE ReportingUser;

GRANT SELECT ON dbo.Sales TO ReportingUser;

ALTER ROLE ReportingUser ADD MEMBER FadlerUser;

> 🔐 *Principle of least privilege: give only what’s necessary, nothing more.*

**🧪 Audit Tips**

* Regularly audit:

SELECT \* FROM sys.database\_role\_members;

SELECT \* FROM sys.server\_principals WHERE type\_desc = 'SQL\_LOGIN';

* Use built-in auditing or extended events for tracking login activity
* Consider enabling **Contained Databases** for better user management across environments

Coming up next in **Chapter 23**, we’ll automate repeatable tasks and maintenance with **SQL Server Agent**.

# Chapter 23: Automating Jobs with SQL Server Agent

**⚙️** What Is SQL Server Agent?

**SQL Server Agent** is a built-in Windows service that runs scheduled jobs, alerts, and automation scripts on your SQL Server instance.

* Supports **T-SQL, SSIS, PowerShell, CmdExec**, and more
* Integrated into SSMS for easy GUI setup
* Essential for **daily maintenance, backups, indexing, alerts**, etc.

> 🧠 *Make sure the Agent service is started—check in SQL Server Configuration Manager or SSMS Object Explorer.*

**🧩**Key Components

| **Component** | **Description** |
| --- | --- |
| **Job** | A container for one or more steps |
| **Step** | A single task (T-SQL, PowerShell, etc.) |
| **Schedule** | Defines when the job runs |
| **Operator** | Alert contact for job success/failure |

**✅ Creating a Job (SSMS GUI)**

1. Expand SQL Server Agent → Jobs → Right-click → **New Job**
2. Add a **name** and **description**
3. Add **steps** with T-SQL or executable actions
4. Set up a **schedule** (recurring, one-time, etc.)
5. Assign **alerts** or **notifications** (email, log, etc.)

**💡 T-SQL Example: Create Job via Script**

sql

USE msdb;

EXEC sp\_add\_job @job\_name = 'Nightly Index Rebuild';

EXEC sp\_add\_jobstep

@job\_name = 'Nightly Index Rebuild',

@step\_name = 'Rebuild All Indexes',

@subsystem = 'TSQL',

@command = 'EXEC sp\_MSforeachtable ''ALTER INDEX ALL ON ? REBUILD''';

EXEC sp\_add\_schedule

@schedule\_name = 'EveryMidnight',

@freq\_type = 4, -- daily

@active\_start\_time = 000000; -- midnight

EXEC sp\_attach\_schedule

@job\_name = 'Nightly Index Rebuild',

@schedule\_name = 'EveryMidnight';

EXEC sp\_add\_jobserver @job\_name = 'Nightly Index Rebuild';

**🔔** Notifications & Alerts

Set up **Database Mail** and configure SQL Server Agent to notify:

* On job success/failure
* On specific error levels (e.g., severity 17+)

EXEC msdb.dbo.sp\_add\_notification

@job\_name = 'Nightly Index Rebuild',

@level = 1,

@operator\_name = 'DBA\_Operator',

@notification\_method = 1; -- Email

**🧠** Best Practices

* Use **descriptive job names** and document job logic
* Log job history and step output for troubleshooting
* Monitor Agent health—configure alerts for job failures
* Store job code in source control if you’re scripting your jobs

In **Chapter 24**, we’ll explore **Database Maintenance Plans**—including backups, stats updates, and consistency checks in a GUI-friendly way.

# Chapter 24: Database Maintenance Plans

**🧭** What Is a Maintenance Plan?

A **Maintenance Plan** is a collection of tasks—like backups, index maintenance, or consistency checks—automated through SQL Server Agent. It uses a friendly GUI to help DBAs set up workflows without heavy scripting.

> 💡 Perfect for small- to mid-sized environments, or when managing multiple servers without third-party tools.

**🧩 Common Tasks in a Maintenance Plan**

| **Task** | **Purpose** |
| --- | --- |
| **Check Database Integrity** | Verifies logical and physical structure |
| **Rebuild/Reorganize Indexes** | Improves performance and storage efficiency |
| **Update Statistics** | Keeps query plans accurate |
| **Back Up Database (Full/Diff/Log)** | Ensures recovery options |
| **Clean Up History** | Removes old job logs, backups, or text files |
| **Execute T-SQL Statements** | Run custom scripts |

**🧱 Creating a Maintenance Plan (via SSMS)**

1. In SSMS → Object Explorer → *Management* → **Maintenance Plans**
2. Right-click → **New Maintenance Plan**
3. Use the **Designer** to drag & drop tasks:
   * Sequence them using connectors
   * Set properties like databases, schedules, thresholds
4. Use **Subplans** for modularity (e.g., one for backups, one for index work)
5. Review the SQL Server Agent job that gets created behind the scenes

**🔄 Example: Weekly Index Maintenance Plan**

* **Monday, 2 AM**: Check Database Integrity
* **Tuesday, 3 AM**: Rebuild Indexes + Update Statistics
* **Wednesday, 2 AM**: Full Database Backup
* **Daily**: Transaction Log Backup + Clean Up Old Files

**🧠 Best Practices**

* Use **“Rebuild Index”** for heavily fragmented tables, **“Reorganize”** for light fragmentation
* Always **update statistics** after index operations
* Configure backup tasks to **verify integrity** and optionally compress
* Set retention policies to avoid bloating your backup folders
* **Schedule during off-hours** to reduce user impact

In Chapter 25, we’ll turn to **High Availability and Disaster Recovery (HADR)** options like Always On and log shipping—key for mission-critical systems.

# Chapter 25: High Availability and Disaster Recovery (HADR)

**🚨 Why HADR Matters**

Failures happen—disk crashes, network drops, power outages. HADR strategies ensure your SQL Server instance stays online or recovers quickly, minimizing business disruption.

**🧰 Key HADR Technologies**

| **Feature** | **Description** | **Use Case** |
| --- | --- | --- |
| **Failover Cluster Instance (FCI)** | Shared storage cluster with automatic failover | On-prem HA |
| **Always On Availability Groups** | Syncs databases across nodes with auto failover | Enterprise HA & read scaling |
| **Log Shipping** | Sends log backups to secondary server for DR | DR with delayed or manual failover |
| **Database Mirroring** | Legacy HA (deprecated in newer versions) | Simple, reliable mirroring (legacy) |
| **Backup/Restore to Cloud** | Manual DR via cloud recovery | Offsite recovery |

**⚙️ Always On Availability Groups**

SQL Server’s flagship HADR solution since 2012.

**Key Features:**

* Multiple readable replicas
* Automatic or manual failover
* Supports encryption, DTC, and listener endpoint

-- Configured through SSMS or PowerShell

-- Requires Windows Failover Clustering and matching editions (Standard or Enterprise)

> 💡 *Use Availability Groups when you need both HA and read scale-out.*

**🔁 Log Shipping**

Moves transaction log backups from primary to standby server on a schedule.

* Easy to configure
* Supports delay for DR scenarios
* Manual failover only

sql

-- Backups copied and restored using Agent jobs

-- Good for geographically distributed DR

**🧠 Choosing the Right Strategy**

| **Requirement** | **Best Option** |
| --- | --- |
| Automatic failover | Always On AG or FCI (Enterprise) |
| Low budget DR | Log Shipping |
| Azure-native HADR | Azure SQL with Geo-Replication |
| Simple HA for legacy systems | Database Mirroring (if supported) |

**🧪 Best Practices**

* Test failover regularly!
* Monitor synchronization health
* Automate alerts for failover events and sync delays
* Validate backups as part of DR strategy
* Document runbooks for manual recovery

# Chapter 27: Implementing a System-Versioned Table in Practice

**🧾 Use Case: Auditing Changes to the Customers Table**

We want to track how customer data (such as credit limits, contact info, or account status) evolves over time—automatically, with no need to handcraft audit triggers.

**🛠️ Step 1: Choose Your Table**

Let’s use Sales.Customers from WideWorldImporters. First, inspect its structure:

SELECT TOP 1 \* FROM Sales.Customers;

**🧱 Step 2: Add Temporal Columns**

You’ll add ValidFrom and ValidTo with system-versioned configuration.

ALTER TABLE Sales.Customers

ADD ValidFrom DATETIME2 GENERATED ALWAYS AS ROW START HIDDEN NOT NULL

CONSTRAINT DF\_Customers\_ValidFrom DEFAULT SYSUTCDATETIME(),

ValidTo DATETIME2 GENERATED ALWAYS AS ROW END HIDDEN NOT NULL

CONSTRAINT DF\_Customers\_ValidTo DEFAULT '9999-12-31 23:59:59.9999999',

PERIOD FOR SYSTEM\_TIME (ValidFrom, ValidTo);

> 🧠 *The* HIDDEN *keyword keeps those columns out of SELECT \* queries—optional but tidy.*

**🔄 Step 3: Enable System Versioning**

You can let SQL Server create the history table automatically or specify your own:

ALTER TABLE Sales.Customers

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

**🧪 Step 4: Test It**

Make a change:

UPDATE Sales.Customers

SET PhoneNumber = '888-555-0199'

WHERE CustomerID = 1;

Now query the history:

SELECT \*

FROM Sales.Customers

FOR SYSTEM\_TIME ALL

WHERE CustomerID = 1;

You’ll see previous versions, including what the record looked like before the update—with automatic timestamps!

**🔍 Optional: Restore Past State**

SELECT \*

FROM Sales.Customers

FOR SYSTEM\_TIME AS OF '2025-05-01T12:00:00'

WHERE CustomerID = 1;

**🧠**

**Tips**

* Add indexes to the history table for high-volume systems.
* Use FOR SYSTEM\_TIME BETWEEN to detect trends or investigate user issues.
* Clean up old history with filtered deletes on the history table, if allowed.

**Scenario: Auditing and Flagging High-Risk Customers**

**Business Need:** The finance team wants to identify customers whose credit limits have decreased over time and notify account managers if the drop exceeds 25%. This helps proactively manage financial risk and customer relationships.

**🧱 Step 1: Enable System-Versioning on Sales.Customers**

ALTER TABLE Sales.Customers

ADD ValidFrom DATETIME2 GENERATED ALWAYS AS ROW START NOT NULL

CONSTRAINT DF\_Customers\_ValidFrom DEFAULT SYSUTCDATETIME(),

ValidTo DATETIME2 GENERATED ALWAYS AS ROW END NOT NULL

CONSTRAINT DF\_Customers\_ValidTo DEFAULT '9999-12-31 23:59:59.9999999',

PERIOD FOR SYSTEM\_TIME (ValidFrom, ValidTo);

ALTER TABLE Sales.Customers

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

**🔍 Step 2: Query Credit Limit Changes Over Time**

WITH CreditChanges AS (

SELECT

CustomerID,

CustomerName,

ValidFrom,

ValidTo,

CreditLimit,

LAG(CreditLimit) OVER (PARTITION BY CustomerID ORDER BY ValidFrom) AS PrevCreditLimit

FROM Sales.Customers

FOR SYSTEM\_TIME ALL

)

SELECT \*

FROM CreditChanges

WHERE PrevCreditLimit IS NOT NULL

AND CreditLimit < PrevCreditLimit \* 0.75;

> This identifies customers whose credit limit dropped by more than 25%.

**🧠 Step 3: Create a Stored Procedure to Flag and Notify**

CREATE PROCEDURE AuditHighRiskCustomers

AS

BEGIN

SET NOCOUNT ON;

SELECT

c.CustomerID,

c.CustomerName,

c.PhoneNumber,

c.PrimaryContactPersonID,

c.CreditLimit,

ch.PrevCreditLimit,

c.ValidFrom

INTO #HighRiskCustomers

FROM (

SELECT

CustomerID,

CustomerName,

CreditLimit,

ValidFrom,

LAG(CreditLimit) OVER (PARTITION BY CustomerID ORDER BY ValidFrom) AS PrevCreditLimit

FROM Sales.Customers

FOR SYSTEM\_TIME ALL

) ch

JOIN Sales.Customers c ON ch.CustomerID = c.CustomerID

WHERE ch.PrevCreditLimit IS NOT NULL

AND ch.CreditLimit < ch.PrevCreditLimit \* 0.75;

-- Simulate notification (e.g., insert into audit log or send email)

INSERT INTO Application.PeopleAuditLog (PersonID, Action, ActionDate)

SELECT PrimaryContactPersonID, 'Credit limit drop >25%', SYSUTCDATETIME()

FROM #HighRiskCustomers;

DROP TABLE #HighRiskCustomers;

END;

**⚡ Step 4: Schedule with SQL Server Agent**

Create a job that runs EXEC AuditHighRiskCustomers weekly and sends an alert if any rows are inserted into the audit log.

**🧪 Optional Enhancements**

* Add a **filtered index** on Sales.Customers(CreditLimit) for faster scans
* Use **Query Store** to monitor performance over time
* Extend the procedure to **email account managers** using Database Mail

This example blends **temporal querying**, **window functions**, **stored procedures**, **auditing**, and **automation**—all grounded in the WWI schema.