1st Edition

MASTERING BUSINESS DATA WITH SQL

A Practical Guide to Querying, Modeling, and Compliance Using SQL Server 2025

AGUS KURNIAWAN

Mastering Business Data with SQL

A Practical Guide to Querying, Modeling, and Compliance Using SQL Server 2025

Agus Kurniawan

Ilmu Data

5 August 2025

Table of Contents

Preface

In today’s data-driven business landscape, mastering SQL is essential for unlocking the full potential of your organization’s information. SQL Server 2025 introduces powerful new features and enhancements, enabling professionals to query, model, and manage data with greater efficiency and compliance.

“Mastering Business Data with SQL: A Practical Guide to Querying, Modeling, and Compliance Using SQL Server 2025” is crafted to help you build a strong foundation in SQL while exploring advanced techniques for real-world business scenarios. This book offers a hands-on approach, guiding you through practical examples and step-by-step instructions to ensure you gain the skills needed to work confidently with business data.

Inside, you will discover:

- Core SQL concepts and the latest features in SQL Server 2025.
- Methods for querying and analyzing business data effectively.
- Best practices for designing and modeling databases for scalability and compliance.
- Advanced topics such as data security, regulatory compliance, and performance optimization.
- Practical case studies demonstrating SQL’s role in solving business challenges.
- Tips for deploying, maintaining, and optimizing SQL Server environments.

Whether you are a beginner or an experienced professional aiming to deepen your expertise, this book serves as a comprehensive resource for students, IT specialists, and anyone seeking to harness the power of SQL Server 2025 in business data management.

Agus Kurniawan

Depok, August 2025

Acknowledgments

I extend my sincere gratitude to the SQL Server community, dedicated contributors, and data professionals whose insights and encouragement have greatly influenced the development of this book.

As you read through these chapters, I hope *Mastering Business Data with SQL: A Practical Guide to Querying, Modeling, and Compliance Using SQL Server 2025* proves both practical and inspiring, empowering you to achieve new levels of proficiency and innovation in business data management.

Section 1: Getting Started with SQL Server 2025

1 Introduction and Setup

1.1 Introduction

This book is designed for **business users, data analysts, and professionals** who work with relational data and need to develop skills in **querying, modeling, and ensuring compliance** using **SQL Server 2025**. You may be:

- A **business analyst** generating reports and insights from enterprise data
- A **data professional** supporting analytics and reporting pipelines
- A **developer** building SaaS solutions with secure, multi-tenant databases
- A **compliance-oriented user** interested in applying data governance, security, and auditing best practices

Unlike books that target full-time database administrators (DBAs) or performance tuners, this book emphasizes **real-world, hands-on skills** for interacting with SQL Server in a **business-driven context**, with a focus on:

- Writing effective and meaningful queries
- Designing relational data models that scale
- Enforcing access control, masking, and auditing for compliance (e.g., GDPR)
- Supporting **multi-tenant** SaaS-style architecture from a data perspective

The book assumes **basic familiarity with databases**, such as tables and rows, but **no prior experience with T-SQL or SQL Server** is required.

1.2 What's New in SQL Server 2025

SQL Server 2025, part of Microsoft's modern data platform, continues its evolution as a hybrid and cloud-integrated database engine. As of July 2025, these are the **notable features** relevant to this book's focus:

- ✅ **Enhanced T-SQL capabilities**, including window function improvements, performance hints, and support for `DATETIME2` granularity enhancements
- ✅ **Built-in support for data classification**, sensitivity labeling, and improved auditing — critical for GDPR/PII compliance
- ✅ **Improved Row-Level Security (RLS)** performance and diagnostics
- ✅ **Support for ledger tables** (blockchain-style tamper-evidence) for scenarios where data integrity tracking is required

- ✅ **SSMS 20.x** and **Azure Data Studio 1.47+** fully support SQL Server 2025's feature set
- ✅ Seamless integration with **Microsoft Purview**, **Power BI**, and **Azure Arc** for hybrid deployments

These features make SQL Server 2025 well-suited for **data-driven businesses**, especially those delivering **multi-tenant services**, operating under **regulatory compliance**, or requiring **enterprise-level data analysis**.

1.3 Tools You'll Use in This Book

To follow along with the exercises and labs in this book, you'll primarily use the following tools:

1.3.1 SQL Server Management Studio (SSMS) 21.x

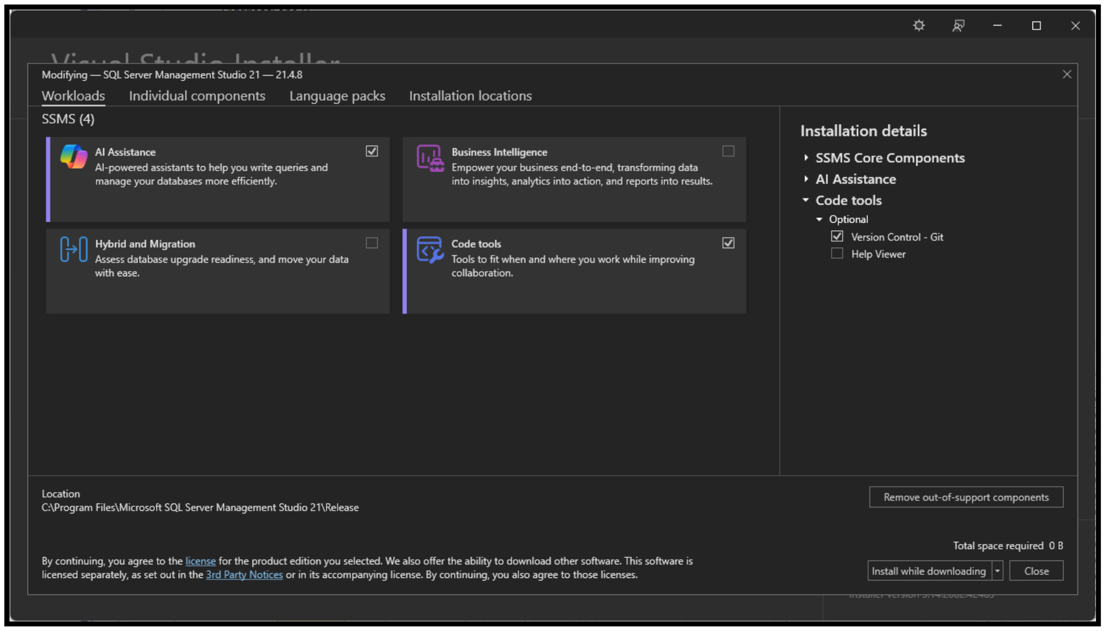
The classic, full-featured Windows tool for managing SQL Server databases. SSMS includes:

- Query editor with IntelliSense for T-SQL
- Object Explorer for browsing schemas, views, procedures
- UI-based features for backups, security, and auditing
- Graphical plans for query analysis

> 🔎 *As of July 2025, the latest SSMS release is **v21.x**, which is compatible with SQL Server 2025 and backward-compatible with earlier versions.*

You can download SSMS from the official Microsoft site: Download SQL Server Management Studio https://learn.microsoft.com/en-us/ssms/install/install.

While installing, you can choose some workload-specific options, but the defaults are generally sufficient for most users. Figure 1.1 shows the installation process.



Modifying — SQL Server Management Studio 21 — 21.4.8

SSMS (4)

- [✓] **AI Assistance** — AI-powered assistants to help you write queries and manage your databases more efficiently.
- [ ] **Business Intelligence** — Empower your business end-to-end, transforming data into insights, insights into action, and reports into results.
- [ ] **Hybrid and Migration** — Assess database upgrade readiness, and move your data with ease.
- [✓] **Code tools** — Tools to fit when and where you work while improving collaboration.

Installation details — Code tools — Optional: [✓] Version Control - Git; [ ] Help Viewer

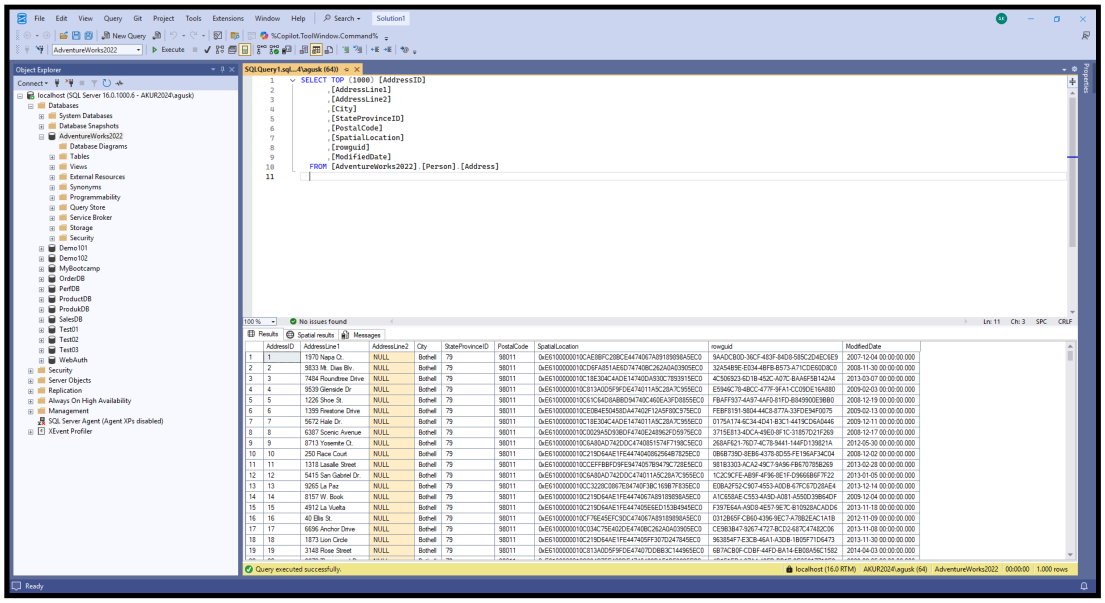
Location
C:\Program Files\Microsoft SQL Server Management Studio 21\Release

Total space required 0 B

By continuing, you agree to the license for the product edition you selected. We also offer the ability to download other software. This software is licensed separately, as set out in the 3rd Party Notices or in its accompanying license. By continuing, you also agree to those licenses.

Figure 1.1: Installing SQL Server Management Studio (SSMS) 21.x.

After installation, you can launch SSMS and connect to your SQL Server instance using Windows Authentication or SQL Server Authentication. The connection dialog allows you to specify the server name, authentication method, and credentials.



```sql
SELECT TOP (1000) [AddressID]
      ,[AddressLine1]
      ,[AddressLine2]
      ,[City]
      ,[StateProvinceID]
      ,[PostalCode]
      ,[SpatialLocation]
      ,[rowguid]
      ,[ModifiedDate]
  FROM [AdventureWorks2022].[Person].[Address]
```

	AddressID	AddressLine1	AddressLine2	City	StateProvinceID	PostalCode	SpatialLocation	rowguid	ModifiedDate
1	1	1970 Napa Ct.	NULL	Bothell	79	98011	0xE6100000010CAE8BFC28BCE4474067A89189898A5EC0	9AADCB0D-36CF-483F-84D8-585C2D4EC6E9	2007-12-04 00:00:00.000
2	2	9833 Mt. Dias Blv.	NULL	Bothell	79	98011	0xE6100000010CD6FA851AE6D74740BC262A0A03905EC0	32A54B9E-E034-4BFB-B573-A71CDE60D8C0	2008-11-30 00:00:00.000
3	3	7484 Roundtree Drive	NULL	Bothell	79	98011	0xE6100000010C18E304C4ADE14740DA930C7893915EC0	4C506923-6D1B-452C-A07C-BAA6F5B142A4	2013-03-07 00:00:00.000
4	4	9539 Glenside Dr	NULL	Bothell	79	98011	0xE6100000010C813A0D5F9FDE474011A5C28A7C955EC0	E5946C78-4BCC-477F-9FA1-CC09DE16A880	2009-02-03 00:00:00.000
5	5	1226 Shoe St.	NULL	Bothell	79	98011	0xE6100000010C61C64D8ABBD94740C460EA3FD8855EC0	FBAFF937-4A97-4AF0-81FD-B849900E9BB0	2008-12-19 00:00:00.000
6	6	1399 Firestone Drive	NULL	Bothell	79	98011	0xE6100000010CE0B4E50458DA47402F12A5F80C975EC0	FEBF8191-9804-44C8-877A-33FDE94F0075	2009-02-13 00:00:00.000
7	7	5672 Hale Dr.	NULL	Bothell	79	98011	0xE6100000010C18E304C4ADE1474011A5C28A7C955EC0	0175A174-6C34-4D41-B3C1-4419CD6A0446	2009-12-11 00:00:00.000
8	8	6387 Scenic Avenue	NULL	Bothell	79	98011	0xE6100000010C0029A5D93BDF4740E248962FD5975EC0	3715E813-4DCA-49E0-8F1C-31857D21F269	2008-12-17 00:00:00.000
9	9	8713 Yosemite Ct.	NULL	Bothell	79	98011	0xE6100000010C6A80AD742DDC4740851574F7198C5EC0	268AF621-76D7-4C78-9441-144FD139821A	2012-05-30 00:00:00.000
10	10	250 Race Court	NULL	Bothell	79	98011	0xE6100000010C219D64AE1FE4474040862564B7825EC0	0B6B739D-8EB6-4378-8D55-FE196AF34C04	2008-12-02 00:00:00.000
11	11	1318 Lasalle Street	NULL	Bothell	79	98011	0xE6100000010CCEFFBBFD9FE9474057B9479C728E5EC0	981B3303-ACA2-49C7-9A96-FB670785B269	2013-02-28 00:00:00.000
12	12	5415 San Gabriel Dr.	NULL	Bothell	79	98011	0xE6100000010C6A80AD742DDC474011A5C28A7C955EC0	1C2C9CFE-AB9F-4F96-8E1F-D9666B6F7F22	2013-01-05 00:00:00.000
13	13	9265 La Paz	NULL	Bothell	79	98011	0xE6100000010CC3228C0867E84740F3BC169B7F835EC0	E0BA2F52-C907-4553-A0DB-67FC67D28AE4	2013-12-14 00:00:00.000
14	14	8157 W. Book	NULL	Bothell	79	98011	0xE6100000010C219D64AE1FE4474067A89189898A5EC0	A1C658AE-C553-4A9D-A081-A550D39B64DF	2009-12-04 00:00:00.000
15	15	4912 La Vuelta	NULL	Bothell	79	98011	0xE6100000010C219D64AE1FE447405E6ED153B4945EC0	F397E64A-A9D8-4E57-9E7C-B10928ACADD6	2013-11-18 00:00:00.000
16	16	40 Ellis St.	NULL	Bothell	79	98011	0xE6100000010CF76E45EFC9DC474067A89189898A5EC0	0312B65F-CB60-4396-9EC7-A78B2EAC1A1B	2012-11-09 00:00:00.000
17	17	6696 Anchor Drive	NULL	Bothell	79	98011	0xE6100000010C034C75E402DE4740BC262A0A03905EC0	CE9B3B47-9267-4727-BCD2-687C47482C06	2013-11-08 00:00:00.000
18	18	1873 Lion Circle	NULL	Bothell	79	98011	0xE6100000010C219D64AE1FE447405FF307D247845EC0	963854F7-E3CB-46A1-A3DB-1B05F71D6473	2013-11-30 00:00:00.000
19	19	3148 Rose Street	NULL	Bothell	79	98011	0xE6100000010C813A0D5F9FDE47407DDBB3C144965EC0	6B7ACB0F-CDBF-44FD-BA14-EB08A56C1582	2014-04-03 00:00:00.000

Query executed successfully. localhost (16.0 RTM) AKUR2024\agusk (64) AdventureWorks2022 00:00:00 1.000 rows

Figure 1.2: SQL Server Management Studio (SSMS) 21.x.

1.3.2 Visual Studio Code with MSSQL Extension

As Microsoft transitions away from Azure Data Studio, **Visual Studio Code (VS Code)** combined with the official **MSSQL extension** is now the **preferred**

lightweight, cross-platform SQL editor. With this setup, you can:

- Connect to SQL Server on Windows, Linux, or macOS
- Run and save T-SQL queries directly from the VS Code editor
- Use IntelliSense, result grid, and connection profiles
- Leverage Git and terminal integration for hybrid workflows

> 🟢 *The **MSSQL extension**, maintained by Microsoft, brings ADS-like features to VS Code and is actively supported as of July 2025.*

> ❗ *Microsoft has officially announced the retirement of Azure Data Studio, and recommends moving to **VS Code + MSSQL extension** for future development: What's Happening to Azure Data Studio*

1.3.3 Optional Tools

While the primary focus is on SSMS and VS Code, you may also find value in exploring additional tools that complement your SQL Server workflow. These optional tools can help with data visualization, automation, and advanced management tasks, depending on your specific needs and environment.

- **Power BI Desktop**: Connects directly to SQL Server to build visual dashboards
- **SQLCMD** or **Azure CLI**: For automation and command-line interaction (covered briefly)

1.4 Exercise 1: Install SQL Server and Restore AdventureWorks2022

1.4.1 Description

In this lab, you will install **SQL Server 2025 Developer Edition**, set up **SQL Server Management Studio (SSMS)**, and restore the **AdventureWorks2022** sample database. This environment will be used for hands-on exercises throughout the book.

1.4.2 Objectives

- Install **SQL Server 2025 Developer Edition**
- Install **SSMS 21.x**

- Download the **AdventureWorks2022.bak** sample database
- Restore the `.bak` file into a new database using SSMS

1.4.3 Prerequisites

- Windows 10/11 or Windows Server 2019/2022
- Administrator access to install software
- Stable internet connection
- At least 4 GB RAM and 10 GB free disk space

1.4.4 Steps

Here's a step-by-step guide to complete this exercise:

1.4.4.1 Step 1: Download and Install SQL Server 2025 Developer Edition

1. Visit the SQL Server 2025 download pagehttps://www.microsoft.com/en-us/sql-server/sql-server-downloads

   Since SQL Server 2025 is not yet released, you can download the **SQL Server 2025** on https://www.microsoft.com/en-us/evalcenter/evaluate-sql-server-2025. The steps will be similar.

2. Download the **Developer Edition** installer (free for development use)

3. Run the installer and choose **Basic installation**

4. Accept license terms and continue

5. Wait until the installation completes

6. Note the **instance name** (default: `MSSQLSERVER`) and make sure SQL Server services are running

1.4.4.2 Step 2: Install SQL Server Management Studio (SSMS) 21.x

1. Go to the official SSMS download page, https://learn.microsoft.com/en-us/ssms/install/install
2. Download the **latest SSMS 21.x** installer (as of July 2025)
3. Run the installer and complete the setup
4. Launch **SSMS** and connect to your local SQL Server instance

1.4.4.3 Step 3: Download AdventureWorks2022 Sample Database

1. Visit the official Microsoft GitHub repository:

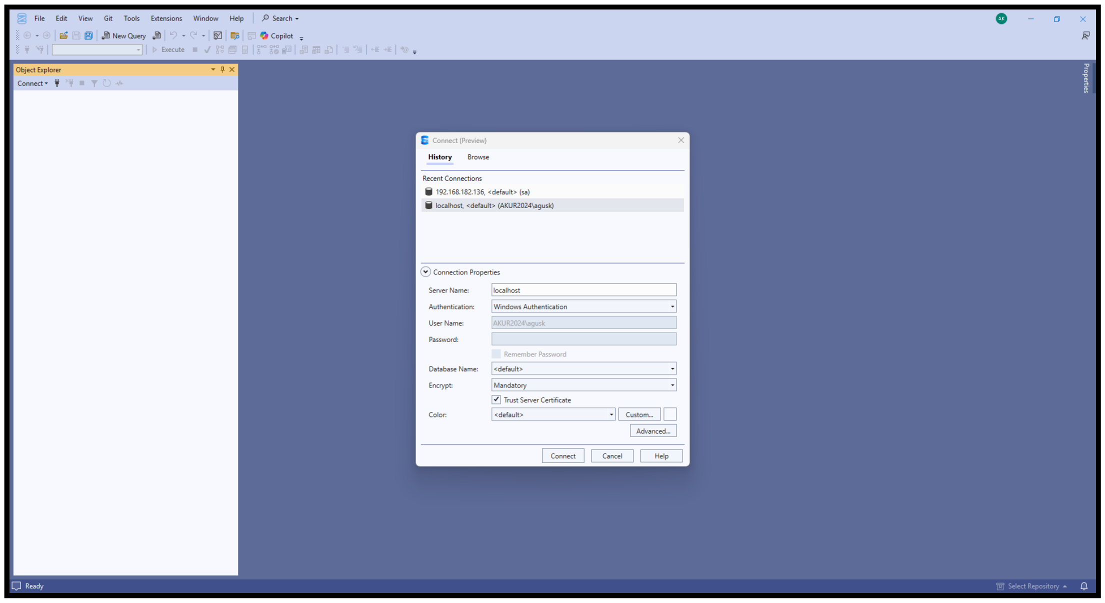
   👉 https://github.com/microsoft/sql-server-samples/releases

2. Locate and download: `AdventureWorks2022.bak` from the **AdventureWorks OLTP** section.

   Direct link (as of July 2025):

   👉 https://github.com/microsoft/sql-server-samples/releases/tag/adventureworks

3. Save the `.bak` file to a known directory, e.g., `C:\Backups\AdventureWorks2022.bak`



Recent Connections
192.168.182.136, <default> (sa)
localhost, <default> (AKUR2024\agusk)

Connection Properties
Server Name: localhost
Authentication: Windows Authentication
User Name: AKUR2024\agusk
Password:
Remember Password
Database Name: <default>
Encrypt: Mandatory
✓ Trust Server Certificate
Color: <default>

Figure 1.3: Connect to SQL Server from SQL Server Management Studio (SSMS) 21.x.

1.4.4.4 Step 4: Restore AdventureWorks2022 in SSMS

1. Open **SSMS** and connect to the SQL Server instance

2. In Object Explorer, right-click on **Databases** → choose **Restore Database…**

3. In the *Source* section:

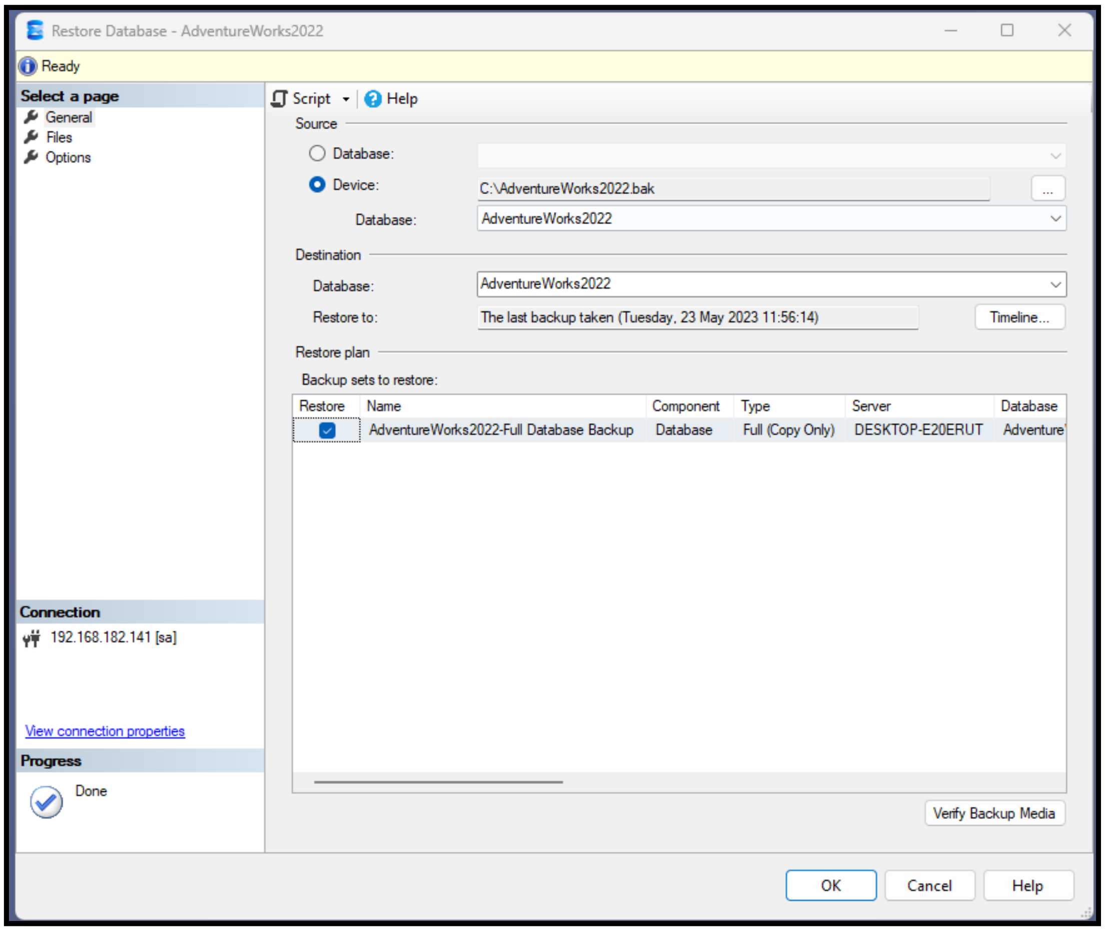
   - Select **Device**
   - Click **Add…** and browse to `C:\Backups\AdventureWorks2022.bak`

4. In the *Destination* section:

   - Database name: `AdventureWorks2022`

5. Click on the **Files** tab:

   - Change the restore file path if needed (ensure it points to a valid `DATA` folder)



Restore Database - AdventureWorks2022

Ready

Source
- ○ Database:
- ● Device: C:\AdventureWorks2022.bak
- Database: AdventureWorks2022

Destination
- Database: AdventureWorks2022
- Restore to: The last backup taken (Tuesday, 23 May 2023 11:56:14)

Restore plan

Backup sets to restore:

Restore	Name	Component	Type	Server	Database
✓	AdventureWorks2022-Full Database Backup	Database	Full (Copy Only)	DESKTOP-E20ERUT	Adventure

Connection: 192.168.182.141 [sa]

Progress: Done

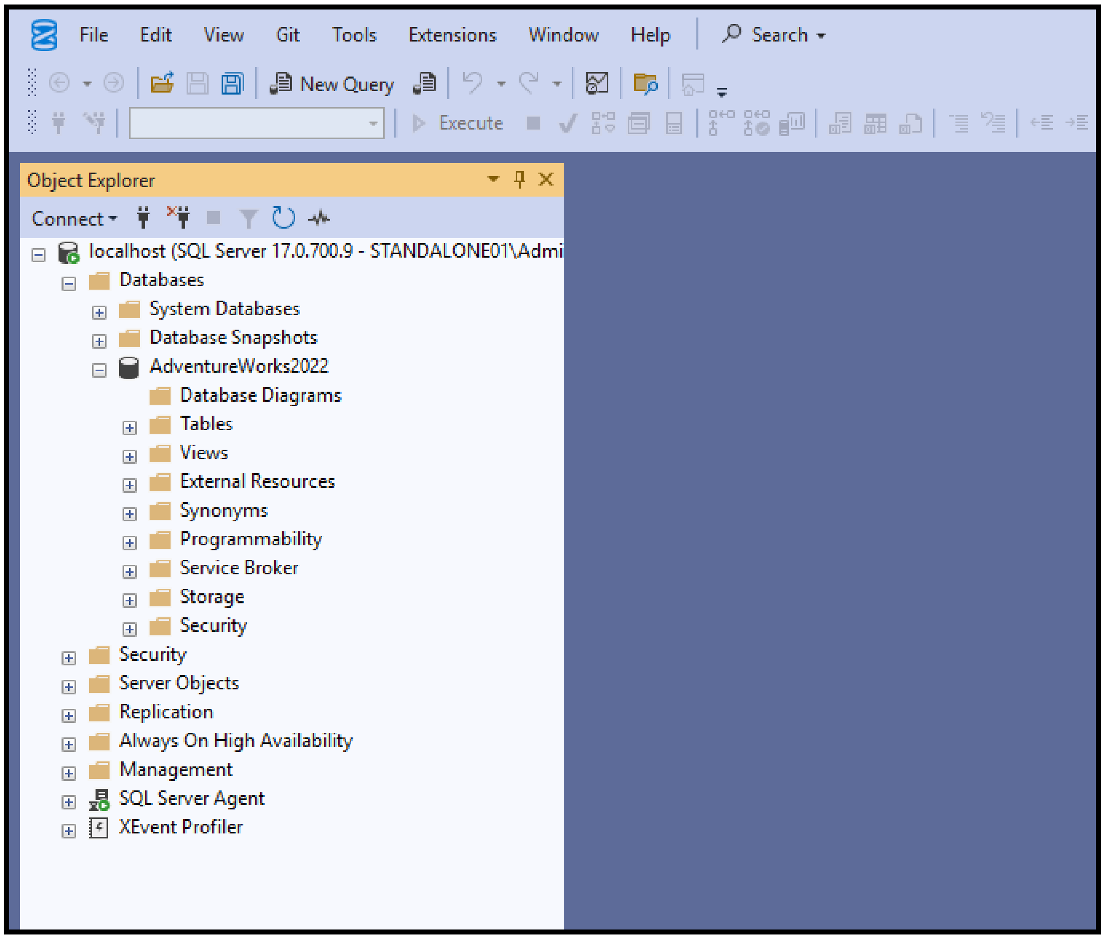
Figure 1.4: Restore AdventureWorks2022.

6. Click **OK** to begin the restore

7. Wait until the success message appears

1.4.4.5 Step 5: Verify the Database

1. In Object Explorer, expand **Databases** → **AdventureWorks2022**



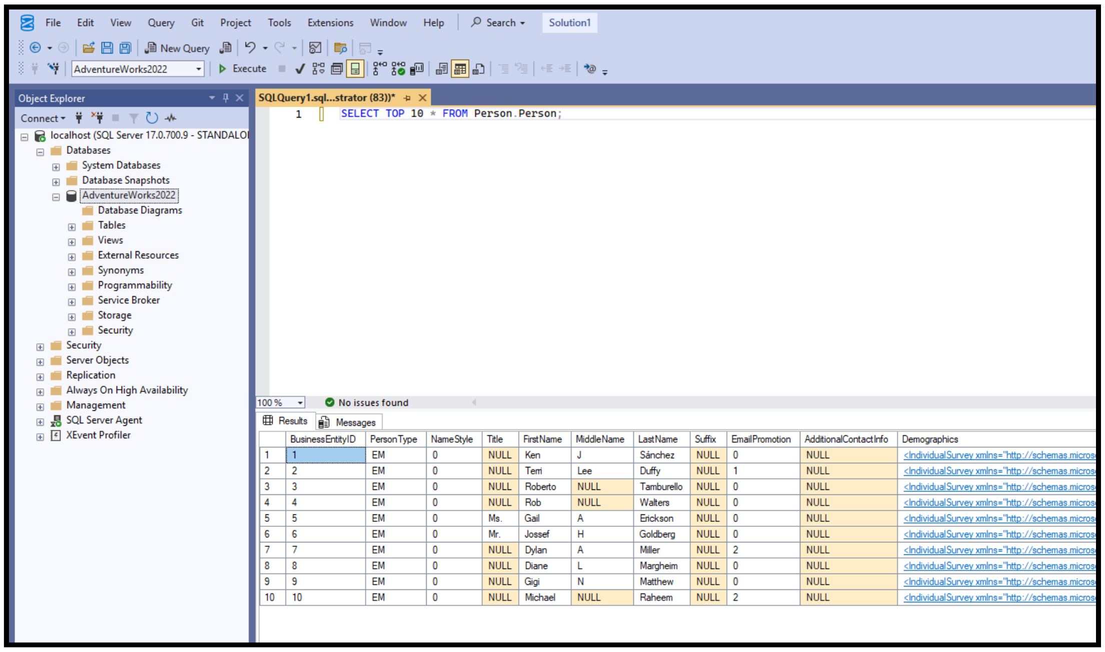
localhost (SQL Server 17.0.700.9 - STANDALONE01\Admi
Databases
System Databases
Database Snapshots
AdventureWorks2022
Database Diagrams
Tables
Views
External Resources
Synonyms
Programmability
Service Broker
Storage
Security
Security
Server Objects
Replication
Always On High Availability
Management
SQL Server Agent
XEvent Profiler

Figure 1.5: Explore AdventureWorks2022 database.

2. Expand **Tables** to verify that objects like `Sales.SalesOrderHeader` and `Person.Person` exist

3. Run a sample query:

```
SELECT TOP 10 * FROM Person.Person;
```



```sql
SELECT TOP 10 * FROM Person.Person;
```

	BusinessEntityID	PersonType	NameStyle	Title	FirstName	MiddleName	LastName	Suffix	EmailPromotion	AdditionalContactInfo	Demographics
1	1	EM	0	NULL	Ken	J	Sánchez	NULL	0	NULL	<IndividualSurvey xmlns="http://schemas.micros
2	2	EM	0	NULL	Terri	Lee	Duffy	NULL	1	NULL	<IndividualSurvey xmlns="http://schemas.micros
3	3	EM	0	NULL	Roberto	NULL	Tamburello	NULL	0	NULL	<IndividualSurvey xmlns="http://schemas.micros
4	4	EM	0	NULL	Rob	NULL	Walters	NULL	0	NULL	<IndividualSurvey xmlns="http://schemas.micros
5	5	EM	0	Ms.	Gail	A	Erickson	NULL	0	NULL	<IndividualSurvey xmlns="http://schemas.micros
6	6	EM	0	Mr.	Jossef	H	Goldberg	NULL	0	NULL	<IndividualSurvey xmlns="http://schemas.micros
7	7	EM	0	NULL	Dylan	A	Miller	NULL	2	NULL	<IndividualSurvey xmlns="http://schemas.micros
8	8	EM	0	NULL	Diane	L	Margheim	NULL	0	NULL	<IndividualSurvey xmlns="http://schemas.micros
9	9	EM	0	NULL	Gigi	N	Matthew	NULL	0	NULL	<IndividualSurvey xmlns="http://schemas.micros
10	10	EM	0	NULL	Michael	NULL	Raheem	NULL	2	NULL	<IndividualSurvey xmlns="http://schemas.micros

Figure 1.6: Perform a query on MSSQL Studio.

1.4.5 Summary

In this exercise, you:

- Installed **SQL Server 2025 Developer Edition** and **SSMS 21.x**
- Downloaded the **AdventureWorks2022** `.bak` file from Microsoft's GitHub repository
- Restored the database into SQL Server using **SSMS**
- Verified the successful restore and ran a test query

You are now ready to begin querying and working with a real-world sample database in the next chapters.

1.5 Exercise 2: Explore SSMS and Run Your First Query

1.5.1 Description

In this hands-on lab, you will explore the **SQL Server Management Studio (SSMS) 21.x** user interface and run your first query against the **AdventureWorks2022** database. This lab helps you become familiar with core features of SSMS that you'll use throughout the book.

1.5.2 Objectives

- Navigate the SSMS interface: Object Explorer, Query Editor, and Results Pane
- Connect to a SQL Server 2025 instance
- Run a basic `SELECT` statement on the restored AdventureWorks2022 database
- View and interpret query results

1.5.3 Prerequisites

- SQL Server 2025 Developer Edition is installed and running
- SQL Server Management Studio (SSMS) 21.x is installed
- AdventureWorks2022 database is already restored (from **Exercise 1**)
- User has access to connect as a SQL Server administrator or equivalent

1.5.4 Steps

Here's a step-by-step guide to complete this exercise:

1.5.4.1 Step 1: Launch SSMS and Connect to SQL Server

1. Open **SQL Server Management Studio (SSMS)**.

2. In the **Connect to Server** dialog:

   - Server type: *Database Engine*
   - Server name: *(local)* or `localhost`
   - Authentication: *Windows Authentication* (or SQL Authentication if configured)

3. Click **Connect**.

4. Once connected, you will see **Object Explorer** on the left panel.

1.5.4.2 Step 2: Explore the SSMS Interface

Familiarize yourself with key UI elements:

- **Object Explorer** (left pane): View servers, databases, tables, views, and more
- **Query Editor**: Write and run T-SQL scripts
- **Toolbar**: Save files, execute queries, and format code
- **Results Pane**: View query output (grid or text)

Expand the following to get familiar:

1. Databases → **AdventureWorks2022**

2. Expand **Tables**, **Views**, and **Security**

1.5.4.3 Step 3: Open a New Query Window

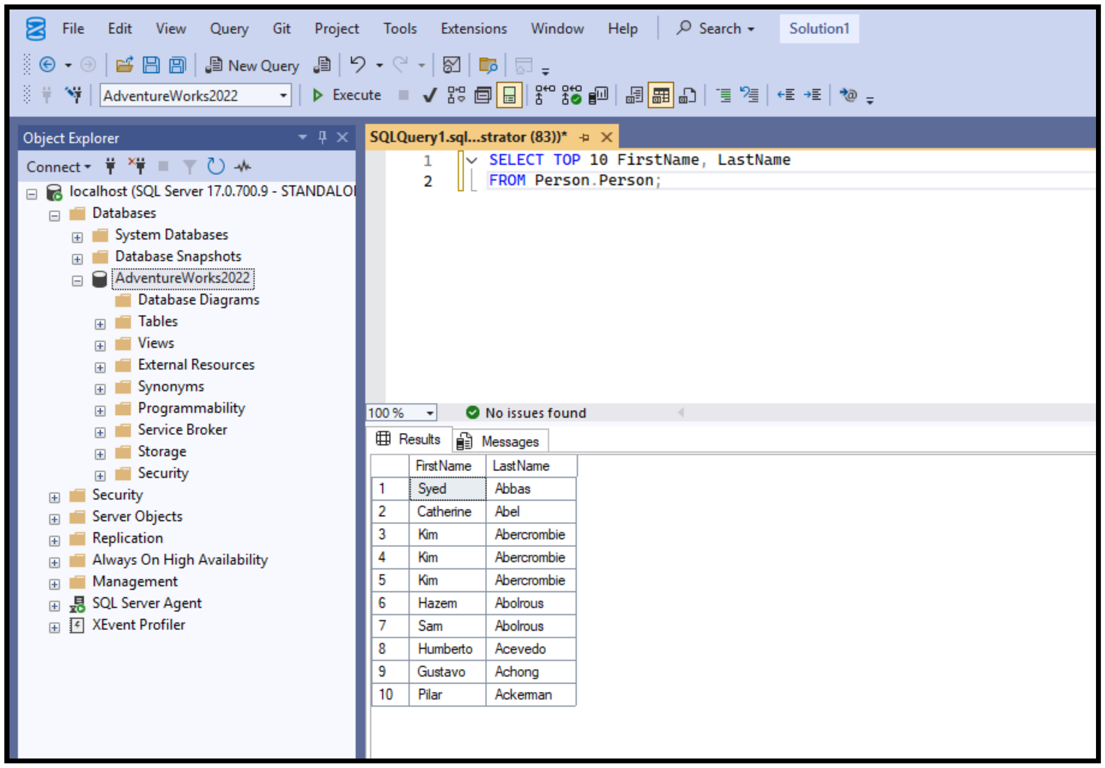
1. In Object Explorer, right-click on the `AdventureWorks2022` database
2. Click **New Query**
3. Ensure the database context (top-left dropdown) is set to `AdventureWorks2022`

1.5.4.4 Step 4: Run a Simple Query

In the new query window, type the following T-SQL statement:

```sql
SELECT TOP 10 FirstName, LastName
FROM Person.Person;
```

Click **Execute** (or press `F5`).



```sql
SELECT TOP 10 FirstName, LastName
FROM Person.Person;
```

	FirstName	LastName
1	Syed	Abbas
2	Catherine	Abel
3	Kim	Abercrombie
4	Kim	Abercrombie
5	Kim	Abercrombie
6	Hazem	Abolrous
7	Sam	Abolrous
8	Humberto	Acevedo
9	Gustavo	Achong
10	Pilar	Ackerman

Figure 1.7: Run a query.

1.5.4.5 Step 5: View the Results

- Review the **Results** tab (default grid view)
- Notice column names and values

- Optionally click the **Messages** tab to view query execution info (e.g., `(10 rows affected)`)

You can also change the output to:

- Text: Right-click in the query window → Results To → Results to Text (`Ctrl+T`)
- File: Right-click → Results To → Results to File (`Ctrl+Shift+F`)

1.5.5 Summary

In this exercise, you:

- Connected to SQL Server 2025 using **SSMS 21.x**
- Navigated the SSMS interface and Object Explorer
- Opened a query window and set the database context
- Ran your first SQL query against the **AdventureWorks2022** database
- Viewed results in the results pane

You're now ready to dive into querying and data exploration in upcoming chapters.

1.6 Conclusion

This book provides a comprehensive introduction to SQL Server 2025, focusing on practical skills for business users and data professionals. By completing the exercises in this chapter, you have set up your environment and gained familiarity with the tools and interfaces that will be used throughout the book.

Section 2: Querying Data – Core Skills

2 SELECT and Filtering Essentials

In this chapter, we introduce the foundation of querying data from SQL Server 2025. You'll learn how to retrieve specific columns using `SELECT`, limit rows with `WHERE`, and organize results with `ORDER BY`. These skills are essential for analyzing business data effectively and precisely.

2.1 The SELECT Statement: Retrieving Data

The `SELECT` statement is the most fundamental command in SQL—it lets you extract data from one or more tables.

Sample syntax:

```sql
SELECT column1, column2, ...
FROM table_name;
```

Here's how to retrieve all columns from the `Customers` table in the `Sales` schema:

```sql
SELECT *
FROM Sales.Customers;
```

Here's how to select only the `CustomerID`, `FirstName`, and `LastName` columns:

```sql
SELECT CustomerID, FirstName, LastName
FROM Sales.Customers;
```

Avoid `SELECT *` in production queries. Always specify the columns you need to improve performance and clarity.

2.2 The WHERE Clause: Filtering Rows

`WHERE` is used to filter rows that meet specific criteria. It allows business users to focus on relevant data only.

Sample syntax:

```sql
SELECT column1, column2, ...
FROM table_name
```

```
WHERE condition;
```

Comparison operators are used in the `WHERE` clause to filter data based on specific conditions. Here are the common operators:

Operator	**Meaning**
=	Equal
<> or !=	Not equal
>	Greater than
<	Less than
>=	Greater or equal
<=	Less or equal

Here’s how to select customers from the ‘North’ region:

```
SELECT CustomerID, Region
FROM Sales.Customers
WHERE Region = 'North';
```

You can combine multiple conditions using `AND` and `OR`:

```
SELECT OrderID, OrderDate, TotalAmount
FROM Sales.Orders
WHERE TotalAmount > 1000 AND Status = 'Completed';
```

The `WHERE` clause can also use special operators for more complex filtering:

- **IN**: Checks if a value exists in a list
- **BETWEEN**: Checks within a range
- **LIKE**: Pattern matching (with `%` and `_`)

Here are some examples:

```
-- Customers from specific regions
SELECT FirstName, LastName
FROM Sales.Customers
WHERE Region IN ('West', 'South');

-- Orders between two dates
SELECT OrderID, OrderDate
FROM Sales.Orders
WHERE OrderDate BETWEEN '2025-01-01' AND '2025-06-30';
```

```sql
-- Customers with names starting with 'J'
SELECT FirstName, LastName
FROM Sales.Customers
WHERE FirstName LIKE 'J%';
```

Explanation:

- The first query retrieves customers from the ‘West’ and ‘South’ regions.
- The second query finds orders placed in the first half of 2025.
- The third query gets customers whose first names start with ‘J’.

2.3 The ORDER BY Clause: Sorting Results

`ORDER BY` is used to sort query results by one or more columns, either ascending (`ASC`, default) or descending (`DESC`).

Basic syntax:

```sql
SELECT column1, column2, ...
FROM table_name
ORDER BY column1 [ASC|DESC], column2 [ASC|DESC];
```

Here’s how to sort customers by their last names in ascending order:

```sql
SELECT FirstName, LastName
FROM Sales.Customers
ORDER BY LastName ASC;
```

Here’s how to sort customers by their total purchase amount in descending order:

```sql
SELECT CustomerID, TotalPurchase
FROM Sales.CustomerRevenue
ORDER BY TotalPurchase DESC;
```

2.4 Combining SELECT, WHERE, and ORDER BY

You can combine `SELECT`, `WHERE`, and `ORDER BY` to create powerful queries that retrieve and organize data effectively.

Here’s an example that retrieves orders over $500 and sorts them by order date:

```sql
SELECT OrderID, CustomerID, OrderDate, TotalAmount
FROM Sales.Orders
WHERE TotalAmount > 500
ORDER BY OrderDate DESC;
```

This retrieves orders over $500 and sorts them from the most recent.

2.5 Exercise 3: Select and Filter Data from AdventureWorks2022

2.5.1 Description

In this exercise, you will learn how to retrieve data from a SQL Server 2025 database using the `SELECT` statement and apply filtering using the `WHERE` clause. You'll explore the `Production.Product` and `Sales.SalesOrderHeader` tables in the **AdventureWorks2022** sample database to extract meaningful information.

This lab builds your foundation for writing real-world business queries using **SQL Server Management Studio (SSMS) 21.x**.

2.5.2 Objectives

- Understand how to use the `SELECT` statement to retrieve specific columns
- Apply the `WHERE` clause to filter rows based on conditions
- Use logical operators such as `AND`, `OR`, and comparison operators
- Retrieve business-relevant data from AdventureWorks2022

2.5.3 Prerequisites

- SQL Server 2025 Developer Edition is installed and running
- SSMS 21.x is installed
- **AdventureWorks2022** database has been restored (from **Exercise 1**)
- User is connected to the SQL Server instance with appropriate permissions

2.5.4 Steps

Here's a step-by-step guide to complete this exercise:

2.5.4.1 Step 1: Open a New Query Window in SSMS

1. Launch **SQL Server Management Studio (SSMS)**
2. Connect to your SQL Server instance
3. In **Object Explorer**, expand **Databases** → right-click on **AdventureWorks2022** → choose **New Query**

4. Ensure the **database context** (in the dropdown near the toolbar) is set to `AdventureWorks2022`

2.5.4.2 Step 2: Retrieve Product Name and List Price

In the query editor, type:

```sql
SELECT Name, ListPrice
FROM Production.Product;
```

Click **Execute** or press `F5`. This returns a list of all products with their list prices.

2.5.4.3 Step 3: Filter Products with Non-Zero Price

Now let’s filter out products that have a price of `0.00`:

```sql
SELECT Name, ListPrice
FROM Production.Product
WHERE ListPrice > 0;
```

This query returns only **sellable products** with a valid list price.

2.5.4.4 Step 4: Add Filtering by Product Color

Let’s get only **red-colored** products:

```sql
SELECT Name, Color, ListPrice
FROM Production.Product
WHERE Color = 'Red';
```

You can change `'Red'` to other values such as `'Black'`, `'Silver'`, etc.

2.5.4.5 Step 5: Combine Multiple Conditions with `AND` and `OR`

Get all **red or black** products with a price above `500`:

```sql
SELECT Name, Color, ListPrice
FROM Production.Product
WHERE (Color = 'Red' OR Color = 'Black')
  AND ListPrice > 500;
```

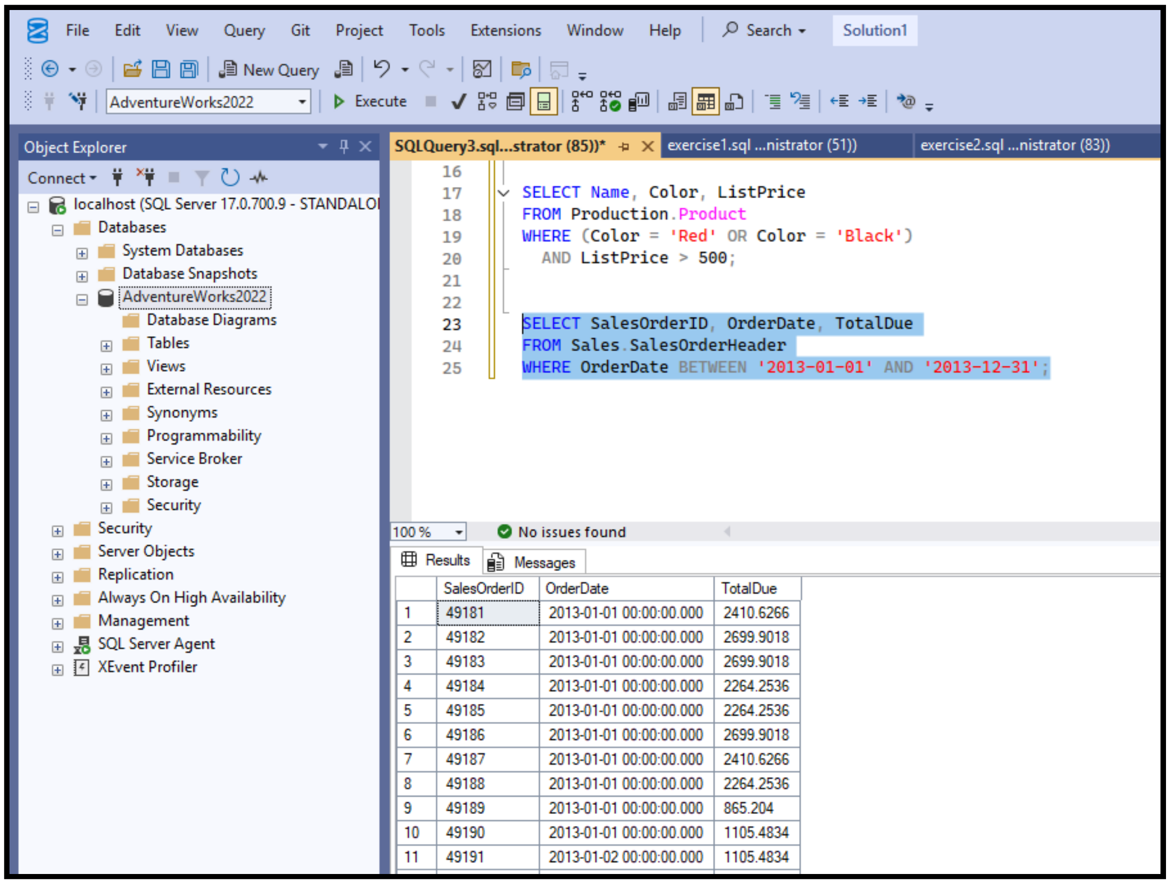
This shows how to use parentheses to control logic with `AND` / `OR`.

2.5.4.6 Step 6: Query Sales Orders by Date Range

Now switch to the **Sales.SalesOrderHeader** table and query orders in 2013:

```
SELECT SalesOrderID, OrderDate, TotalDue
FROM Sales.SalesOrderHeader
WHERE OrderDate BETWEEN '2013-01-01' AND '2013-12-31';
```

This helps you retrieve **filtered transactional data**.



```
SELECT Name, Color, ListPrice
FROM Production.Product
WHERE (Color = 'Red' OR Color = 'Black')
  AND ListPrice > 500;


SELECT SalesOrderID, OrderDate, TotalDue
FROM Sales.SalesOrderHeader
WHERE OrderDate BETWEEN '2013-01-01' AND '2013-12-31';
```

	SalesOrderID	OrderDate	TotalDue
1	49181	2013-01-01 00:00:00.000	2410.6266
2	49182	2013-01-01 00:00:00.000	2699.9018
3	49183	2013-01-01 00:00:00.000	2699.9018
4	49184	2013-01-01 00:00:00.000	2264.2536
5	49185	2013-01-01 00:00:00.000	2264.2536
6	49186	2013-01-01 00:00:00.000	2699.9018
7	49187	2013-01-01 00:00:00.000	2410.6266
8	49188	2013-01-01 00:00:00.000	2264.2536
9	49189	2013-01-01 00:00:00.000	865.204
10	49190	2013-01-01 00:00:00.000	1105.4834
11	49191	2013-01-02 00:00:00.000	1105.4834

Figure 2.1: Perform a query get orders by date range.

2.5.5 Summary

In this exercise, you:

- Used the `SELECT` statement to query specific columns
- Applied `WHERE` filters to extract meaningful subsets of data
- Combined logical conditions with `AND` and `OR`
- Explored data from both **Product** and **SalesOrderHeader** tables in AdventureWorks2022

These skills are essential for any business analyst or developer working with relational data in SQL Server.

2.6 Exercise 4: Filter Sales by Region and Date

2.6.1 Description

In this hands-on lab, you will practice writing SQL queries to filter sales orders based on both **region** (territory) and **date range**, using the **AdventureWorks2022** database in **SQL Server 2025**. You will also use the `ORDER BY` clause to sort the results for better readability and analysis.

2.6.2 Objectives

- Join `Sales.SalesOrderHeader` with `Sales.SalesTerritory`
- Apply multiple filters using `WHERE` with `AND` and `BETWEEN`
- Sort results using the `ORDER BY` clause
- Retrieve region-specific sales activity within a date range

2.6.3 Prerequisites

- SQL Server 2025 Developer Edition installed and running
- SSMS 21.x is installed
- AdventureWorks2022 database is restored (from **Exercise 1**)
- Familiarity with `SELECT`, `WHERE`, and `JOIN` from previous exercises

2.6.4 Steps

Here’s a step-by-step guide to complete this exercise:

2.6.4.1 Step 1: Open a New Query Window in SSMS

1. Launch **SQL Server Management Studio (SSMS)**
2. Connect to your SQL Server instance
3. In **Object Explorer**, right-click **AdventureWorks2022 → New Query**
4. Ensure that the database context is set to `AdventureWorks2022`

2.6.4.2 Step 2: View Sales Order Data

Start with a basic query to explore `Sales.SalesOrderHeader`:

```
SELECT TOP 10 SalesOrderID, OrderDate, TerritoryID, TotalDue
FROM Sales.SalesOrderHeader;
```

> *This gives you a sense of available fields, including region (TerritoryID).*

2.6.4.3 Step 3: Join with Sales Territory Table

To retrieve the **region name**, join with `Sales.SalesTerritory`:

```
SELECT
    h.SalesOrderID,
    h.OrderDate,
    t.Name AS Territory,
    h.TotalDue
FROM Sales.SalesOrderHeader h
JOIN Sales.SalesTerritory t ON h.TerritoryID = t.TerritoryID;
```

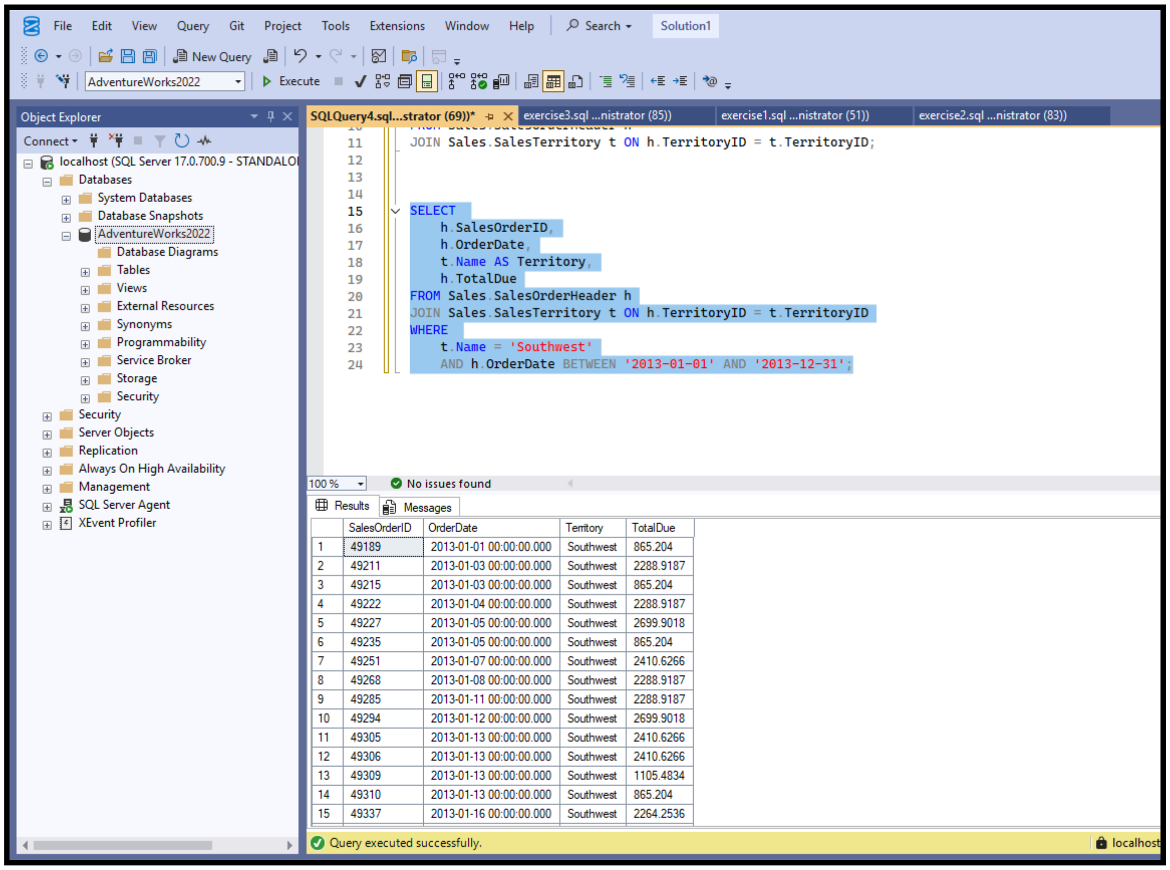
This query combines sales orders with their respective territories, allowing you to see which region each order belongs to.

2.6.4.4 Step 4: Filter by Region and Date Range

Now filter for sales in the **“Southwest”** region during the year **2013**:

```
SELECT
    h.SalesOrderID,
    h.OrderDate,
    t.Name AS Territory,
    h.TotalDue
FROM Sales.SalesOrderHeader h
JOIN Sales.SalesTerritory t ON h.TerritoryID = t.TerritoryID
WHERE
    t.Name = 'Southwest'
    AND h.OrderDate BETWEEN '2013-01-01' AND '2013-12-31';
```

> *You can try other territory names such as `'Northwest'`, `'Central'`, or `'Canada'`.*



```
JOIN Sales.SalesTerritory t ON h.TerritoryID = t.TerritoryID;



SELECT
    h.SalesOrderID,
    h.OrderDate,
    t.Name AS Territory,
    h.TotalDue
FROM Sales.SalesOrderHeader h
JOIN Sales.SalesTerritory t ON h.TerritoryID = t.TerritoryID
WHERE
    t.Name = 'Southwest'
    AND h.OrderDate BETWEEN '2013-01-01' AND '2013-12-31';
```

	SalesOrderID	OrderDate	Territory	TotalDue
1	49189	2013-01-01 00:00:00.000	Southwest	865.204
2	49211	2013-01-03 00:00:00.000	Southwest	2288.9187
3	49215	2013-01-03 00:00:00.000	Southwest	865.204
4	49222	2013-01-04 00:00:00.000	Southwest	2288.9187
5	49227	2013-01-05 00:00:00.000	Southwest	2699.9018
6	49235	2013-01-05 00:00:00.000	Southwest	865.204
7	49251	2013-01-07 00:00:00.000	Southwest	2410.6266
8	49268	2013-01-08 00:00:00.000	Southwest	2288.9187
9	49285	2013-01-11 00:00:00.000	Southwest	2288.9187
10	49294	2013-01-12 00:00:00.000	Southwest	2699.9018
11	49305	2013-01-13 00:00:00.000	Southwest	2410.6266
12	49306	2013-01-13 00:00:00.000	Southwest	2410.6266
13	49309	2013-01-13 00:00:00.000	Southwest	1105.4834
14	49310	2013-01-13 00:00:00.000	Southwest	865.204
15	49337	2013-01-16 00:00:00.000	Southwest	2264.2536

```
Query executed successfully.
```

Figure 2.2: Perform filtering data by region and date range.

2.6.4.5 Step 5: Sort Results by Total Sales Value

Add `ORDER BY` to sort by `TotalDue` in descending order:

```sql
SELECT
    h.SalesOrderID,
    h.OrderDate,
    t.Name AS Territory,
    h.TotalDue
FROM Sales.SalesOrderHeader h
JOIN Sales.SalesTerritory t ON h.TerritoryID = t.TerritoryID
WHERE
    t.Name = 'Southwest'
    AND h.OrderDate BETWEEN '2013-01-01' AND '2013-12-31'
ORDER BY h.TotalDue DESC;
```

This helps prioritize the highest value orders for analysis.

2.6.5 Summary

In this lab, you:

- Joined the `SalesOrderHeader` and `SalesTerritory` tables
- Filtered records by region and date range using `WHERE` and `BETWEEN`
- Used `ORDER BY` to sort sales by their total value
- Retrieved actionable sales insights for a specific region and time period

These skills are essential for generating regional reports and analyzing sales performance in SQL Server 2025.

2.7 Conclusion

This chapter introduced the foundational SQL skills needed to retrieve and filter data effectively. You learned how to use the `SELECT` statement, apply filters with `WHERE`, and sort results with `ORDER BY`. These skills are crucial for any data professional working with SQL Server 2025, enabling you to extract meaningful insights from your data.

3 Expressions, NULLs, and Logic

This chapter explores how to write **expressions**, handle **NULL values**, and apply **conditional logic** using `CASE`. These are essential tools for business analysts and data professionals who must generate derived insights, perform transformations, and manage missing or incomplete data.

3.1 Using Expressions in SELECT

Expressions allow you to perform calculations, manipulate strings, and format output directly in your SQL queries.

In general, expressions can be categorized into:

- Arithmetic expressions
- String expressions
- Date/time expressions

Arithmetic expressions perform calculations using numeric data types. They can include addition, subtraction, multiplication, and division.

Here's how to calculate the total price of items in an order:

```sql
SELECT ProductID, Quantity, UnitPrice, Quantity * UnitPrice AS TotalPrice
FROM Sales.OrderDetails;
```

String expressions concatenate or manipulate text data. You can combine fields, format names, or create dynamic labels.

Here's how to create a full name from first and last names:

```sql
SELECT FirstName + ' ' + LastName AS FullName
FROM Sales.Customers;
```

Date/time expressions allow you to manipulate dates, such as calculating future dates or extracting parts of a date.

Here's how to calculate an expected delivery date by adding 30 days to the order date:

```
SELECT OrderID, OrderDate,
       DATEADD(day, 30, OrderDate) AS ExpectedDelivery
FROM Sales.Orders;
```

3.2 Understanding NULLs

`NULL` represents **missing or unknown** data. It’s not the same as `0` or an empty string.

Any comparison with `NULL` results in `UNKNOWN` (not `TRUE` or `FALSE`), which affects filtering and logic.

```
SELECT *
FROM HR.Employees
WHERE ManagerID = NULL; -- This does NOT return expected rows
```

Use **`IS NULL`** or **`IS NOT NULL`**:

```
SELECT *
FROM HR.Employees
WHERE ManagerID IS NULL;
```

Here are two common functions to handle `NULL` values:

Function	Description
`ISNULL()`	Replace `NULL` with a given value
`COALESCE()`	Return the first non-null expression

Here’s how to replace `NULL` phone numbers with a default value:

```
SELECT CustomerID, ISNULL(PhoneNumber, 'Not Provided') AS ContactNumber
FROM Sales.Customers;
```

The `COALESCE` function returns the first non-null value from a list of expressions. It’s useful for providing fallback values.

```
SELECT ProductID, COALESCE(SalePrice, ListPrice, 0) AS EffectivePrice
FROM Inventory.Products;
```

3.3 The CASE Expression: Conditional Logic

The `CASE` expression allows you to perform **if-then-else** logic in SQL queries. This is useful for categorization, flagging, and conditional formatting.

Let’s start with a simple example that maps specific values to new labels.

```
SELECT
    OrderID,
    Status,
    CASE Status
        WHEN 'Completed' THEN 'Green'
        WHEN 'Pending' THEN 'Yellow'
        ELSE 'Red'
    END AS StatusColor
FROM Sales.Orders;
```

For more complex conditions, you can use the searched `CASE` syntax, which allows for multiple conditions.

```
SELECT
    CustomerID,
    TotalPurchase,
    CASE
        WHEN TotalPurchase >= 10000 THEN 'Platinum'
        WHEN TotalPurchase >= 5000 THEN 'Gold'
        WHEN TotalPurchase >= 1000 THEN 'Silver'
        ELSE 'Bronze'
    END AS LoyaltyLevel
FROM Sales.CustomerRevenue;
```

The `CASE` expression is versatile and can be used for various purposes:

- Group numeric values into buckets (e.g., sales tier)
- Convert raw codes into readable labels
- Flag risky or abnormal data

3.4 Combining Expressions, NULLs, and CASE

In real-world SQL queries, you often need to combine expressions, handle `NULL` values, and apply conditional logic all at once. This is especially important when creating calculated fields, generating business metrics, or preparing data for reports and dashboards.

By integrating arithmetic or string expressions with `CASE` statements and `NULL` handling functions, you can produce more robust and meaningful query results. The following examples demonstrate how these concepts work together to solve practical business problems.

Here’s how to categorize customer revenue while handling potential `NULL` values:

```
SELECT
    CustomerID,
    Revenue,
    CASE
        WHEN Revenue IS NULL THEN 'Unknown'
        WHEN Revenue > 10000 THEN 'High'
        ELSE 'Normal'
    END AS RevenueStatus
FROM Sales.Customers;
```

This query checks if the `Revenue` is `NULL` and categorizes it accordingly, ensuring that all customers are accounted for, even those with missing data.

A more complex example that combines multiple concepts:

```
SELECT
    OrderID,
    Quantity,
    UnitPrice,
    Quantity * UnitPrice AS TotalAmount,
    CASE
        WHEN Quantity * UnitPrice >= 5000 THEN 'Large Order'
        ELSE 'Standard'
    END AS OrderSize
FROM Sales.OrderDetails;
```

This query calculates the total amount for each order and categorizes it as either a “Large Order” or “Standard” based on the total value, demonstrating how to derive insights from existing data while applying conditional logic.

3.5 Exercise 5: Add Calculated Columns and Handle Missing Data

3.5.1 Description

In this hands-on lab, you will learn how to add **calculated columns** in SQL queries using **expressions**, such as arithmetic operations and string concatenation. You will also learn how to handle **missing or NULL values** using the `IS NULL` condition and the `COALESCE()` function. These techniques help create cleaner, business-ready output from raw data.

3.5.2 Objectives

- Create derived columns using arithmetic and string expressions
- Use the `IS NULL` condition to identify missing data
- Apply `COALESCE()` to substitute default values for NULLs
- Enhance result sets for reporting and analytics

3.5.3 Prerequisites

- SQL Server 2025 Developer Edition is installed and running
- SSMS 21.x is installed
- AdventureWorks2022 database is restored (from **Exercise 1**)
- Familiarity with basic `SELECT` and `WHERE` clauses

3.5.4 Steps

Here’s a step-by-step guide to complete this exercise:

3.5.4.1 Step 1: Open a New Query Window in SSMS

1. Launch **SQL Server Management Studio (SSMS)**
2. Connect to your SQL Server instance
3. Right-click on the `AdventureWorks2022` database → **New Query**
4. Ensure that the database context is set to `AdventureWorks2022`

3.5.4.2 Step 2: Calculate Discounted List Price

Let’s assume a 10% promotional discount on all products. You can calculate the discounted price with a derived column:

```sql
SELECT
    Name,
    ListPrice,
    ListPrice * 0.9 AS DiscountedPrice
FROM Production.Product
WHERE ListPrice > 0;
```

> *This adds a new column called `DiscountedPrice` without altering the actual table.*

3.5.4.3 Step 3: Combine First and Last Names

Use `+` to concatenate `FirstName` and `LastName` into a full name:

```sql
SELECT
    FirstName + ' ' + LastName AS FullName,
    EmailPromotion
FROM Person.Person;
```

3.5.4.4 Step 4: Identify NULL Values Using `IS NULL`

Let’s find products with **no color information**:

```
SELECT
    Name,
    Color
FROM Production.Product
WHERE Color IS NULL;
```

> *This query returns products where the `Color` column is missing (i.e., NULL).*

3.5.4.5 Step 5: Replace NULL Values Using `COALESCE`

To improve output for reporting, use `COALESCE()` to replace NULLs with a default value:

```
SELECT
    Name,
    COALESCE(Color, 'Not Specified') AS ProductColor
FROM Production.Product;
```

> *`COALESCE(Color, 'Not Specified')` returns `'Not Specified'` when `Color` is NULL.*

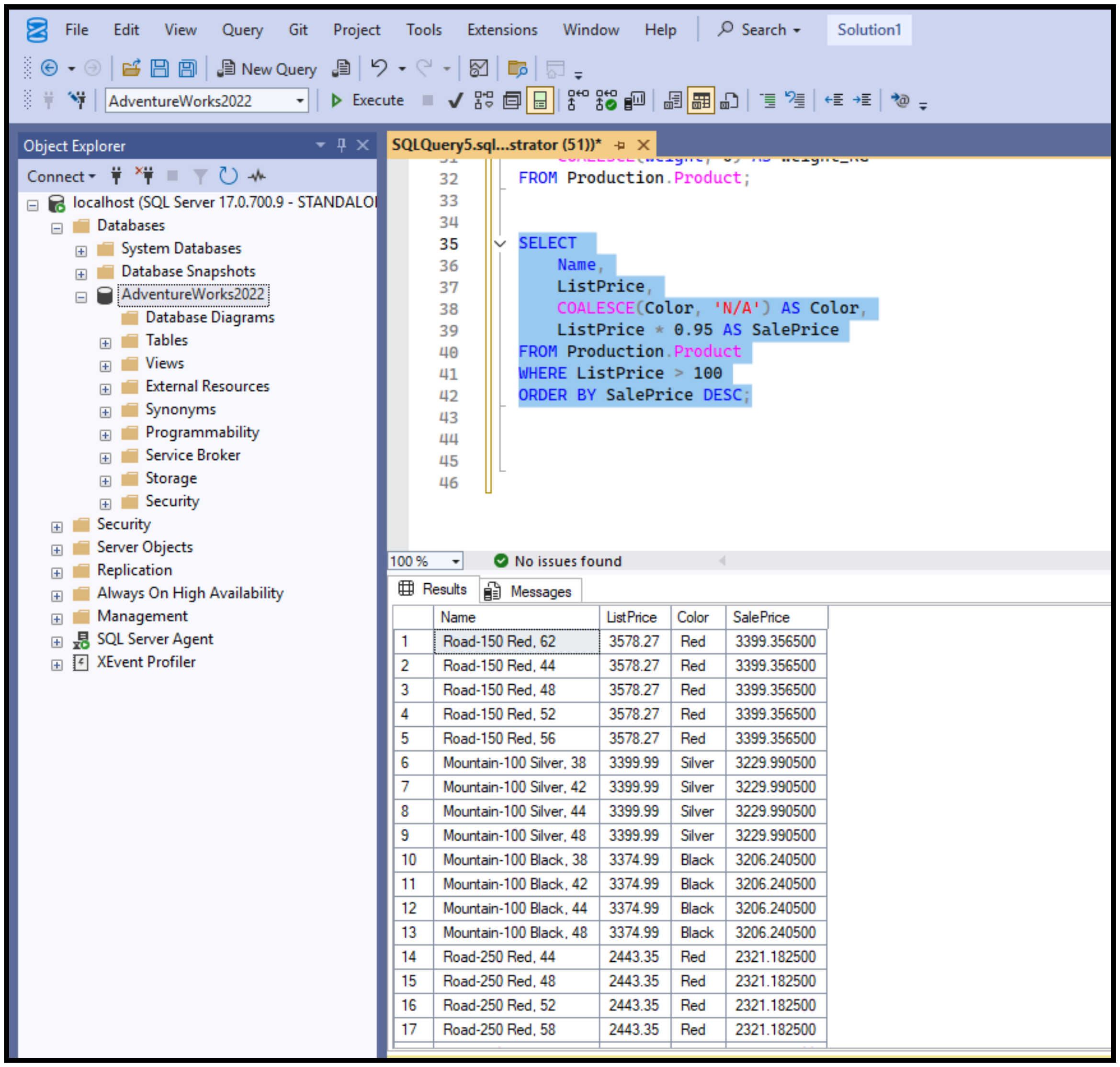
You can also use this for numeric or date columns:

```
SELECT
    ProductID,
    Weight,
    COALESCE(Weight, 0) AS Weight_KG
FROM Production.Product;
```

3.5.4.6 Step 6: Combine Expressions

Let’s combine calculations and NULL handling:

```
SELECT
    Name,
    ListPrice,
    COALESCE(Color, 'N/A') AS Color,
    ListPrice * 0.95 AS SalePrice
FROM Production.Product
WHERE ListPrice > 100
ORDER BY SalePrice DESC;
```



```sql
FROM Production.Product;


SELECT
    Name,
    ListPrice,
    COALESCE(Color, 'N/A') AS Color,
    ListPrice * 0.95 AS SalePrice
FROM Production.Product
WHERE ListPrice > 100
ORDER BY SalePrice DESC;
```

	Name	ListPrice	Color	SalePrice
1	Road-150 Red, 62	3578.27	Red	3399.356500
2	Road-150 Red, 44	3578.27	Red	3399.356500
3	Road-150 Red, 48	3578.27	Red	3399.356500
4	Road-150 Red, 52	3578.27	Red	3399.356500
5	Road-150 Red, 56	3578.27	Red	3399.356500
6	Mountain-100 Silver, 38	3399.99	Silver	3229.990500
7	Mountain-100 Silver, 42	3399.99	Silver	3229.990500
8	Mountain-100 Silver, 44	3399.99	Silver	3229.990500
9	Mountain-100 Silver, 48	3399.99	Silver	3229.990500
10	Mountain-100 Black, 38	3374.99	Black	3206.240500
11	Mountain-100 Black, 42	3374.99	Black	3206.240500
12	Mountain-100 Black, 44	3374.99	Black	3206.240500
13	Mountain-100 Black, 48	3374.99	Black	3206.240500
14	Road-250 Red, 44	2443.35	Red	2321.182500
15	Road-250 Red, 48	2443.35	Red	2321.182500
16	Road-250 Red, 52	2443.35	Red	2321.182500
17	Road-250 Red, 58	2443.35	Red	2321.182500

Figure 3.1: Perform query to combine expression.

3.5.5 Summary

In this lab, you:

- Created calculated columns using arithmetic and string operations
- Identified missing data using `IS NULL`
- Used `COALESCE()` to replace NULLs with default values
- Combined expressions for enhanced output

These techniques are useful for transforming raw data into business-friendly reports and preparing it for dashboards or exports.

3.6 Exercise 6: Use CASE for Business Rule Logic

3.6.1 Description

In this hands-on lab, you will learn how to use the `CASE` expression in SQL Server 2025 to apply **conditional logic in query results**. The `CASE` expression allows you to implement simple business rules directly in your SQL queries—without writing procedural code. You'll use data from the **AdventureWorks2022** database to classify product pricing and evaluate sales performance.

3.6.2 Objectives

- Understand how to use the `CASE` expression in `SELECT` statements
- Apply conditional logic to classify numeric and text-based fields
- Enhance SQL output with human-readable rule-based columns
- Practice combining `CASE` with filtering and sorting logic

3.6.3 Prerequisites

- SQL Server 2025 Developer Edition installed and running
- SSMS 21.x installed
- AdventureWorks2022 database restored (from **Exercise 1**)
- Familiarity with basic `SELECT` and `WHERE` clauses

3.6.4 Steps

Here's a step-by-step guide to complete this exercise:

3.6.4.1 Step 1: Open a New Query Window in SSMS

1. Launch **SQL Server Management Studio (SSMS)**
2. Connect to your SQL Server instance
3. In Object Explorer, right-click on **AdventureWorks2022** → choose **New Query**
4. Ensure that the database context is set to `AdventureWorks2022`

3.6.4.2 Step 2: Classify Products by Price Tier

Use `CASE` to group products based on their `ListPrice` into categories:

```sql
SELECT
    Name,
    ListPrice,
    CASE
        WHEN ListPrice = 0 THEN 'Free'
        WHEN ListPrice < 100 THEN 'Low Price'
        WHEN ListPrice BETWEEN 100 AND 500 THEN 'Mid Range'
        WHEN ListPrice > 500 THEN 'Premium'
        ELSE 'Unknown'
    END AS PriceCategory
FROM Production.Product
ORDER BY ListPrice;
```

> 💡 *This creates a new column `PriceCategory` based on product pricing logic.*

3.6.4.3 Step 3: Show Promotion Status for Customers

Use `CASE` to display a human-readable status from the `EmailPromotion` field:

```sql
SELECT
    FirstName + ' ' + LastName AS FullName,
    EmailPromotion,
    CASE
        WHEN EmailPromotion = 0 THEN 'No Promotions'
        WHEN EmailPromotion = 1 THEN 'Subscribed - Basic'
        WHEN EmailPromotion = 2 THEN 'Subscribed - Advanced'
        ELSE 'Unknown'
    END AS PromotionStatus
FROM Person.Person;
```

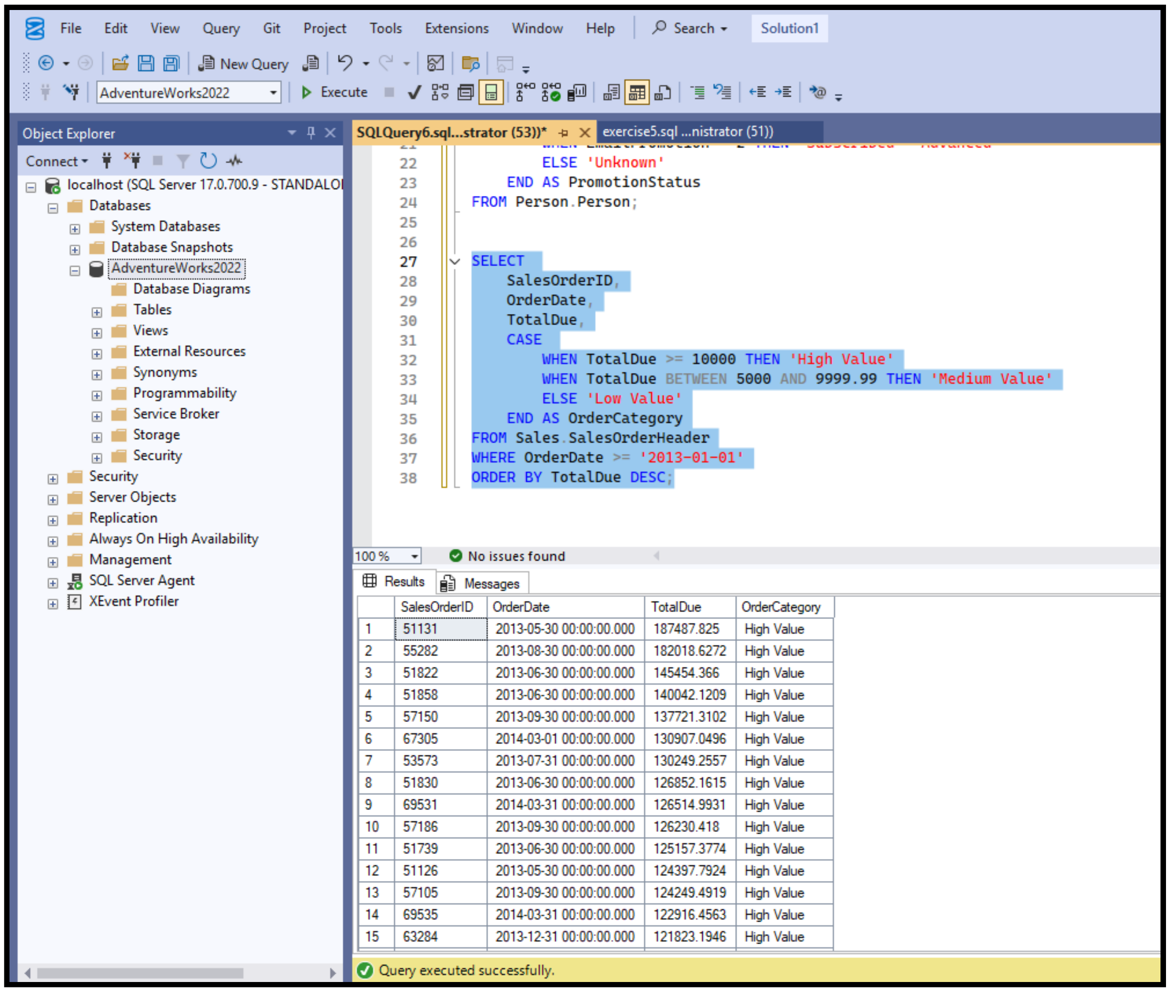
> *`EmailPromotion` is an integer, and the `CASE` expression maps it to readable labels.*

3.6.4.4 Step 4: Flag High-Value Sales Orders

Use `CASE` to add a flag column to identify large sales:

```sql
SELECT
    SalesOrderID,
    OrderDate,
    TotalDue,
    CASE
        WHEN TotalDue >= 10000 THEN 'High Value'
        WHEN TotalDue BETWEEN 5000 AND 9999.99 THEN 'Medium Value'
        ELSE 'Low Value'
    END AS OrderCategory
FROM Sales.SalesOrderHeader
WHERE OrderDate >= '2013-01-01'
ORDER BY TotalDue DESC;
```

You can use this logic for dashboards, alerts, or reports.



```sql
            ELSE 'Unknown'
        END AS PromotionStatus
    FROM Person.Person;


    SELECT
        SalesOrderID,
        OrderDate,
        TotalDue,
        CASE
            WHEN TotalDue >= 10000 THEN 'High Value'
            WHEN TotalDue BETWEEN 5000 AND 9999.99 THEN 'Medium Value'
            ELSE 'Low Value'
        END AS OrderCategory
    FROM Sales.SalesOrderHeader
    WHERE OrderDate >= '2013-01-01'
    ORDER BY TotalDue DESC;
```

	SalesOrderID	OrderDate	TotalDue	OrderCategory
1	51131	2013-05-30 00:00:00.000	187487.825	High Value
2	55282	2013-08-30 00:00:00.000	182018.6272	High Value
3	51822	2013-06-30 00:00:00.000	145454.366	High Value
4	51858	2013-06-30 00:00:00.000	140042.1209	High Value
5	57150	2013-09-30 00:00:00.000	137721.3102	High Value
6	67305	2014-03-01 00:00:00.000	130907.0496	High Value
7	53573	2013-07-31 00:00:00.000	130249.2557	High Value
8	51830	2013-06-30 00:00:00.000	126852.1615	High Value
9	69531	2014-03-31 00:00:00.000	126514.9931	High Value
10	57186	2013-09-30 00:00:00.000	126230.418	High Value
11	51739	2013-06-30 00:00:00.000	125157.3774	High Value
12	51126	2013-05-30 00:00:00.000	124397.7924	High Value
13	57105	2013-09-30 00:00:00.000	124249.4919	High Value
14	69535	2014-03-31 00:00:00.000	122916.4563	High Value
15	63284	2013-12-31 00:00:00.000	121823.1946	High Value

No issues found

Query executed successfully.

Figure 3.2: Perform query to flag high-value sales orders.

3.6.5 Summary

In this lab, you:

- Used the `CASE` expression to implement conditional logic in query results
- Created rule-based output fields such as `PriceCategory`, `PromotionStatus`, and `OrderCategory`
- Combined `CASE` with sorting and filtering for better insights

`CASE` is a powerful tool for translating business logic directly into your SQL queries — a must-have skill for analysts and developers working with SQL Server 2025.

3.7 Conclusion

In this chapter, we explored how to use expressions, handle NULL values, and apply conditional logic with the `CASE` expression in SQL Server 2025. These techniques are essential for transforming raw data into meaningful insights, enabling better decision-making and reporting.

Section 3: Data Modeling and Design

4 Relational Database Design Basics

This chapter introduces essential concepts of relational database design. We'll explore **tables**, **data types**, **keys**, and the principles of **normalization (1NF to 3NF)**. These concepts ensure that business data is stored efficiently, consistently, and with minimal redundancy—key goals in enterprise data systems.

4.1 Tables and Data Types

We'll start with the fundamental building blocks of relational databases: **tables** and **data types**. Understanding these concepts is crucial for designing effective database schemas that meet business requirements.

4.1.1 What is a Table?

A **table** is a structured set of data made up of rows and columns. Each **row** (or record) represents a single entity instance, and each **column** holds a specific attribute.

Here's a simple example of a `Customers` table:

CustomerID	FirstName	LastName	Email	Region
1	Indah	Chen	indah@ilmudata.id	West
2	Jane	Brown	jane@ilmudata.id	South

4.1.2 Data Types in SQL Server 2025

SQL Server provides a rich set of data types to define what kind of data can be stored in each column.

Here are some common data types used in SQL Server 2025:

Category	Data Types
Numbers	`INT`, `BIGINT`, `DECIMAL(p,s)`, `FLOAT`

Category	**Data Types**
Characters	`CHAR(n)`, `VARCHAR(n)`, `TEXT`
Date/Time	`DATE`, `DATETIME2`, `TIME`
Logical	`BIT`
Unique IDs	`UNIQUEIDENTIFIER`

Following is a SQL statement to create a `Customers` table with various data types:

```sql
CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50),
    Email VARCHAR(100) UNIQUE,
    Region VARCHAR(30),
    CreatedAt DATETIME2 DEFAULT GETDATE()
);
```

4.2 Keys in Relational Tables

Keys are essential for uniquely identifying records in a table and establishing relationships between tables. They help maintain data integrity and enforce business rules.

4.2.1 Primary Key (PK)

A **Primary Key** is a column (or set of columns) that uniquely identifies each row in a table. It ensures that no two rows can have the same value in the primary key column(s).

Here’s how to define a primary key in SQL Server:

```sql
CONSTRAINT PK_Customers PRIMARY KEY (CustomerID)
```

4.2.2 Foreign Key (FK)

A **Foreign Key** is a column (or set of columns) that creates a link between two tables. It enforces referential integrity by ensuring that the value in the foreign key column matches a value in the primary key column of another table.

Here’s how to define a foreign key in SQL Server:

```
FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
```

4.2.3 Candidate Key

A **Candidate Key** is any column (or set of columns) that can uniquely identify a row in a table. A table can have multiple candidate keys, but only one is chosen as the primary key.

4.2.4 Surrogate Key vs. Natural Key

A **Surrogate Key** is an artificial key created to uniquely identify a record, often using an auto-incrementing integer or a unique identifier (GUID). A **Natural Key** is a real-world attribute that naturally identifies a record, such as a Social Security Number or email address.

4.3 Introduction to Normalization

Normalization is the process of organizing data in a database to reduce redundancy and improve data integrity. It involves decomposing tables into smaller, related tables and defining relationships between them.

Normalization is a design process that organizes data to:

- Minimize redundancy,
- Avoid update anomalies,
- Improve integrity.

4.3.1 First Normal Form (1NF)

First Normal Form (1NF) requires that all columns in a table contain atomic values (indivisible) and that each column contains only one value per row. This means no repeating groups or arrays.

Here’s an example of a table that violates 1NF:

OrderID	Customer	ProductNames
1	John	TV, Laptop, Headphones

To convert this table to 1NF, we need to ensure that each product is stored in a separate row:

OrderID	**Customer**	**ProductName**
1	John	TV
1	John	Laptop
1	John	Headphones

4.4 Second Normal Form (2NF)

Second Normal Form (2NF) builds on 1NF by ensuring that all non-key attributes are fully functionally dependent on the entire primary key. This means eliminating partial dependencies, where a non-key attribute depends only on part of a composite primary key.

Follow these rules to achieve 2NF:

- Be in **1NF**, and
- **All non-key attributes must depend on the entire primary key** (eliminate partial dependencies).

Here's a table that violates 2NF:

In a table with a composite key:

```sql
CREATE TABLE OrderDetails (
    OrderID INT,
    ProductID INT,
    ProductName VARCHAR(100), -- partial dependency
    Quantity INT,
    PRIMARY KEY (OrderID, ProductID)
);
```

This table has a composite primary key (`OrderID`, `ProductID`), but `ProductName` only depends on `ProductID`, not the entire key.

To fix this, we need to move `ProductName` to a separate table:

- Move `ProductName` to a separate `Products` table.

```
CREATE TABLE Products (
    ProductID INT PRIMARY KEY,
    ProductName VARCHAR(100)
);
```

4.5 Third Normal Form (3NF)

Third Normal Form (3NF) goes a step further by ensuring that all non-key attributes are not only fully functionally dependent on the primary key but also that there are no transitive dependencies. This means that non-key attributes should not depend on other non-key attributes.

To achieve 3NF, a table must:

- Be in **2NF**, and
- **No transitive dependencies** (non-key attributes must depend only on the key).

Here’s a table that violates 3NF:

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    Name VARCHAR(50),
    DepartmentID INT,
    DepartmentName VARCHAR(50) -- transitive dependency
);
```

In this example, `DepartmentName` depends on `DepartmentID`, which is not the primary key. This creates a transitive dependency.

To fix this, we need to remove the transitive dependency by moving `DepartmentName` to a separate table:

- Move `DepartmentName` to a separate `Departments` table.

```
CREATE TABLE Departments (
    DepartmentID INT PRIMARY KEY,
    DepartmentName VARCHAR(50)
);
```

4.6 Exercise 7: Basic ERD Design

4.6.1 Description

Understanding how to represent different types of relationships in SQL Server is essential for effective relational database design. In this hands-on lab, we'll explore four fundamental relationship types—**one-to-one**, **one-to-many**, **many-to-many**, and **self-referencing**—and implement them in a SQL Server 2025 database. Each relationship will be implemented through proper table creation and foreign key constraints.

4.6.2 Objectives

- Learn how to create and enforce:
    - One-to-One relationships
    - One-to-Many relationships
    - Many-to-Many relationships (via junction tables)
    - Self-referencing relationships (hierarchical)
- Understand when and why to use each type

4.6.3 Prerequisites

- SQL Server 2025 Developer Edition
- SQL Server Management Studio (SSMS) 21.x or higher
- Basic knowledge of `CREATE TABLE` and `FOREIGN KEY` syntax

4.6.4 Steps

Here's a step-by-step guide to implementing these relationships in SQL Server. Each step includes the SQL code needed to create the tables and relationships, along with explanations of the design choices made.

4.6.4.1 Step 1: Create the Database

We begin by creating a new sandbox database called `ERDDesignDemo`.

```
CREATE DATABASE ERDDesignDemo;
GO

USE ERDDesignDemo;
GO
```

4.6.4.2 Step 2: One-to-One Relationship (Employee ↔ EmployeeDetail)

A **one-to-one relationship** means that each row in one table is linked to a single row in another.

We'll simulate an `Employee` table with a one-to-one `EmployeeDetail`.

```
CREATE TABLE Employee (
    EmployeeID INT PRIMARY KEY,
    FullName NVARCHAR(100),
    HireDate DATE
);

CREATE TABLE EmployeeDetail (
    EmployeeID INT PRIMARY KEY,
    Address NVARCHAR(200),
    Phone NVARCHAR(20),
    FOREIGN KEY (EmployeeID) REFERENCES Employee(EmployeeID)
);
```

📝 Explanation:

- `EmployeeDetail.EmployeeID` is both a primary and foreign key, ensuring one-to-one mapping.

4.6.4.3 Step 3: One-to-Many Relationship (Customer → Orders)

A **one-to-many relationship** means one record in a table is linked to multiple records in another.

```
CREATE TABLE Customer (
    CustomerID INT PRIMARY KEY,
    CustomerName NVARCHAR(100)
);

CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    CustomerID INT,
    OrderDate DATE,
    FOREIGN KEY (CustomerID) REFERENCES Customer(CustomerID)
);
```

📝 Explanation:

- One customer can have many orders, but each order belongs to one customer.

4.6.4.4 Step 4: Many-to-Many Relationship (Student ↔ Course)

To model **many-to-many**, we use a **junction table** between `Student` and `Course`.

```sql
CREATE TABLE Student (
    StudentID INT PRIMARY KEY,
    StudentName NVARCHAR(100)
);

CREATE TABLE Course (
    CourseID INT PRIMARY KEY,
    CourseName NVARCHAR(100)
);

CREATE TABLE StudentCourse (
    StudentID INT,
    CourseID INT,
    PRIMARY KEY (StudentID, CourseID),
    FOREIGN KEY (StudentID) REFERENCES Student(StudentID),
    FOREIGN KEY (CourseID) REFERENCES Course(CourseID)
);
```

📝 Explanation:

- A student can enroll in many courses, and a course can have many students. The `StudentCourse` table manages the link.

4.6.4.5 Step 5: Self-Referencing Relationship (Employee → Manager)

A **self-referencing relationship** models hierarchical structures, like managers in an organization.

```sql
CREATE TABLE OrgEmployee (
    EmpID INT PRIMARY KEY,
    EmpName NVARCHAR(100),
    ManagerID INT NULL,
    FOREIGN KEY (ManagerID) REFERENCES OrgEmployee(EmpID)
);
```

📝 Explanation:

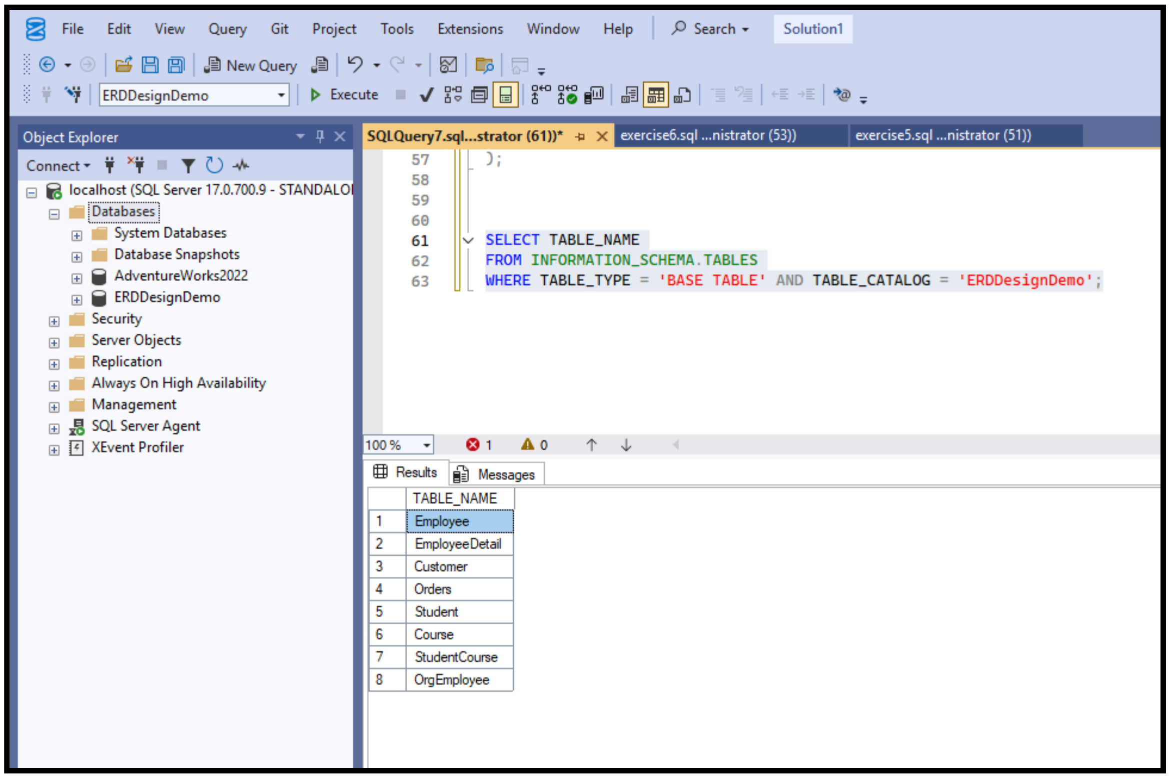
- The `ManagerID` column refers back to `EmpID`, enabling an employee to report to another employee.

4.6.4.6 Step 6: Verify the Schema Structure

Run a query to validate the structure by listing all created tables.

```sql
SELECT TABLE_NAME
FROM INFORMATION_SCHEMA.TABLES
WHERE TABLE_TYPE = 'BASE TABLE' AND TABLE_CATALOG = 'ERDDesignDemo';
```

This confirms the objects exist in the current database.



```sql
);



SELECT TABLE_NAME
FROM INFORMATION_SCHEMA.TABLES
WHERE TABLE_TYPE = 'BASE TABLE' AND TABLE_CATALOG = 'ERDDesignDemo';
```

	TABLE_NAME
1	Employee
2	EmployeeDetail
3	Customer
4	Orders
5	Student
6	Course
7	StudentCourse
8	OrgEmployee

Figure 4.1: Verifiying the Schema Structure.

4.6.5 Step 7: Show ERD Diagram

To visualize the relationships, you can use SQL Server Management Studio’s built-in diagram feature: 1. Right-click on the `Database Diagrams` folder in `ERDDesignDemo`. 2. Select **New Database Diagram**. 3. Add the tables you created. 4. Arrange them to show the relationships visually.

This helps you see how the tables relate to each other in a graphical format.

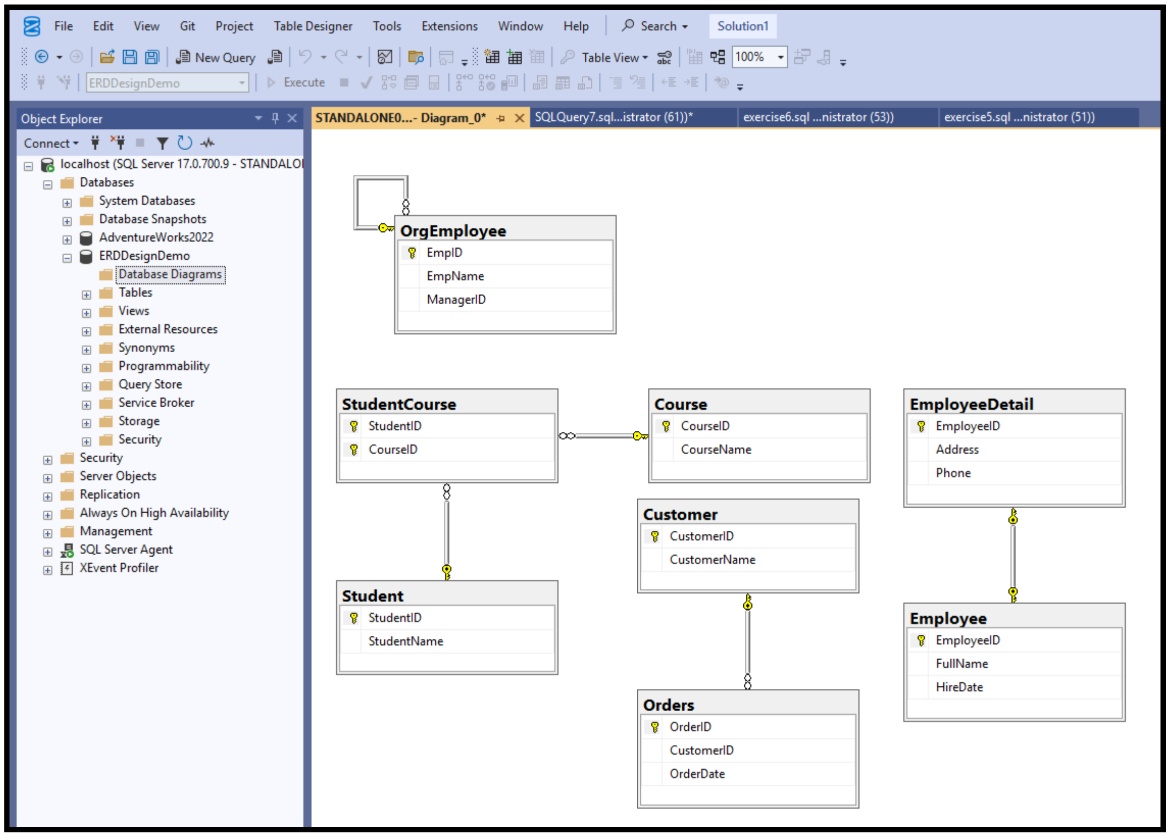


Figure 4.2: Showing ERD diagram for ERDDesignDemo.

4.6.6 Summary

In this hands-on lab, you've learned how to implement the most fundamental database relationship types in SQL Server:

Relationship Type	**Example Tables**
One-to-One	`Employee` ↔ `EmployeeDetail`
One-to-Many	`Customer` → `Orders`
Many-to-Many	`Student` ↔ `Course` via `StudentCourse`
Self-Referencing	`OrgEmployee` → `ManagerID`

Understanding and applying these patterns is crucial when designing normalized, efficient relational models for real-world applications.

4.7 Exercise 8: Design Schema for a Subscription Business

4.7.1 Description

In this lab, you’ll walk through designing a relational database schema for a basic **subscription-based business**. You’ll start from identifying business entities, normalize them into separate tables, and build relationships that comply with **Third Normal Form (3NF)**. You’ll implement the design by creating tables in a new SQL Server 2025 database.

4.7.2 Objectives

- Analyze a real-world business scenario into logical database entities
- Normalize the data to eliminate redundancy and maintain data integrity
- Use SQL Server 2025 to create normalized tables and define relationships
- Apply primary and foreign keys for relational consistency

4.7.3 Prerequisites

- SQL Server 2025 Developer Edition installed and running
- SSMS 21.x installed
- Basic SQL DDL (Data Definition Language) knowledge
- Familiarity with primary/foreign keys and normalization principles

4.7.4 Steps

Here’s a step-by-step guide to designing the schema for a subscription business:

4.7.4.1 Step 1: Understand the Business Requirements

We are modeling a subscription-based service where users can subscribe to different plans. Each subscription is billed monthly.

Entities Identified:

- **Customer**: who subscribes
- **SubscriptionPlan**: the available subscription types (Basic, Premium)
- **Subscription**: the link between customer and plan

- **Payment**: monthly billing records

> *Each of these will become a table.*

4.7.4.2 Step 2: Create a New Database

Create a new isolated database named `SubscriptionDB` to contain our schema.

```
CREATE DATABASE SubscriptionDB;
GO

USE SubscriptionDB;
GO
```

> *This step ensures we don't affect other existing databases.*

4.7.4.3 Step 3: Create `Customer` Table (1NF Compliant)

What we're doing: Define a table that stores customer personal information with atomic values.

```
CREATE TABLE Customer (
    CustomerID INT PRIMARY KEY IDENTITY,
    FirstName NVARCHAR(100) NOT NULL,
    LastName NVARCHAR(100) NOT NULL,
    Email NVARCHAR(255) UNIQUE NOT NULL,
    JoinDate DATE DEFAULT GETDATE()
);
```

> *The `Email` column is set as unique. `JoinDate` defaults to current date.*

4.7.4.4 Step 4: Create `SubscriptionPlan` Table

Store plan types, pricing, and billing frequency. This helps normalize data instead of storing this info in every subscription record.

```
CREATE TABLE SubscriptionPlan (
    PlanID INT PRIMARY KEY IDENTITY,
    PlanName NVARCHAR(50) NOT NULL,
    MonthlyPrice DECIMAL(10,2) NOT NULL,
    Description NVARCHAR(255)
);
```

This allows easy management of plan changes and reuse across customers.

4.7.4.5 Step 5: Create `Subscription` Table (2NF and 3NF Compliant)

This table links each customer to a plan. It contains foreign keys to both `Customer` and `SubscriptionPlan`.

```
CREATE TABLE Subscription (
    SubscriptionID INT PRIMARY KEY IDENTITY,
    CustomerID INT NOT NULL,
    PlanID INT NOT NULL,
    StartDate DATE NOT NULL,
    EndDate DATE NULL,
    IsActive BIT NOT NULL DEFAULT 1,
    FOREIGN KEY (CustomerID) REFERENCES Customer(CustomerID),
    FOREIGN KEY (PlanID) REFERENCES SubscriptionPlan(PlanID)
);
```

This avoids redundancy and maintains clear entity relationships.

4.7.4.6 Step 6: Create `Payment` Table for Monthly Billing

Track each payment per subscription per billing cycle. This further normalizes the schema for scalability.

```
CREATE TABLE Payment (
    PaymentID INT PRIMARY KEY IDENTITY,
    SubscriptionID INT NOT NULL,
    PaymentDate DATE NOT NULL,
    Amount DECIMAL(10,2) NOT NULL,
    PaymentStatus NVARCHAR(50) CHECK (PaymentStatus IN ('Paid', 'Failed', 'Pending')),
    FOREIGN KEY (SubscriptionID) REFERENCES Subscription(SubscriptionID)
);
```

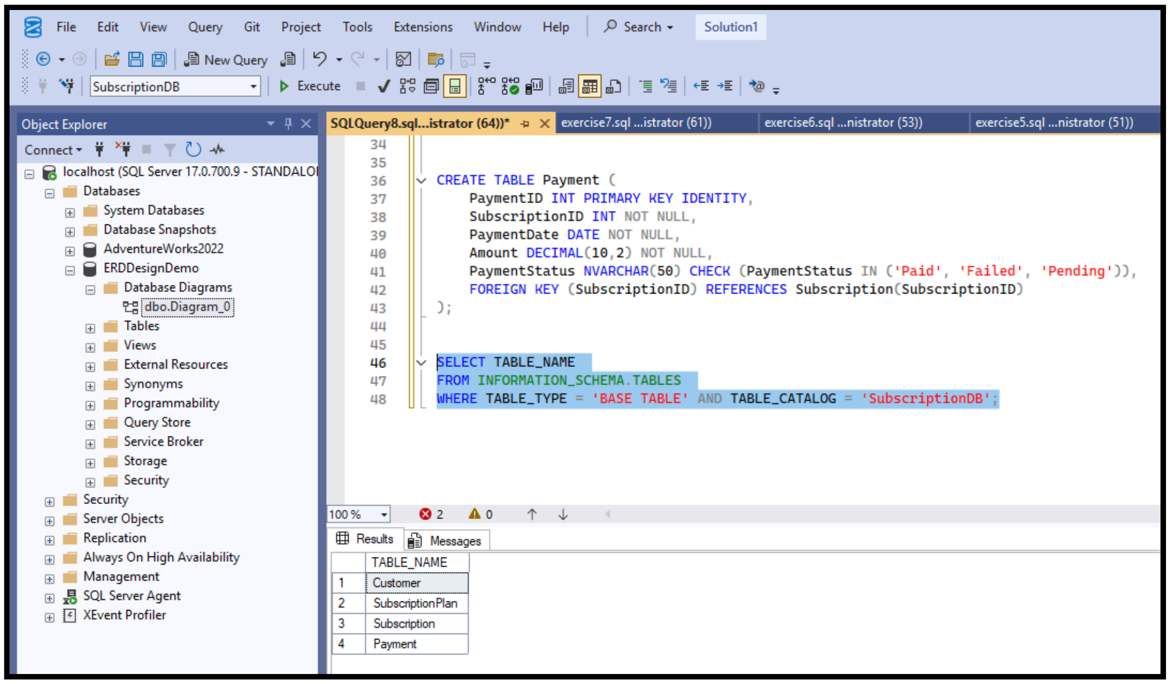
This structure allows multiple payments per subscription and clear billing history.

4.7.4.7 Step 7: Verify the Schema Structure

Run a query to validate the structure by listing all created tables.

```sql
SELECT TABLE_NAME
FROM INFORMATION_SCHEMA.TABLES
WHERE TABLE_TYPE = 'BASE TABLE' AND TABLE_CATALOG = 'SubscriptionDB';
```

> *This confirms the objects exist in the current database.*



```sql
CREATE TABLE Payment (
    PaymentID INT PRIMARY KEY IDENTITY,
    SubscriptionID INT NOT NULL,
    PaymentDate DATE NOT NULL,
    Amount DECIMAL(10,2) NOT NULL,
    PaymentStatus NVARCHAR(50) CHECK (PaymentStatus IN ('Paid', 'Failed', 'Pending')),
    FOREIGN KEY (SubscriptionID) REFERENCES Subscription(SubscriptionID)
);


SELECT TABLE_NAME
FROM INFORMATION_SCHEMA.TABLES
WHERE TABLE_TYPE = 'BASE TABLE' AND TABLE_CATALOG = 'SubscriptionDB';
```

```
   TABLE_NAME
1  Customer
2  SubscriptionPlan
3  Subscription
4  Payment
```

Figure 4.3: Showing all tables for SubscriptionDB database.

4.7.5 Step 8: Show ERD Diagram

To visualize the relationships, you can use SQL Server Management Studio’s built-in diagram feature: 1. Right-click on the `Database Diagrams` folder in `SubscriptionDB`. 2. Select **New Database Diagram**. 3. Add the tables you created. 4. Arrange them to show the relationships visually.

> *This helps you see how the tables relate to each other in a graphical format.*

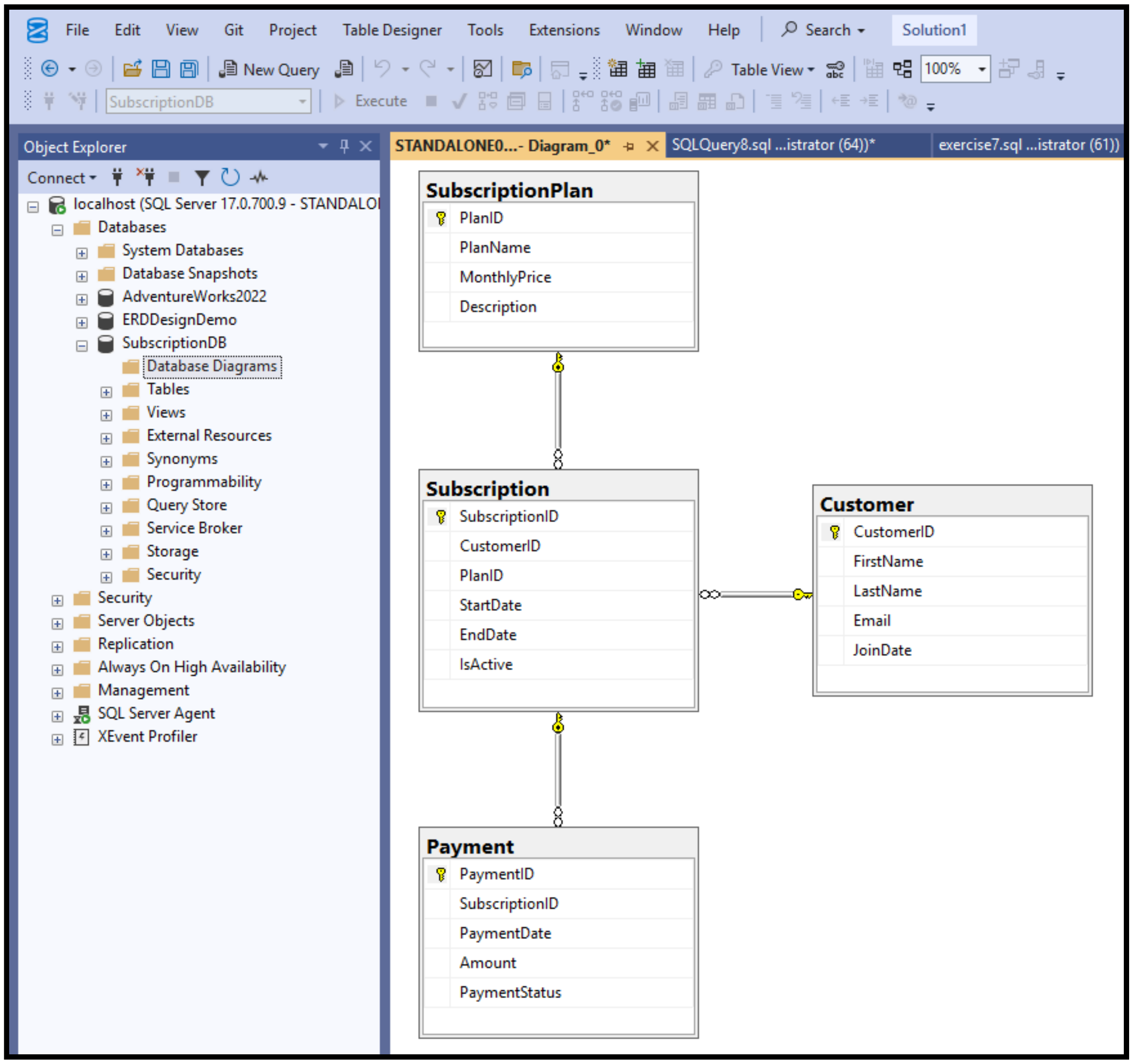


Figure 4.4: Showing ERD diagram for SubscriptionDB.

4.7.6 Summary

In this hands-on lab, you:

- Identified key entities and their relationships for a subscription business
- Applied **1NF**, **2NF**, and **3NF** principles to structure your tables
- Created tables using SQL Server 2025 with proper **primary and foreign keys**
- Designed a scalable and clean schema ready for query, analytics, and compliance

This foundational schema can now be extended for reporting, analytics, and business rules in upcoming chapters.

4.8 Exercise 9: Insert and Query Sample Data

4.8.1 Description

This lab guides you through inserting sample records into the subscription database schema you created in the previous lab. After populating tables, you'll verify relational integrity using `SELECT` queries and `JOIN` clauses across related tables.

4.8.2 Objectives

- Populate normalized tables using SQL `INSERT` statements
- Verify referential integrity and relationships using `INNER JOIN` queries
- Understand how table relations reflect real-world business data

4.8.3 Prerequisites

- SQL Server 2025 Developer Edition is installed and running
- SSMS 21.x is installed
- `SubscriptionDB` schema from **Exercise 8** is already created
- Basic familiarity with SQL `INSERT` and `SELECT` statements

4.8.4 Steps

Here's a step-by-step guide to complete this exercise:

4.8.4.1 Step 1: Connect and Use the `SubscriptionDB`

Connect to SQL Server and ensure we're using the correct database.

```
USE SubscriptionDB;
GO
```

This ensures all operations are scoped within the correct database.

4.8.4.2 Step 2: Insert Sample Data into `SubscriptionPlan`

Add subscription plans customers can choose from.

```sql
INSERT INTO SubscriptionPlan (PlanName, MonthlyPrice, Description)
VALUES
('Basic Plan', 9.99, 'Access to standard features'),
('Premium Plan', 19.99, 'Access to all features including analytics'),
('Enterprise Plan', 49.99, 'Custom enterprise support and reporting');
```

This query inserts three different subscription plans into the `SubscriptionPlan` table.

> *Each plan has a name, price, and description. These are referenced in subscriptions.*

4.8.4.3 Step 3: Insert Sample Data into `Customer`

Add some example customers to simulate real subscribers.

```sql
INSERT INTO Customer (FirstName, LastName, Email)
VALUES
('Linda', 'Alice', 'linda.alice@ilmudata.id'),
('Ujang', 'Smith', 'ujang.smith@ilmudata.id'),
('Cindy', 'Lee', 'cindy.lee@ilmudata.id');
```

This inserts three customers into the `Customer` table.

> *The `Email` field must remain unique per customer.*

4.8.4.4 Step 4: Insert Data into `Subscription`

Link customers to plans, simulating active subscriptions.

```sql
-- Ujang subscribes to Basic
INSERT INTO Subscription (CustomerID, PlanID, StartDate)
VALUES (1, 1, '2025-07-01');

-- Ujang subscribes to Premium
INSERT INTO Subscription (CustomerID, PlanID, StartDate)
VALUES (2, 2, '2025-07-01');

-- Cindy subscribes to Enterprise
INSERT INTO Subscription (CustomerID, PlanID, StartDate)
VALUES (3, 3, '2025-07-01');
```

> *Ensure `CustomerID` and `PlanID` match those inserted earlier. You can verify with `SELECT` if unsure.*

4.8.4.5 Step 5: Insert Sample Data into `Payment`

Simulate first billing cycle payments for each subscription.

```sql
INSERT INTO Payment (SubscriptionID, PaymentDate, Amount, PaymentStatus)
VALUES
(1, '2025-07-05', 9.99, 'Paid'),
(2, '2025-07-05', 19.99, 'Paid'),
(3, '2025-07-05', 49.99, 'Pending');
```

> *This gives each subscription an associated billing record.*

4.8.4.6 Step 6: Query Data with Joins to Verify Relationships

Run `JOIN` queries to inspect data across tables and verify referential integrity.

```sql
-- List subscriptions with customer and plan info
SELECT
    s.SubscriptionID,
    c.FirstName + ' ' + c.LastName AS CustomerName,
    sp.PlanName,
    sp.MonthlyPrice,
    s.StartDate,
    s.IsActive
FROM Subscription s
JOIN Customer c ON s.CustomerID = c.CustomerID
JOIN SubscriptionPlan sp ON s.PlanID = sp.PlanID;
```

> *This should return all customer-plan mappings clearly.*

```sql
-- Show payment history with customer info
SELECT
    p.PaymentID,
    c.FirstName + ' ' + c.LastName AS Customer,
    sp.PlanName,
    p.PaymentDate,
    p.Amount,
    p.PaymentStatus
FROM Payment p
JOIN Subscription s ON p.SubscriptionID = s.SubscriptionID
JOIN Customer c ON s.CustomerID = c.CustomerID
JOIN SubscriptionPlan sp ON s.PlanID = sp.PlanID;
```

> *You’ll see how payment records are tied to both plans and customers.*

4.8.5 Summary

In this hands-on lab, you:

- Populated all key tables in a normalized subscription business schema
- Practiced structured `INSERT` commands
- Verified data relationships with `JOIN`-based queries
- Saw how relational integrity provides accurate, multi-table views of business data

✅ This lab prepares you for data retrieval, reporting, and enforcing business rules in future chapters.

4.9 Exercise 10: Apply Normalization to Improve Table Design

4.9.1 Description

This exercise introduces database normalization by starting with a poorly designed table containing redundant and inconsistent data. You will progressively apply the first three normal forms (1NF, 2NF, and 3NF) by decomposing the unnormalized table into well-structured relational tables.

Normalization helps eliminate redundancy, maintain data integrity, and simplify future queries or updates. In this lab, we focus on applying normalization concepts practically in SQL Server 2025.

4.9.2 Objectives

- Identify design problems in an unnormalized table
- Apply 1NF, 2NF, and 3NF to improve data structure
- Implement normalized tables using SQL Server
- Understand the trade-offs of normalization in real-world design

4.9.3 Prerequisites

- SQL Server 2025 Developer Edition
- SQL Server Management Studio (SSMS) 21.x or later
- Basic understanding of `CREATE TABLE`, `INSERT`, and `SELECT`
- Prior completion of Exercise 7 (Basic ERD Design) is helpful but not mandatory

4.9.4 Steps

Here's a step-by-step guide to complete this exercise:

4.9.4.1 Step 1: Create the Database

We'll create a new database for this exercise called `NormalizationDemo`.

```sql
CREATE DATABASE NormalizationDemo;
GO

USE NormalizationDemo;
GO
```

4.9.4.2 Step 2: Start with an Unnormalized Table

The following table stores customer orders but mixes customer, order, and product info in a single table.

```sql
CREATE TABLE Orders_Unnormalized (
    OrderID INT,
    CustomerName NVARCHAR(100),
    CustomerPhone NVARCHAR(20),
    Product1 NVARCHAR(100),
    Product2 NVARCHAR(100),
    Product3 NVARCHAR(100)
);
```

Insert some sample records:

```sql
INSERT INTO Orders_Unnormalized VALUES
(1, 'Nadia', '1234567890', 'Laptop', 'Mouse', 'Keyboard'),
(2, 'Marcel', '0987654321', 'Monitor', NULL, NULL);
```

📝 **Explanation:** This structure violates **1NF** due to repeating groups (`Product1`, `Product2`, `Product3`). It's also hard to manage product details or query specific items.

4.9.4.3 Step 3: Apply First Normal Form (1NF)

We’ll remove repeating groups and move each product to its own row.

```
CREATE TABLE Orders_1NF (
    OrderID INT,
    CustomerName NVARCHAR(100),
    CustomerPhone NVARCHAR(20),
    Product NVARCHAR(100)
);

CREATE TABLE Orders_1NF (
    OrderID INT,
    CustomerName NVARCHAR(100),
    CustomerPhone NVARCHAR(20),
    Product NVARCHAR(100)
);

INSERT INTO Orders_1NF VALUES
(1, 'Zahra', '1234567890', 'Laptop'),
(1, 'Zahra', '1234567890', 'Mouse'),
(1, 'Zahra', '1234567890', 'Keyboard'),
(2, 'Thariq', '0987654321', 'Monitor');
```

📝 **Explanation:** Now each row represents a single product per order. This satisfies **1NF**, but still has redundant customer data.

4.9.4.4 Step 4: Apply Second Normal Form (2NF)

Next, we remove **partial dependencies**—data that depends only on part of a composite key (e.g., `CustomerName` on `OrderID`).

We split the table into separate `Customers` and `Orders` tables.

```
CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY,
    CustomerName NVARCHAR(100),
    CustomerPhone NVARCHAR(20)
);

CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    CustomerID INT,
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);

CREATE TABLE OrderDetails (
    OrderID INT,
    Product NVARCHAR(100),
    PRIMARY KEY (OrderID, Product),
    FOREIGN KEY (OrderID) REFERENCES Orders(OrderID)
);
```

Insert normalized data:

```
INSERT INTO Customers VALUES
(1, 'Zahra', '1234567890'),
(2, 'Thariq', '0987654321');

INSERT INTO Orders VALUES
(1, 1),
(2, 2);

INSERT INTO OrderDetails VALUES
(1, 'Laptop'),
(1, 'Mouse'),
(1, 'Keyboard'),
(2, 'Monitor');
```

📝 **Explanation:** This eliminates the redundant storage of customer information across multiple order rows.

4.9.4.5 Step 5: Apply Third Normal Form (3NF)

Now assume product names have additional properties. We’ll factor products out to a `Products` table and use IDs.

```
CREATE TABLE Products (
    ProductID INT PRIMARY KEY,
    ProductName NVARCHAR(100)
);

CREATE TABLE OrderDetails_3NF (
    OrderID INT,
    ProductID INT,
    PRIMARY KEY (OrderID, ProductID),
    FOREIGN KEY (OrderID) REFERENCES Orders(OrderID),
    FOREIGN KEY (ProductID) REFERENCES Products(ProductID)
);
```

Insert products:

```
INSERT INTO Products VALUES
(1, 'Laptop'),
(2, 'Mouse'),
(3, 'Keyboard'),
(4, 'Monitor');

INSERT INTO OrderDetails_3NF VALUES
(1, 1),
(1, 2),
(1, 3),
(2, 4);
```

📝 **Explanation:** This final form ensures all non-key attributes depend only on the key, the whole key, and nothing but the key—thus achieving **3NF**.

4.9.5 Summary

This exercise showed how to refactor an unnormalized flat table into a clean, normalized relational model through:

Normal Form	Key Improvement
1NF	Removed repeating columns
2NF	Eliminated partial dependencies
3NF	Removed transitive dependencies via lookup

Normalization is key to long-term scalability, consistency, and easier querying in any real-world SQL Server system. You now have a solid foundation to spot and improve poor database designs.

4.10 Conclusion

In this chapter, we covered the basics of relational database design, including tables, data types, keys, and normalization principles. We also explored how to implement these concepts in SQL Server 2025 through practical exercises. Understanding these foundational elements is crucial for building efficient and maintainable databases that meet business needs.

5 Views and Logical Data Modeling

This chapter introduces the concept of **views** in SQL Server 2025, a powerful tool for abstracting complex data relationships. Views function as **virtual tables**, allowing you to create simplified, reusable representations of data. We also explore how views support **role-based access control** and **logical data modeling**, critical for secure and scalable enterprise reporting.

5.1 What Is a View?

A **view** is a saved SQL query that presents data as a **virtual table**. It does not store data physically, but provides a consistent, queryable layer over one or more base tables.

Views can encapsulate complex joins, aggregations, and business logic, making it easier for users to access and analyze data without needing to understand the underlying table structures.

Views offer several advantages:

- Abstracts complex joins or business rules
- Simplifies queries for end users
- Enables consistent definitions of business metrics
- Supports security and compliance through access control

5.2 Creating Views in SQL Server 2025

To create a view, you use the `CREATE VIEW` statement followed by the view name and the `SELECT` query that defines its structure. Here’s a basic example:

```
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

For instance, to create a view that shows customer orders with relevant details:

```
CREATE VIEW Sales.vw_CustomerOrders AS
SELECT
    c.CustomerID,
    c.FirstName,
    c.LastName,
    o.OrderID,
    o.OrderDate,
    o.TotalAmount
FROM Sales.Customers c
JOIN Sales.Orders o ON c.CustomerID = o.CustomerID;
```

This view combines customer and order data, allowing users to query customer orders without needing to write complex joins each time.

Users can now query:

```
SELECT * FROM Sales.vw_CustomerOrders
WHERE OrderDate >= '2025-01-01';
```

5.3 Views as Virtual Tables

Views behave much like physical tables in that you can query them using standard SQL statements. When you select from a view, SQL Server dynamically generates the result set based on the underlying query definition.

You can join views with other views or tables, enabling you to build complex queries on top of simplified, reusable data structures. However, it’s important to remember that views do not store data themselves; instead, they produce results at runtime each time they are queried.

You can join views just like you would with tables. For example, if you have a view for customer orders and another for order details, you can combine them:

```
SELECT *
FROM Sales.vw_CustomerOrders co
JOIN Sales.vw_OrderDetails od ON co.OrderID = od.OrderID;
```

5.4 Updatable Views

Not all views are updatable, but many **simple views** allow updates.

An updatable view allows you to modify data through the view as if it were a table. For example, if you have a view that shows customer names and emails, you can update the email directly through the view:

```
CREATE VIEW HR.vw_Employees AS
SELECT EmployeeID, FirstName, LastName
FROM HR.Employees;
```

You can then update the view like this:

```
UPDATE HR.vw_Employees
SET LastName = 'Anderson'
WHERE EmployeeID = 101;
```

Certain conditions make a view non-updatable. For example, if a view includes:

- Contains aggregate functions such as `SUM` or `AVG`
- Uses set operations like `DISTINCT`, `GROUP BY`, or `UNION`
- Includes joins across multiple tables (in most cases)

5.5 Role-Based Schema Simplification Using Views

Views can be tailored to different user roles, providing a simplified schema that exposes only the necessary data. This is particularly useful in environments with diverse user groups, such as finance, HR, and sales teams.

For example, you might have a complex sales database with sensitive financial data. Instead of exposing the entire schema, you can create views that present only the relevant information for each department:

- **Finance users** see financial metrics.
- **HR users** see employee data.

- **Sales team** sees customer and order history.

Each group can be given access to tailored views:

For HR users, you might create a view that shows employee names and departments without sensitive salary information:

```
CREATE VIEW Secure.vw_EmployeeDirectory AS
SELECT FirstName, LastName, Department, HireDate
FROM HR.Employees;
```

This view allows HR personnel to access employee information without exposing sensitive data like salaries or personal identifiers.

For the sales team, you might create a view that aggregates customer orders by region:

```
CREATE VIEW Secure.vw_SalesSummary AS
SELECT CustomerID, Region, SUM(TotalAmount) AS TotalSales
FROM Sales.Orders
GROUP BY CustomerID, Region;
```

This view provides a high-level summary of sales data, allowing sales representatives to focus on their performance metrics without needing to understand the underlying table structures.

5.6 Security and Compliance with Views

Views can enhance security by restricting access to sensitive data. By granting users permissions on views instead of base tables, you can control what data they can see and modify.

Here are some key benefits of using views for security:

- Views can **restrict access** to sensitive columns.
- Views support **schema-level abstraction** for regulatory compliance (e.g., GDPR, HIPAA).

You can create a view that masks sensitive information, such as email addresses, while still allowing users to see other relevant data:

```sql
CREATE VIEW Compliance.vw_CustomerPublic AS
SELECT
    CustomerID,
    FirstName,
    LEFT(Email, CHARINDEX('@', Email)) + '***.com' AS MaskedEmail
FROM Sales.Customers;
```

This view allows users to see customer names while masking their email addresses, ensuring compliance with data protection regulations.

5.7 Indexed Views (Materialized Views)

Indexed views, also known as materialized views, are a powerful feature in SQL Server that allows you to store the results of a view physically. This can significantly improve query performance, especially for complex aggregations or joins.

An **indexed view** is physically stored with a **clustered index**—like a materialized view.

Indexed views are particularly useful for:

- Aggregate data for reports
- Improve performance on frequent heavy queries

To create an indexed view, you first define the view and then create a clustered index on it. Here's an example of creating an indexed view that summarizes sales by region:

```sql
CREATE VIEW Sales.vw_TotalRevenueByRegion
WITH SCHEMABINDING AS
SELECT Region, SUM(TotalAmount) AS Revenue
FROM dbo.Orders
GROUP BY Region;

CREATE UNIQUE CLUSTERED INDEX idx_TotalRevenue
ON Sales.vw_TotalRevenueByRegion(Region);
```

This indexed view allows you to quickly retrieve total revenue by region without recalculating the aggregation each time the view is queried. It is important to note that indexed views have specific requirements and limitations, such as:

*Note: `WITH SCHEMABINDING` is **mandatory** for indexed views.*

5.8 Exercise 11: Create Reusable Views for Sales Analysis

5.8.1 Description

In this exercise, you'll create SQL views to simplify recurring sales queries. Views provide an abstraction over complex `JOIN` and `WHERE` logic, making analysis more accessible to business users and report developers. We'll simulate a basic sales reporting layer by joining customer, product, and sales information into reusable views. These views reflect how logical data modeling is implemented in SQL Server.

5.8.2 Objectives

- Understand the purpose and benefits of views for logical modeling
- Learn to build `CREATE VIEW` statements with `JOIN`, `WHERE`, and calculated columns
- Simplify multi-table reporting using views
- Query from views as if they were tables

5.8.3 Prerequisites

- SQL Server 2025 Developer Edition installed and running
- SQL Server Management Studio (SSMS) 21.x installed
- Basic understanding of `SELECT`, `JOIN`, and filtering in SQL
- Internet access to download sample data (optional)

⚠️ *Note: We will create a fresh database named `SalesAnalysisDB` for this exercise.*

5.8.4 Steps

Here's a step-by-step guide to complete this exercise:

5.8.4.1 Step 1: Create a New Database

We begin by creating an isolated database to work with views without modifying existing systems.

```sql
CREATE DATABASE SalesAnalysisDB;
GO

USE SalesAnalysisDB;
GO
```

> *This will be the workspace for our view-based modeling tasks.*

5.8.4.2 Step 2: Create Sample Tables

What we're doing: Define simple `Customer`, `Product`, and `SalesOrder` tables to simulate transactional data.

```sql
CREATE TABLE Customer (
    CustomerID INT PRIMARY KEY IDENTITY,
    FullName NVARCHAR(100),
    Region NVARCHAR(50)
);

CREATE TABLE Product (
    ProductID INT PRIMARY KEY IDENTITY,
    ProductName NVARCHAR(100),
    Category NVARCHAR(50),
    Price DECIMAL(10,2)
);

CREATE TABLE SalesOrder (
    SalesOrderID INT PRIMARY KEY IDENTITY,
    CustomerID INT,
    ProductID INT,
    OrderDate DATE,
    Quantity INT,
    FOREIGN KEY (CustomerID) REFERENCES Customer(CustomerID),
    FOREIGN KEY (ProductID) REFERENCES Product(ProductID)
);
```

> *These tables reflect a simplified model suitable for building analytical views.*

5.8.4.3 Step 3: Insert Sample Data

What we're doing: Add representative data to simulate real sales records.

```sql
-- Insert Customers
INSERT INTO Customer (FullName, Region)
VALUES ('Minji Kim', 'North'), ('Hiroshi Sato', 'South'), ('Priya Singh', 'East');

-- Insert Products
INSERT INTO Product (ProductName, Category, Price)
VALUES
('Subscription A', 'Service', 15.00),
('Subscription B', 'Service', 25.00),
('Consulting Package', 'Consulting', 100.00);

-- Insert Sales Orders
INSERT INTO SalesOrder (CustomerID, ProductID, OrderDate, Quantity)
VALUES
(1, 1, '2025-07-01', 2),
(2, 2, '2025-07-02', 1),
(3, 3, '2025-07-03', 3);
```

> *This dataset is enough to support basic aggregation and filtering through views.*

5.8.4.4 Step 4: Create a View for Sales Summary

Build a view to show aggregated sales metrics like total sales amount and quantity per order.

```sql
CREATE VIEW vw_SalesSummary AS
SELECT
    so.SalesOrderID,
    c.FullName AS CustomerName,
    c.Region,
    p.ProductName,
    p.Category,
    so.Quantity,
    p.Price,
    (so.Quantity * p.Price) AS TotalAmount,
    so.OrderDate
FROM SalesOrder so
JOIN Customer c ON so.CustomerID = c.CustomerID
JOIN Product p ON so.ProductID = p.ProductID;
```

This reusable view simplifies multi-table joins and includes a calculated column for total order amount.

5.8.4.5 Step 5: Query the View

Query the view just like a regular table to see how it simplifies access to sales data.

```
SELECT * FROM vw_SalesSummary
WHERE Region = 'North'
ORDER BY OrderDate DESC;
```

This returns all sales for customers in the North region, sorted by the most recent orders.

5.8.4.6 Step 6: Create Another View: Total Sales per Region

Create a view that groups and aggregates sales by region.

```
CREATE VIEW vw_TotalSalesByRegion AS
SELECT
    c.Region,
    SUM(p.Price * so.Quantity) AS TotalSales,
    COUNT(DISTINCT so.SalesOrderID) AS OrdersCount
FROM SalesOrder so
JOIN Customer c ON so.CustomerID = c.CustomerID
JOIN Product p ON so.ProductID = p.ProductID
GROUP BY c.Region;
```

This view is useful for generating dashboards that show high-level business metrics per region.

5.8.4.7 Step 7: Query the Aggregated View

Run a query to show summarized metrics across regions.

```
SELECT * FROM vw_TotalSalesByRegion
ORDER BY TotalSales DESC;
```

> *You now have a reusable, analytical view for regional sales comparison.*

5.8.5 Summary

In this hands-on lab, you:

- Created sample transactional tables simulating a business environment
- Populated them with initial data to mimic real-world sales
- Defined views that abstract away join logic and calculation complexity
- Demonstrated how views can model reusable logical representations for analysts and BI tools

✅ Views make querying easier, cleaner, and more consistent—especially in collaborative or BI-focused environments.

5.9 Exercise 12: Simplify Complex Joins via Views

5.9.1 Description

In many real-world database environments, data is often stored across multiple related tables—especially in normalized systems. Writing queries that involve multiple joins can become tedious, especially for business analysts or application developers who are only interested in specific high-level information. In this lab, you'll learn how to create a view that joins multiple tables (`SalesOrder`, `Customer`, `Product`) into a single logical object. This view will help simplify downstream queries by encapsulating complex relationships behind a reusable layer.

5.9.2 Objectives

- Understand how views help abstract join logic
- Create a SQL view that joins multiple tables
- Use calculated columns within views
- Query the view as if it were a single flat table

5.9.3 Prerequisites

- SQL Server 2025 Developer Edition running
- SQL Server Management Studio (SSMS) 21.x installed
- Exercise 11 completed (database `SalesAnalysisDB` with related tables and sample data)

5.9.4 Steps

Here's a step-by-step guide to complete this exercise:

5.9.4.1 Step 1: Review Existing Schema

What we're doing: Before we build the view, it's helpful to recall the structure of the tables we'll join.

- `SalesOrder` contains references to `CustomerID` and `ProductID`
- `Customer` includes `FullName` and `Region`
- `Product` includes `ProductName`, `Category`, and `Price`

These tables are all linked via foreign keys, and we'll leverage those relationships in our view.

5.9.4.2 Step 2: Create a New View with Multiple Joins

What we're doing: We'll write a view named `vw_SalesDetails` that joins the three tables and adds a computed column.

```
USE SalesAnalysisDB;
GO
```

```
CREATE VIEW vw_SalesDetails AS
SELECT
    so.SalesOrderID,
    so.OrderDate,
    c.CustomerID,
    c.FullName AS CustomerName,
    c.Region,
    p.ProductID,
    p.ProductName,
    p.Category,
    p.Price,
    so.Quantity,
    (so.Quantity * p.Price) AS TotalAmount
FROM SalesOrder so
JOIN Customer c ON so.CustomerID = c.CustomerID
JOIN Product p ON so.ProductID = p.ProductID;
```

This view combines customer, product, and sales order data into a single logical structure. It allows users to see all relevant information in one place without needing to write complex joins each time.

This view flattens out the data structure and includes a calculated field for `TotalAmount`.

5.9.4.3 Step 3: Query the View

Now that the view is created, we can query it like a regular table to get a full picture of sales.

```
SELECT * FROM vw_SalesDetails
ORDER BY OrderDate DESC;
```

You’ll see customer, product, and transaction data joined and formatted in a single result set.

5.9.4.4 Step 4: Filter and Project Specific Columns

We can use standard filtering and projection to refine our results without rewriting the joins.

```
SELECT
    CustomerName,
    Region,
    ProductName,
    Quantity,
    TotalAmount
FROM vw_SalesDetails
WHERE Region = 'South' AND Category = 'Service';
```

This shows how views can support business-level queries with minimal SQL complexity for the user.

5.9.4.5 Step 5: View Query Plan (Optional)

To understand performance, you can check the execution plan in SSMS by clicking “Include Actual Execution Plan” before running a query on the view.

> *This helps confirm that the view is executed dynamically and doesn’t materialize data unless indexed.*

5.9.5 Summary

In this lab, you:

- Built a view that joins multiple normalized tables
- Included calculated fields to provide richer data output
- Simplified how users access and analyze relational data
- Demonstrated how views improve productivity and consistency

5.10 Conclusion

In this chapter, we explored the concept of views in SQL Server 2025, including their creation, usage, and benefits. We learned how views can simplify complex queries, enhance security through role-based access control, and support logical data modeling. Additionally, we discussed indexed views for performance optimization.

6 Designing Multi-Tenant and SaaS Databases

This chapter explores how to design SQL Server 2025 databases to support **multi-tenant** Software-as-a-Service (SaaS) applications. You'll learn architectural patterns—**shared database**, **schema-per-tenant**, and how to enforce **tenant isolation** using identity filtering. These practices help deliver secure, scalable services across many customers (tenants) while maintaining performance and compliance.

6.1 What Is a Multi-Tenant Database?

A **multi-tenant database** serves multiple customers (tenants) using shared infrastructure, with varying levels of data isolation.

Here are the primary goals of a multi-tenant database design:

- **Scalability**: Support many tenants without duplicating infrastructure.
- **Security**: Isolate data between tenants.
- **Cost-efficiency**: Share computing and storage resources.

6.2 Multi-Tenant Patterns in SQL Server

SQL Server 2025 supports several architectural patterns for multi-tenant applications. The choice of pattern depends on your specific requirements for data isolation, scalability, and management complexity.

6.2.1 Pattern 1: Shared Database, Shared Schema

Well-known in SaaS, this pattern allows multiple tenants to share the same database and tables. Each table includes a `TenantID` column to identify which tenant owns the data.

With this approach: * All tenants share **the same database and tables**. * A `TenantID` column is added to every table to identify which tenant owns which data.

For instance, an `Orders` table might look like this:

```
CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    TenantID INT NOT NULL,
    CustomerID INT,
    OrderDate DATE,
    TotalAmount DECIMAL(10,2)
);
```

This allows you to filter queries by `TenantID` to ensure each tenant only sees their own data:

```
SELECT * FROM Orders WHERE TenantID = @TenantID;
```

Well-suited for SaaS applications with many small tenants:

- Low cost and resource usage.
- Easy to scale and manage.
- Simplifies updates and deployments.

Weigh the benefits against challenges:

- Risk of data leakage if queries are not properly filtered.
- Performance issues with large datasets.
- Complex queries may require additional logic to ensure tenant isolation.
- Harder to enforce row-level security.

6.2.2 Pattern 2: Shared Database, Schema-Per-Tenant

In this pattern, each tenant has its own schema within a shared database. This provides better data isolation while still allowing shared infrastructure.

> *One **shared database**, but each tenant has their own **schema**.*

This approach allows you to create separate schemas for each tenant, which can help with data isolation and management. For example, you might have schemas like `tenant_101`, `tenant_102`, etc.

```
CREATE SCHEMA tenant_101;
CREATE TABLE tenant_101.Orders (...);
```

With schema-per-tenant, you get:

- Better data isolation compared to shared schema.
- Easier to manage tenant-specific changes.
- Allows for tenant-specific optimizations (e.g., indexing).
- Simplifies compliance with data protection regulations.
- Easier to enforce row-level security.

We also face challenges:

- More complex management as the number of schemas grows.
- Increased metadata overhead.

6.2.3 Pattern 3: Database-Per-Tenant

In this pattern, each tenant has its own dedicated database. This provides the highest level of data isolation and security.

> *Each tenant has a **separate database**.*

Advantages of this approach include:

- Complete data isolation.
- Simplified compliance with data protection regulations.
- Easier to manage tenant-specific performance tuning.
- Simplifies backup and restore operations for individual tenants.
- Allows for different database configurations per tenant (e.g., performance, storage).

We also face challenges:

- Higher resource usage due to multiple databases.
- Increased management overhead for backups, updates, and monitoring.
- More complex deployment and scaling strategies.
- Difficult to apply updates globally.

> *Note: SQL Server 2025 supports elastic pools and automation, but this pattern is best for large tenants.*

6.3 Tenant Isolation and Identity Filtering

In multi-tenant applications, **tenant isolation** is crucial to prevent data leakage between tenants. SQL Server provides several mechanisms to enforce this isolation, including:

- **Identity filtering**: Ensuring queries only return data for the current tenant.
- **Row-Level Security (RLS)**: Automatically filtering rows based on the tenant context.
- **Views**: Creating tenant-specific views to simplify access control.
- **Stored procedures**: Encapsulating tenant logic to enforce isolation.

6.3.1 Option 1: Manual Filtering by Tenant ID

The simplest way to enforce tenant isolation is to include a `TenantID` column in every table and filter queries by this ID. This approach requires you to manually add the `TenantID` filter in every query.

> *Include `TenantID` in every table and apply it to every query.*

For example, to retrieve orders for a specific tenant:

```sql
SELECT OrderID, CustomerID, TotalAmount
FROM Orders
WHERE TenantID = @TenantID;
```

This method is straightforward but requires discipline to ensure every query includes the `TenantID` filter. It can lead to potential data leaks if queries are not properly constructed.

> ***Best Practice**: Always filter by `TenantID` in app layer and/or views.*

6.3.2 Option 2: Use Row-Level Security (RLS)

Row-Level Security (RLS) allows you to define security policies that automatically filter rows based on the user or session context. This provides a robust way to enforce tenant isolation without requiring manual filtering in every query.

> *SQL Server's **Row-Level Security** automatically filters rows based on the user or session context.*

Let's create a simple RLS policy to enforce tenant isolation:

```
CREATE FUNCTION Security.fn_TenantFilter(@TenantID INT)
RETURNS TABLE
WITH SCHEMABINDING
AS
RETURN SELECT 1 AS Result
WHERE @TenantID = CAST(SESSION_CONTEXT(N'TenantID') AS INT);
```

This function checks if the `TenantID` in the session context matches the `TenantID` in the row. Next, we create a security policy that uses this function:

```
CREATE SECURITY POLICY Security.TenantSecurityPolicy
ADD FILTER PREDICATE Security.fn_TenantFilter(TenantID)
ON dbo.Orders
WITH (STATE = ON);
```

Now, when a user queries the `Orders` table, SQL Server automatically filters rows based on the `TenantID` set in the session context.

We can set the `TenantID` in the session context at the start of each user session:

```
EXEC sp_set_session_context @key = N'TenantID', @value = 101;
```

This ensures that all subsequent queries in that session automatically filter by the tenant ID, providing a secure and efficient way to enforce tenant isolation.

All queries automatically return data only for `TenantID = 101`.

6.4 Managing Identity and Shared Metadata

In shared-schema systems, you often need **shared metadata tables** (e.g., product catalogs) while keeping tenant data isolated.

For example, you might have a `Products` table that is shared across all tenants, while the `Orders` table is tenant-specific. Here's how you can structure your queries to handle this:

- `Products`: Shared
- `Orders`: Tenant-isolated

To retrieve product information for a specific tenant's orders, you can join the `Products` table with the `Orders` table while filtering by `TenantID`:

```
SELECT p.ProductName, o.TotalAmount
FROM Products p
JOIN Orders o ON p.ProductID = o.ProductID
WHERE o.TenantID = @TenantID;
```

This allows you to access shared metadata while ensuring tenant isolation for transactional data.

6.5 Best Practices for Multi-Tenant SQL Server Design

Here are some best practices to follow when designing multi-tenant databases in SQL Server:

Practice	Benefit
Add `TenantID` to every tenant-specific table	Data isolation and filtering
Enforce `TenantID` via views or RLS	Prevents accidental data leaks
Use views for tenant-specific access	Simplifies reporting and filtering
Avoid cross-tenant joins	Keeps boundaries clean
Encrypt tenant data at rest and in transit	Compliance and security

6.6 Exercise 13: Add Tenant Column and Apply Security Filters

6.6.1 Description

In Software-as-a-Service (SaaS) environments, it's common to host data for multiple customers (tenants) in a single database. A popular approach is the **shared-database, shared-schema** model, where tables include a `TenantID` column to isolate data between tenants. In this lab, you'll design a simple multi-tenant schema with a `TenantID` column, and then implement security filtering using **views** to restrict tenant access. This approach provides both simplicity and scalability for multi-tenant applications.

6.6.2 Objectives

- Create a database and schema that supports multiple tenants
- Add a `TenantID` column to relevant tables
- Insert data for multiple tenants
- Create filtered views to enforce row-level security logic

6.6.3 Prerequisites

- SQL Server 2025 Developer Edition installed
- SQL Server Management Studio (SSMS) 21.x
- Basic knowledge of `CREATE TABLE`, `INSERT`, and `VIEW` syntax
- A new database will be created for this lab

6.6.4 Steps

Here's a step-by-step guide to complete this exercise:

6.6.4.1 Step 1: Create a New Database

We'll begin by creating a new database for this exercise named `SaaSAppDB`.

```
CREATE DATABASE SaaSAppDB;
GO

USE SaaSAppDB;
GO
```

6.6.4.2 Step 2: Create Tables with `TenantID`

We define two main tables: `Tenant` to register each tenant, and `Customer` to store customer data tied to a `TenantID`. This is the core of our multi-tenant schema.

```
-- Tenant registration table
CREATE TABLE Tenant (
    TenantID INT PRIMARY KEY IDENTITY(1,1),
    TenantName NVARCHAR(100) NOT NULL
);

-- Shared Customer table
CREATE TABLE Customer (
    CustomerID INT PRIMARY KEY IDENTITY(1000,1),
    TenantID INT NOT NULL,
    FullName NVARCHAR(100),
    Email NVARCHAR(100),
    CreatedAt DATETIME2 DEFAULT SYSDATETIME(),
    FOREIGN KEY (TenantID) REFERENCES Tenant(TenantID)
);
```

The `Customer` table includes `TenantID` as a foreign key, which ensures each customer is scoped to a tenant.

6.6.4.3 Step 3: Insert Sample Data for Multiple Tenants

Now, we insert demo tenants and customers to simulate a real SaaS setup.

```
-- Insert two tenants
INSERT INTO Tenant (TenantName) VALUES ('Ilmu Data.'), ('Neuville Ltd.');

-- Insert customers for each tenant
INSERT INTO Customer (TenantID, FullName, Email)
VALUES
(1, 'Amina Okoro', 'amina@ilmudata.id'),
(1, 'Jeroen van Dijk', 'jeroen@ilmudata.id'),
(1, 'Élodie Martin', 'elodie@ilmudata.id'),
(1, 'Niran Chaiyawat', 'niran@ilmudata.id'),
(1, 'Lucas Silva', 'lucas@ilmudata.id'),
(2, 'Kwame Mensah', 'kwame@neuville.id'),
(2, 'Sanne de Vries', 'sanne@neuville.id'),
(2, 'Julien Dubois', 'julien@neuville.id'),
(2, 'Anong Srisuk', 'anong@neuville.id'),
(2, 'Bruna Costa', 'bruna@neuville.id');
```

This creates two distinct sets of customers, one for each tenant. Each customer is associated with a specific `TenantID`, ensuring data isolation.

6.6.4.4 Step 4: Create a View to Filter by TenantID

Instead of querying `Customer` directly, we'll create a **filtered view** for a specific tenant. This ensures tenant isolation.

```
-- View for Ilmu Data (TenantID = 1)
CREATE VIEW vw_IlmuData_Customers AS
SELECT * FROM Customer WHERE TenantID = 1;
```

> *Later, a stored procedure or app logic can switch tenant context dynamically.*

6.6.4.5 Step 5: Use the View to Query Tenant-Specific Data

Query the view to get only Ilmu Data's customers—this simulates how the application would restrict data visibility.

```
SELECT FullName, Email, CreatedAt
FROM vw_IlmuData_Customers;
```

> *Output should only include rows where `TenantID = 1`.*

6.6.4.6 Step 6 (Optional): Create a Parameterized Filtering Procedure

If you want a reusable query for multiple tenants, you can use a stored procedure with `TenantID` as a parameter:

```
CREATE PROCEDURE GetCustomersByTenant
    @TenantID INT
AS
BEGIN
    SELECT FullName, Email, CreatedAt
    FROM Customer
    WHERE TenantID = @TenantID;
END;
GO

-- Usage
EXEC GetCustomersByTenant @TenantID = 2;
```

> *This is a flexible way to support multiple tenants securely from backend logic.*

6.6.5 Summary

In this hands-on lab, you:

- Designed a shared-schema, multi-tenant database model
- Added a `TenantID` column to enforce row-level separation
- Created filtered views to isolate tenant data
- Explored how to use stored procedures for secure tenant access

✅ This pattern is foundational for SaaS applications running on a single database while maintaining logical data separation per tenant.

6.7 Exercise 14: Build Views and Indexes per Tenant

6.7.1 Description

In multi-tenant database designs, isolating tenant data using views is common for security and logical separation. However, performance may degrade as data grows. A common strategy is to **create filtered indexes** on tenant-specific data, paired with **views per tenant** or **parameterized views**. This lab demonstrates how to build efficient views and **tenant-filtered indexes** to enhance query performance in SQL Server 2025.

6.7.2 Objectives

- Create a schema with tenant data
- Define views per tenant for logical access control
- Build filtered indexes to improve performance
- Evaluate execution plans with and without indexes

6.7.3 Prerequisites

- SQL Server 2025 Developer Edition
- SQL Server Management Studio (SSMS) 21.x
- Basic familiarity with `CREATE VIEW`, `CREATE INDEX`, and `SELECT` queries
- No prior index tuning experience required

6.7.4 Steps

Here’s a step-by-step guide to complete this exercise:

6.7.4.1 Step 1: Create the Multi-Tenant Database

We’ll start by creating a new database `SaaSPerfDB`.

```sql
CREATE DATABASE SaaSPerfDB;
GO

USE SaaSPerfDB;
GO
```

6.7.4.2 Step 2: Define Tables with TenantID

We define two core tables: `Tenant` and `Invoice`. Each invoice belongs to a tenant using the `TenantID` field.

```sql
CREATE TABLE Tenant (
    TenantID INT PRIMARY KEY IDENTITY(1,1),
    TenantName NVARCHAR(100) NOT NULL
);

CREATE TABLE Invoice (
    InvoiceID INT PRIMARY KEY IDENTITY(1000,1),
    TenantID INT NOT NULL,
    InvoiceDate DATE,
    CustomerName NVARCHAR(100),
    Amount DECIMAL(10,2),
    Status NVARCHAR(50),
    FOREIGN KEY (TenantID) REFERENCES Tenant(TenantID)
);
```

6.7.4.3 Step 3: Insert Sample Data for Multiple Tenants

We insert two tenants with several invoices each, simulating a real-world workload.

```
-- Add tenants
INSERT INTO Tenant (TenantName) VALUES ('RetailCorp'), ('HealthPlus');

-- Add invoices
INSERT INTO Invoice (TenantID, InvoiceDate, CustomerName, Amount, Status)
VALUES
(1, '2025-01-15', 'Pram Jatmiko', 1200, 'Paid'),
(1, '2025-02-10', 'Olga Ivanova', 1500, 'Pending'),
(1, '2025-03-05', 'John Miller', 1800, 'Paid'),
(2, '2025-01-12', 'Siti Aisyah', 980, 'Paid'),
(2, '2025-03-10', 'Ivan Petrov', 2150, 'Pending');
```

This creates two tenants with invoices, each having a unique `TenantID`.

> *Each tenant now has its own subset of invoices.*

6.7.4.4 Step 4: Create Tenant-Specific Views

For security and simplicity, we’ll define views to isolate each tenant’s data.

```
-- View for RetailCorp (TenantID = 1)
CREATE VIEW vw_RetailCorp_Invoices AS
SELECT * FROM Invoice WHERE TenantID = 1;
GO

-- View for HealthPlus (TenantID = 2)
CREATE VIEW vw_HealthPlus_Invoices AS
SELECT * FROM Invoice WHERE TenantID = 2;
GO
```

> *These views simulate per-tenant data access in a SaaS application.*

6.7.4.5 Step 5: Query Views (Without Indexes)

Now, test the views to simulate how a tenant would access their data:

```
SELECT * FROM vw_RetailCorp_Invoices;
```

Open the **execution plan** in SSMS to see that it scans the entire table.

> *Even though you're filtering by `TenantID`, without an index, SQL Server performs a table scan.*

6.7.4.6 Step 6: Add Filtered Indexes for Each Tenant

To optimize performance, we create **filtered indexes** that include only rows per tenant:

```
-- Index for RetailCorp
CREATE NONCLUSTERED INDEX IX_Invoice_Tenant1
ON Invoice (InvoiceDate)
WHERE TenantID = 1;

-- Index for HealthPlus
CREATE NONCLUSTERED INDEX IX_Invoice_Tenant2
ON Invoice (InvoiceDate)
WHERE TenantID = 2;
```

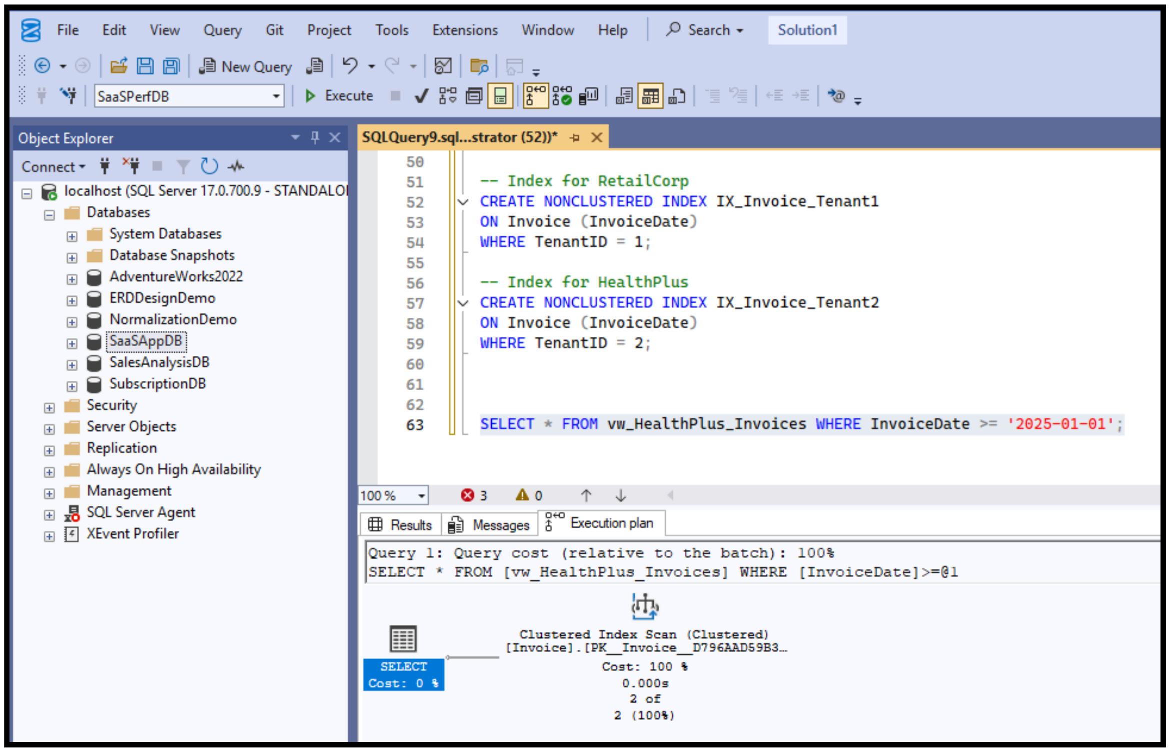
These indexes reduce the I/O cost when querying tenant views or filtering by `TenantID`.

6.7.4.7 Step 7: Re-Run Queries and Review Execution Plan

Re-run the earlier view queries and inspect the execution plan again:

```
SELECT * FROM vw_HealthPlus_Invoices WHERE InvoiceDate >= '2025-01-01';
```

You should now observe an **index seek** instead of a scan, showing improved performance.



```sql
-- Index for RetailCorp
CREATE NONCLUSTERED INDEX IX_Invoice_Tenant1
ON Invoice (InvoiceDate)
WHERE TenantID = 1;

-- Index for HealthPlus
CREATE NONCLUSTERED INDEX IX_Invoice_Tenant2
ON Invoice (InvoiceDate)
WHERE TenantID = 2;



SELECT * FROM vw_HealthPlus_Invoices WHERE InvoiceDate >= '2025-01-01';
```

```
Query 1: Query cost (relative to the batch): 100%
SELECT * FROM [vw_HealthPlus_Invoices] WHERE [InvoiceDate]>=@1
```

Figure 6.1: Observing view queries performance.

6.7.5 Summary

In this lab, you learned how to:

- Implement tenant views to isolate data
- Use filtered indexes for performance gains
- Optimize access to large multi-tenant tables
- Analyze query performance using SSMS tools

This technique is vital in SaaS systems where maintaining both **data security** and **query responsiveness** is crucial.

6.8 Conclusion

In this chapter, we explored how to design multi-tenant databases in SQL Server 2025, focusing on architectural patterns like shared database, schema-per-tenant, and database-per-tenant. We also discussed tenant isolation techniques such as identity filtering and row-level security. By following best practices and leveraging SQL Server features, you can build secure, scalable multi-tenant applications that meet diverse customer needs.

Section 4: Aggregation, Data Combination and Analytical Query Techniques

7 Grouping, Aggregation, and PIVOTs

This chapter introduces how to summarize and transform large datasets using **GROUP BY**, **aggregate functions**, **HAVING filters**, and **PIVOT queries** in SQL Server 2025. These techniques are essential for reporting, analytics, and deriving business intelligence directly from structured data.

7.1 What Is Aggregation?

Aggregation refers to the process of calculating summary values (e.g., totals, averages, counts) across groups of rows. It allows you to condense large datasets into meaningful insights, making it easier to analyze trends and patterns.

7.2 GROUP BY: Summarizing Rows by Category

When you want to summarize data, you use the `GROUP BY` clause to group rows that share a common value in one or more columns. This is often combined with aggregate functions like `SUM`, `AVG`, `COUNT`, etc.

Here's the basic syntax:

```
SELECT column1, AGG_FUNCTION(column2)
FROM table
GROUP BY column1;
```

Following are some commonly used aggregate functions in SQL Server:

Function	Description
`COUNT()`	Number of rows
`SUM()`	Total value

Function	Description
`AVG()`	Average value
`MIN()`	Smallest value
`MAX()`	Largest value

Now, let’s look at some examples of using `GROUP BY` with aggregate functions.

We can calculate total sales for each region using the `SUM` function:

```sql
SELECT Region, SUM(TotalAmount) AS TotalSales
FROM Sales.Orders
GROUP BY Region;
```

This query groups the `Orders` table by `Region` and calculates the total sales amount for each region.

We can also calculate the average order value per customer:

```sql
SELECT CustomerID, AVG(TotalAmount) AS AvgOrderValue
FROM Sales.Orders
GROUP BY CustomerID;
```

This query groups the `Orders` table by `CustomerID` and calculates the average order value for each customer.

7.3 HAVING: Filtering Groups

Sometimes, you want to filter the results of a `GROUP BY` query based on the aggregated values. This is where the `HAVING` clause comes in. It allows you to specify conditions on aggregate functions, similar to how `WHERE` filters individual rows.

> *`HAVING` filters **grouped results**, similar to how `WHERE` filters rows.*

Here’s the syntax for using `HAVING`:

```
SELECT column1, AGG_FUNCTION(column2)
FROM table
GROUP BY column1
HAVING AGG_FUNCTION(column2) condition;
```

For example, to find regions with total sales greater than 100,000:

```
SELECT Region, SUM(TotalAmount) AS TotalSales
FROM Sales.Orders
GROUP BY Region
HAVING SUM(TotalAmount) > 100000;
```

> ***Tip**: Use `HAVING` **only with aggregate functions**.*

7.4 Multiple Columns in GROUP BY

You can group by multiple columns to create more detailed summaries. For example, to get total sales by both region and product category:

```
SELECT
    Region,
    YEAR(OrderDate) AS OrderYear,
    SUM(TotalAmount) AS TotalSales
FROM Sales.Orders
GROUP BY Region, YEAR(OrderDate);
```

This query groups the `Orders` table by both `Region` and the year of the `OrderDate`, providing a breakdown of sales by region and year.

7.5 PIVOT: Rotating Data for Reports

The `PIVOT` operator allows you to transform row values into columns, making it ideal for creating cross-tab reports. This is particularly useful for summarizing data in a more readable format.

Here's the basic syntax for a `PIVOT` query:

```
SELECT *
FROM (
    SELECT column_to_group, column_to_pivot, value_column
    FROM your_table
) AS SourceTable
```

```
PIVOT (
    AGG_FUNCTION(value_column)
    FOR column_to_pivot IN ([col1], [col2], [col3])
) AS PivotTable;
```

For example, to create a report showing monthly sales totals by region, you can use the following `PIVOT` query:

```
SELECT *
FROM (
    SELECT
        Region,
        FORMAT(OrderDate, 'MMM') AS OrderMonth,
        TotalAmount
    FROM Sales.Orders
) AS RawData
PIVOT (
    SUM(TotalAmount)
    FOR OrderMonth IN ([Jan], [Feb], [Mar], [Apr], [May], [Jun])
) AS MonthlySales;
```

This query transforms the monthly sales data into a format where each month becomes a column, allowing for easy comparison of sales across regions.

Here’s another example that shows how to pivot order details by status:

```
SELECT *
FROM (
    SELECT ProductName, Status, Quantity
    FROM Sales.OrderDetails
) AS SourceData
PIVOT (
    SUM(Quantity)
    FOR Status IN ([Pending], [Shipped], [Cancelled])
) AS PivotResult;
```

This query summarizes the quantity of products ordered by their status, creating a clear view of how many items are pending, shipped, or cancelled.

7.6 Unpivoting (Optional Advanced)

`UNPIVOT` rotates columns back into rows—useful when data is stored in a wide format but you need a tall structure.

Here’s the syntax for `UNPIVOT`:

```sql
SELECT CustomerID, Metric, Value
FROM SalesMetrics
UNPIVOT (
    Value FOR Metric IN (TotalSales, TotalOrders, TotalReturns)
) AS Unpivoted;
```

7.7 Exercise 15: Generate Monthly Revenue Summaries

7.7.1 Description

In this exercise, you’ll write SQL queries to generate **monthly revenue summaries** from the `AdventureWorks2022` database. You’ll group sales data by **month and year**, calculate **total revenue**, and prepare it for potential reporting or dashboarding use. This is a foundational exercise in analyzing business trends over time using SQL Server 2025.

7.7.2 Objectives

By the end of this exercise, you will be able to:

- Use `GROUP BY` with `DATEPART()` to extract month and year.
- Calculate monthly revenue using `SUM()`.
- Sort and interpret aggregated results.

7.7.3 Prerequisites

- SQL Server 2025 installed and running.
- SQL Server Management Studio (SSMS) 21.x installed.
- `AdventureWorks2022` database restored (from Exercise 1).
- A basic understanding of SELECT queries and aggregate functions.

7.7.4 Steps

Here’s a step-by-step guide to complete this exercise:

7.7.4.1 Step 1: Connect to SQL Server and Select Database

Before running any queries, open SSMS and connect to your SQL Server 2025 instance. Once connected, set the context to use the `AdventureWorks2022` database.

```
USE AdventureWorks2022;
GO
```

📌 *This command sets your working database so you can query tables like `Sales.SalesOrderHeader` and `Sales.SalesOrderDetail`.*

7.7.4.2 Step 2: Explore the Sales Data

Let’s review the structure of the sales orders. The key table is `Sales.SalesOrderHeader`, which contains order dates and total due amounts.

```
SELECT TOP 10 OrderDate, TotalDue
FROM Sales.SalesOrderHeader
ORDER BY OrderDate DESC;
```

📌 *You’ll notice that `OrderDate` is a datetime column and `TotalDue` includes tax, shipping, and discounts—perfect for calculating revenue.*

7.7.4.3 Step 3: Group Sales by Month and Year

Now let’s write a query that groups sales by **month** and **year**, and then calculates the **total revenue** per group.

```
SELECT
    DATEPART(YEAR, OrderDate) AS OrderYear,
    DATEPART(MONTH, OrderDate) AS OrderMonth,
    SUM(TotalDue) AS MonthlyRevenue
FROM Sales.SalesOrderHeader
GROUP BY
    DATEPART(YEAR, OrderDate),
    DATEPART(MONTH, OrderDate)
ORDER BY
    OrderYear,
    OrderMonth;
```

📌 *We use `DATEPART(YEAR, OrderDate)` and `DATEPART(MONTH, OrderDate)` to extract year and month. `SUM(TotalDue)` calculates the total revenue for each period. Finally, we `ORDER BY` to make the report chronological.*

7.7.4.4 Step 4: Format Month-Year for Readability (Optional)

To improve readability, you can format the month and year into a single column (e.g., `2025-01`).

```
SELECT
    FORMAT(OrderDate, 'yyyy-MM') AS OrderPeriod,
    SUM(TotalDue) AS MonthlyRevenue
FROM Sales.SalesOrderHeader
GROUP BY FORMAT(OrderDate, 'yyyy-MM')
ORDER BY OrderPeriod;
```

📌 *`FORMAT()` helps you create a user-friendly label for reporting. It's especially useful if you're feeding data into Excel or Power BI dashboards.*

7.7.4.5 Step 5: Filter Specific Year (Optional)

If you're only interested in a specific year—say 2013—you can use `WHERE` with `DATEPART`.

```
SELECT
    FORMAT(OrderDate, 'yyyy-MM') AS OrderPeriod,
    SUM(TotalDue) AS MonthlyRevenue
FROM Sales.SalesOrderHeader
WHERE DATEPART(YEAR, OrderDate) = 2013
GROUP BY FORMAT(OrderDate, 'yyyy-MM')
ORDER BY OrderPeriod;
```

📌 *This query helps focus the summary on just one year, which is useful for year-over-year analysis.*

7.7.5 Summary

In this exercise, you:

- Connected to the `AdventureWorks2022` database using SSMS.
- Explored sales data in the `Sales.SalesOrderHeader` table.
- Used `GROUP BY` and `DATEPART` to summarize revenue by month and year.
- Enhanced readability using `FORMAT()`.
- Applied filtering to focus on specific years.

✅ *This foundational analysis is essential for generating executive summaries, financial reports, and feeding BI dashboards. You now have the SQL skills to explore temporal trends and revenue cycles in a business context.*

7.8 Exercise 16: Create Pivoted Sales Report

7.8.1 Description

In this hands-on exercise, you will learn how to use SQL Server’s `PIVOT` operator to **transform row-level data into column-based summaries**. This is especially useful when creating **cross-tab reports** such as monthly totals by region, sales by product category, or order status breakdowns. You will use data from the `AdventureWorks2022` database.

7.8.2 Objectives

By the end of this exercise, you will be able to:

- Use a subquery to prepare data for pivoting.
- Apply the `PIVOT` operator to aggregate values and convert rows to columns.
- Generate a dynamic report layout that’s easier to interpret visually.

7.8.3 Prerequisites

- SQL Server 2025 installed and running.
- SQL Server Management Studio (SSMS) 21.x.
- `AdventureWorks2022` database restored (as completed in Exercise 1).
- Familiarity with `GROUP BY` and aggregate functions like `SUM()`.

7.8.4 Steps

Here’s a step-by-step guide to complete this exercise:

7.8.4.1 Step 1: Connect to SQL Server and Select the Database

Start by connecting to your SQL Server instance in SSMS. Then, set your active database context to `AdventureWorks2022`.

```
USE AdventureWorks2022;
GO
```

📌 *This ensures all your queries run against the correct dataset.*

7.8.4.2 Step 2: Understand the Source Data for Pivoting

We will build a report showing **total sales (TotalDue)** for each **Sales Territory** across **different order statuses** (e.g., In Progress, Shipped, Cancelled).

Let's explore the base table:

```
SELECT TOP 10
    soh.SalesOrderID,
    st.Name AS Territory,
    soh.Status,
    soh.TotalDue
FROM Sales.SalesOrderHeader soh
JOIN Sales.SalesTerritory st ON soh.TerritoryID = st.TerritoryID;
```

📌 *This query retrieves the key fields we'll use: territory, status, and total revenue. The `Status` column is an integer from 1–6 indicating order state.*

7.8.4.3 Step 3: Prepare the Data to Be Pivoted

Let's build the inner query that `PIVOT` will use:

```
SELECT
    st.Name AS Territory,
    soh.Status,
    soh.TotalDue
FROM Sales.SalesOrderHeader soh
JOIN Sales.SalesTerritory st ON soh.TerritoryID = st.TerritoryID;
```

📌 *This query selects the **territory**, **status**, and **sales amount**, which we will later convert into a pivoted format.*

7.8.4.4 Step 4: Apply the PIVOT Operator

We now use `PIVOT` to transform **status values into columns**, showing the **total revenue per status** for each territory.

```
SELECT *
FROM (
    SELECT
        st.Name AS Territory,
        soh.Status,
        soh.TotalDue
    FROM Sales.SalesOrderHeader soh
    JOIN Sales.SalesTerritory st ON soh.TerritoryID = st.TerritoryID
) AS SourceData
PIVOT (
    SUM(TotalDue)
    FOR Status IN ([1], [2], [3], [4], [5], [6])
) AS PivotTable
ORDER BY Territory;
```

📌 *Each number in `IN ([1], [2], ..., [6])` corresponds to a sales order status. The pivoted result shows **territories as rows** and **statuses as columns** with aggregated `TotalDue`.*

> *Tip: Status codes can be interpreted as: `1 = In Process`, `2 = Approved`, `3 = Backordered`, etc. (check BOL for full mapping).*

7.8.4.5 Step 5: Optional – Add Readable Status Labels (via View or Report Layer)

While the pivot output shows numeric status columns (1 to 6), you may use aliases or map them in your application/reporting layer for readability.

Example:

```
-- In reporting tool: rename column [1] as 'In Process', [2] as 'Approved', etc.
```

📌 *SQL Server's PIVOT requires static column names—dynamic pivoting requires dynamic SQL, which is beyond this beginner exercise.*

7.8.5 Summary

In this lab, you:

- Explored order data grouped by sales territory and status.
- Wrote a query to prepare the data for pivoting.
- Used the `PIVOT` operator to turn row values (`Status`) into column headers.
- Created a cross-tab report that summarizes total revenue by territory and order status.

✅ *Pivoting is essential for transforming operational data into clear business reports—especially in dashboards, Excel exports, or Power BI models.*

7.9 Exercise 17: Filter Aggregated Results Using HAVING

7.9.1 Description

In this exercise, you'll learn how to use the `HAVING` clause to **filter grouped results** based on aggregate values. Unlike `WHERE`, which filters individual rows, `HAVING` filters **after aggregation**, making it ideal for reporting scenarios where only high-performing products, customers, or territories should be included.

7.9.2 Objectives

By the end of this exercise, you will be able to:

- Differentiate between `WHERE` and `HAVING`.
- Use `HAVING` to filter groups by aggregated conditions.
- Build queries that report only relevant summarized data (e.g., top-selling products).

7.9.3 Prerequisites

- SQL Server 2025 installed and running.
- SQL Server Management Studio (SSMS) 21.x.
- `AdventureWorks2022` database restored (as per Exercise 1).
- Familiarity with `GROUP BY` and aggregate functions like `SUM()` or `COUNT()`.

7.9.4 Steps

Here’s a step-by-step guide to complete this exercise:

7.9.4.1 Step 1: Connect to SQL Server and Use the Database

Start by connecting to your SQL Server instance in SSMS, and set the context to `AdventureWorks2022`.

```
USE AdventureWorks2022;
GO
```

📌 *Always ensure you’re working in the correct database before running queries.*

7.9.4.2 Step 2: Explore the Sales Data by Territory

We’ll group order data by **sales territory** to calculate **total revenue** and **number of orders** per region.

```
SELECT
    st.Name AS Territory,
    COUNT(soh.SalesOrderID) AS OrderCount,
    SUM(soh.TotalDue) AS TotalRevenue
FROM Sales.SalesOrderHeader soh
JOIN Sales.SalesTerritory st ON soh.TerritoryID = st.TerritoryID
GROUP BY st.Name
ORDER BY TotalRevenue DESC;
```

📌 *This gives a summary of sales per territory. Every territory is included at this point, regardless of performance.*

7.9.4.3 Step 3: Apply HAVING to Filter Only High-Revenue Territories

Now let’s say you only want to include **territories with revenue over $10,000,000**. You’ll add a `HAVING` clause to filter based on the `SUM()` result.

```
SELECT
    st.Name AS Territory,
    COUNT(soh.SalesOrderID) AS OrderCount,
    SUM(soh.TotalDue) AS TotalRevenue
FROM Sales.SalesOrderHeader soh
```

```sql
JOIN Sales.SalesTerritory st ON soh.TerritoryID = st.TerritoryID
GROUP BY st.Name
HAVING SUM(soh.TotalDue) > 10000000
ORDER BY TotalRevenue DESC;
```

📌 *`HAVING` filters the grouped result after aggregation. Unlike `WHERE`, you* ***can use aggregate functions*** *like `SUM()` inside `HAVING`.*

7.9.4.4 Step 4: Filter Groups by Multiple Conditions

You can combine aggregate conditions using logical operators like `AND` or `OR`.

```sql
SELECT
    st.Name AS Territory,
    COUNT(soh.SalesOrderID) AS OrderCount,
    SUM(soh.TotalDue) AS TotalRevenue
FROM Sales.SalesOrderHeader soh
JOIN Sales.SalesTerritory st ON soh.TerritoryID = st.TerritoryID
GROUP BY st.Name
HAVING
    SUM(soh.TotalDue) > 5000000 AND
    COUNT(soh.SalesOrderID) > 100;
```

📌 *This returns only territories that have both high total revenue* ***and*** *a high number of orders—great for identifying your best-performing regions.*

7.9.4.5 Step 5: Optional – Compare with WHERE

Try filtering with `WHERE` to see how it differs.

```sql
-- This filters individual orders BEFORE aggregation
SELECT
    st.Name AS Territory,
    COUNT(soh.SalesOrderID) AS OrderCount,
    SUM(soh.TotalDue) AS TotalRevenue
FROM Sales.SalesOrderHeader soh
JOIN Sales.SalesTerritory st ON soh.TerritoryID = st.TerritoryID
WHERE soh.TotalDue > 10000 -- filters orders over 10K only
GROUP BY st.Name
ORDER BY TotalRevenue DESC;
```

📌 *This query only includes orders over $10,000 before grouping. Compare this with `HAVING`, which filters based on the summary outcome.*

7.9.5 Summary

In this exercise, you:

- Used `GROUP BY` to summarize sales data by territory.
- Applied `HAVING` to filter results **after aggregation**.
- Combined `HAVING` conditions for more precise reporting.
- Learned the difference between `WHERE` (row-level) and `HAVING` (group-level) filtering.

✅ *`HAVING` is a crucial tool for data analysts when refining reports to show only relevant, high-level summaries—particularly in dashboards and KPI reports.*

7.10 Conclusion

In this chapter, we explored how to summarize and transform data using `GROUP BY`, aggregate functions, and `PIVOT` queries in SQL Server 2025. We learned how to group rows by categories, filter groups with `HAVING`, and create cross-tab reports using `PIVOT`. These techniques are essential for reporting, analytics, and deriving business intelligence directly from structured data.

8 Joins and UNION Queries

Real-world business data is rarely stored in a single table. In this chapter, you'll learn how to **combine data from multiple tables** using different types of **JOINs**, and how to **merge result sets** using `UNION` and `UNION ALL`. These techniques are fundamental for querying **relational datasets** in SQL Server 2025.

8.1 Introduction to Joins

A **JOIN** allows you to **combine rows** from two or more tables based on a related column, usually a **primary-foreign key relationship**.

Joins can be categorized into several types, each serving different purposes:

Join Type	Description
`INNER JOIN`	Returns only matching rows in both tables
`LEFT JOIN`	Returns all rows from the left table and matches from the right
`FULL JOIN`	Returns all rows when there's a match in one or both tables

8.2 INNER JOIN

An `INNER JOIN` returns only the rows where there is a match in both tables. It's the most common type of join and is used when you want to retrieve related data from multiple tables.

> *Returns only rows where there is a match in **both** tables.*

Following is the basic syntax for an `INNER JOIN`:

```
SELECT a.Column1, b.Column2
FROM TableA a
INNER JOIN TableB b ON a.Key = b.Key;
```

For example, to retrieve orders along with customer details, you can use an `INNER JOIN` between the `Orders` and `Customers` tables:

```
SELECT o.OrderID, o.OrderDate, c.FirstName, c.LastName
FROM Sales.Orders o
INNER JOIN Sales.Customers c ON o.CustomerID = c.CustomerID;
```

This query retrieves all orders along with the corresponding customer names, but only for customers who have placed orders.

8.3 LEFT JOIN (LEFT OUTER JOIN)

Returns **all rows from the left table**, and matching rows from the right. If there’s no match, right-side columns are `NULL`.

For example, to find all customers and their orders, including those who haven’t placed any orders, you can use a `LEFT JOIN`:

```
SELECT c.CustomerID, c.FirstName, o.OrderID
FROM Sales.Customers c
LEFT JOIN Sales.Orders o ON c.CustomerID = o.CustomerID;
```

> *Useful for identifying records **without matching data** (e.g., customers who haven’t ordered).*

8.4 FULL JOIN (FULL OUTER JOIN)

Returns all rows from **both** tables. If there’s no match on either side, the unmatched side will contain `NULL`.

For example, to retrieve all customers and all orders, including those without matches, you can use a `FULL JOIN`:

```
SELECT c.CustomerID, o.OrderID, c.FirstName, o.OrderDate
FROM Sales.Customers c
FULL JOIN Sales.Orders o ON c.CustomerID = o.CustomerID;
```

> *Ideal when comparing two sets and wanting **everything**, including non-matching entries.*

8.5 UNION vs UNION ALL

Both are used to **combine rows** from two or more **separate queries** with the **same number and type of columns**.

You can see the differences in how they handle duplicates:

Feature	`UNION`	`UNION ALL`
Removes duplicates	Yes	No
Includes all rows	No (only unique rows)	Yes (all rows, including duplicates)
Performance	Slower (due to duplicate removal)	Faster (no duplicate check)
Use case	When you need unique results	When you want all results, including duplicates

For both `UNION` and `UNION ALL`, the syntax is similar:

```
SELECT column1, column2 FROM TableA
UNION
SELECT column1, column2 FROM TableB;
```

To combine domestic and international orders into a single result set, you can use `UNION ALL`:

```
SELECT OrderID, CustomerID, 'Domestic' AS Source
FROM Sales.DomesticOrders
```

```
UNION ALL

SELECT OrderID, CustomerID, 'International' AS Source
FROM Sales.InternationalOrders;
```

8.6 Best Practices

- Always specify **columns** instead of `SELECT *` in joins for clarity and performance.
- When using `LEFT JOIN`, test for missing matches with `IS NULL`.
- Always **align data types and column order** when using `UNION` or `UNION ALL`.

8.7 Exercise 18: Combine Customer, Order, and Region Data

8.7.1 Description

In this exercise, you will combine **customer**, **sales order**, and **region** information using `JOIN` operations in SQL Server 2025. The goal is to produce a report that provides meaningful business insights—such as who ordered, what they ordered, when, and from which region—by querying across **multiple related tables** in the `AdventureWorks2022` database.

8.7.2 Objectives

By the end of this exercise, you will be able to:

- Use `INNER JOIN` to combine logically related tables.
- Join three or more tables in a single query.
- Select and format relevant columns to generate a sales report.

8.7.3 Prerequisites

- SQL Server 2025 installed and running.
- SQL Server Management Studio (SSMS) 21.x.

- `AdventureWorks2022` database restored (from Exercise 1).
- Understanding of primary/foreign key relationships and basic `SELECT` queries.

8.7.4 Steps

Here's a step-by-step guide to complete this exercise:

8.7.4.1 Step 1: Set the Working Database

First, open SSMS, connect to your SQL Server instance, and set your session to use the `AdventureWorks2022` database.

```sql
USE AdventureWorks2022;
GO
```

📌 *This ensures that your queries are executed against the correct data context.*

8.7.4.2 Step 2: Explore the Relevant Tables

You will use the following tables:

- `Sales.Customer`: customer details
- `Sales.SalesOrderHeader`: order headers with dates and totals
- `Sales.SalesTerritory`: region/territory information

Let's quickly preview these tables.

```sql
-- Preview Customers
SELECT TOP 5 CustomerID, PersonID, StoreID, TerritoryID
FROM Sales.Customer;

-- Preview Orders
SELECT TOP 5 SalesOrderID, CustomerID, OrderDate, TotalDue
FROM Sales.SalesOrderHeader;

-- Preview Territories
SELECT TOP 5 TerritoryID, Name AS RegionName
FROM Sales.SalesTerritory;
```

📌 *These previews help you understand how tables are related. For example, `Customer.TerritoryID` links to `SalesTerritory.TerritoryID`.*

8.7.4.3 Step 3: Join Customers with Orders

Let’s join the `Customer` and `SalesOrderHeader` tables using `CustomerID` to retrieve customer-related orders.

```
SELECT
    c.CustomerID,
    soh.SalesOrderID,
    soh.OrderDate,
    soh.TotalDue
FROM Sales.Customer c
INNER JOIN Sales.SalesOrderHeader soh
    ON c.CustomerID = soh.CustomerID;
```

📌 *This query lists all customer orders with totals. The `INNER JOIN` ensures we only get orders from existing customers.*

8.7.4.4 Step 4: Join Region Data with Customer and Orders

Now let’s add the third table, `SalesTerritory`, using `TerritoryID`. This enriches the result with regional context.

```
SELECT
    c.CustomerID,
    soh.SalesOrderID,
    soh.OrderDate,
    soh.TotalDue,
    st.Name AS Region
FROM Sales.Customer c
INNER JOIN Sales.SalesOrderHeader soh
    ON c.CustomerID = soh.CustomerID
INNER JOIN Sales.SalesTerritory st
    ON c.TerritoryID = st.TerritoryID
ORDER BY soh.OrderDate DESC;
```

📌 *The report now includes not just who and what was ordered, but also **where the customer is located**, using region names like “Northwest”, “Southwest”, etc.*

8.7.4.5 Step 5: Optional – Add Filtering for a Specific Region

To generate a regional report (e.g., for the “Southwest”), you can add a `WHERE` clause.

```
SELECT
    c.CustomerID,
    soh.SalesOrderID,
    soh.OrderDate,
    soh.TotalDue,
    st.Name AS Region
FROM Sales.Customer c
INNER JOIN Sales.SalesOrderHeader soh
    ON c.CustomerID = soh.CustomerID
INNER JOIN Sales.SalesTerritory st
    ON c.TerritoryID = st.TerritoryID
WHERE st.Name = 'Southwest'
ORDER BY soh.OrderDate DESC;
```

📌 *This version filters the report to a specific region—ideal for regional sales managers.*

8.7.5 Summary

In this hands-on lab, you:

- Used `INNER JOIN` to combine customer and order data.
- Extended your query to include regional sales territory information.
- Generated a combined report with customer ID, order details, and region.
- Practiced adding filters for region-specific reporting.

✅ *By mastering multi-table joins, you’re able to build comprehensive views of your business data—critical for analysis, reporting, and executive decision-making.*

8.8 Exercise 19: Merge Archived and Active Records

8.8.1 Description

In this hands-on lab, you’ll learn how to combine data from **two similar tables**—one active and one archived—using the `UNION` operator. You’ll simulate a scenario where sales records are stored in an **active** table (`Sales.SalesOrderHeader`) and an **archived** version (`Sales.SalesOrderHeaderArchive`, created for this exercise). You'll use `UNION` to merge both sources into a single view while tagging each record with a **source indicator**.

8.8.2 Objectives

By the end of this exercise, you will be able to:

- Understand the difference between `UNION` and `UNION ALL`.
- Merge datasets with similar structure.
- Add source indicators (e.g., "Active" vs. "Archive") to distinguish merged rows.

8.8.3 Prerequisites

- SQL Server 2025 installed and running.
- SQL Server Management Studio (SSMS) 21.x.
- `AdventureWorks2022` database restored.
- `Sales.SalesOrderHeaderArchive` table created (included below).
- Basic understanding of `SELECT` and `JOIN` statements.

8.8.4 Steps

Here's a step-by-step guide to complete this exercise:

8.8.4.1 Step 1: Set the Active Database

Ensure you're working in the `AdventureWorks2022` database:

```
USE AdventureWorks2022;
GO
```

📌 *This will ensure your queries execute against the right context.*

8.8.4.2 Step 2: Create an Archive Table for Simulation

For the purpose of this lab, let's create a simplified archive table by copying a subset of historical data from `Sales.SalesOrderHeader`.

```
SELECT *
INTO Sales.SalesOrderHeaderArchive
FROM Sales.SalesOrderHeader
WHERE OrderDate < '2013-01-01';
```

📌 *This creates `SalesOrderHeaderArchive` with the same schema and data from older records.*

8.8.4.3 Step 3: Preview the Active and Archived Data

Use basic queries to inspect both tables:

```
-- Active Orders
SELECT TOP 5 SalesOrderID, OrderDate, TotalDue
FROM Sales.SalesOrderHeader
ORDER BY OrderDate DESC;

-- Archived Orders
SELECT TOP 5 SalesOrderID, OrderDate, TotalDue
FROM Sales.SalesOrderHeaderArchive
ORDER BY OrderDate DESC;
```

📌 *This step confirms both tables contain similarly structured data.*

8.8.4.4 Step 4: Combine the Two Sources with UNION

Let's now merge the two datasets using `UNION` and add a **source label** column to track where each row comes from.

```
SELECT
    SalesOrderID,
    OrderDate,
    TotalDue,
    'Active' AS Source
FROM Sales.SalesOrderHeader

UNION

SELECT
    SalesOrderID,
    OrderDate,
    TotalDue,
    'Archive' AS Source
```

```
FROM Sales.SalesOrderHeaderArchive
ORDER BY OrderDate DESC;
```

📌 *This combines the datasets while removing duplicates (if any) between them. The `Source` column helps users distinguish between current and historical orders.*

8.8.4.5 Step 5: Use UNION ALL to Preserve Duplicates

In some scenarios, you might want to preserve all records, even if duplicates exist. Use `UNION ALL` instead:

```
SELECT
    SalesOrderID,
    OrderDate,
    TotalDue,
    'Active' AS Source
FROM Sales.SalesOrderHeader

UNION ALL

SELECT
    SalesOrderID,
    OrderDate,
    TotalDue,
    'Archive' AS Source
FROM Sales.SalesOrderHeaderArchive
ORDER BY OrderDate DESC;
```

📌 *Unlike `UNION`, `UNION ALL` does not perform a distinct sort, and it's generally faster for large datasets.*

8.8.4.6 Step 6: Filter the Combined Result by Source

You can now filter the merged result to view only one category—for example, archived orders only:

```
SELECT *
FROM (
    SELECT SalesOrderID, OrderDate, TotalDue, 'Active' AS Source
    FROM Sales.SalesOrderHeader
    UNION ALL
    SELECT SalesOrderID, OrderDate, TotalDue, 'Archive' AS Source
    FROM Sales.SalesOrderHeaderArchive
) AS CombinedOrders
WHERE Source = 'Archive';
```

📌 *This is useful for reports or dashboards where you want a flexible way to switch between current and archived data.*

8.8.5 Summary

In this hands-on lab, you:

- Created an archive version of `SalesOrderHeader` to simulate historical data.
- Combined archived and active sales records using `UNION` and `UNION ALL`.
- Added a `Source` column to track where each record came from.
- Filtered the combined results to view specific sources.

✅ *Using `UNION` allows you to build unified datasets from distributed sources while maintaining clarity and flexibility—essential in environments that split data for performance or compliance.*

8.9 Conclusion

In this chapter, you learned how to effectively combine data from multiple tables using various types of joins (`INNER JOIN`, `LEFT JOIN`, `FULL JOIN`) and how to merge datasets using `UNION` and `UNION ALL`. These techniques are essential for querying relational databases, enabling you to create comprehensive reports and insights from your data.

9 Trends, Time, and Window Functions

This chapter introduces **window functions** and **date/time functions** in SQL Server 2025. These are powerful tools for tracking business **trends**, computing **rankings**, and performing **time-based analysis**, such as comparing current sales to previous months or identifying top-performing products per region.

9.1 Introduction to Window Functions

Window functions perform calculations **across a set of rows related to the current row**, without collapsing the result into a single group.

They differ from `GROUP BY` because they **preserve row-level detail** while enabling analytics over partitions (e.g., over each customer, region, or month).

9.2 ROW_NUMBER, RANK, and DENSE_RANK

These functions assign a **rank or number** to rows based on specified ordering.

For example, to assign a unique number to each order per customer:

```sql
SELECT
    CustomerID, OrderDate, TotalAmount,
    ROW_NUMBER() OVER (PARTITION BY CustomerID ORDER BY OrderDate) AS OrderNumber
FROM Sales.Orders;
```

> *Example Use Case: Identify each customer's* ***first order****, or paginate data.*

`RANK()` assigns a unique rank to each row within a partition, but if two rows have the same value, they receive the same rank, and the next rank will skip numbers.

```sql
SELECT
    ProductID, Category, TotalSales,
    RANK() OVER (PARTITION BY Category ORDER BY TotalSales DESC) AS SalesRank
FROM Sales.ProductSales;
```

Use case: Top-selling products in each category.

`DENSE_RANK()` assigns ranks without gaps, meaning if two rows share the same rank, the next rank will be the immediate next number.

```
SELECT
    Region, SalesRep, Revenue,
    DENSE_RANK() OVER (PARTITION BY Region ORDER BY Revenue DESC) AS RepRank
FROM Sales.RegionalRevenue;
```

9.3 LEAD and LAG: Accessing Adjacent Rows

These functions let you compare **current row values with previous or next rows**, useful for trends and comparisons.

`LAG()` retrieves a value from a previous row in the result set, allowing you to compare current values with past ones.

```
SELECT
    OrderID, CustomerID, OrderDate, TotalAmount,
    LAG(TotalAmount, 1) OVER (PARTITION BY CustomerID ORDER BY OrderDate) AS PrevOrderAmour
FROM Sales.Orders;
```

`LEAD()` retrieves a value from a subsequent row, enabling you to compare current values with future ones.

```
SELECT
    OrderID, CustomerID, OrderDate, TotalAmount,
    LEAD(TotalAmount) OVER (PARTITION BY CustomerID ORDER BY OrderDate) AS NextOrderAmount
FROM Sales.Orders;
```

Use case: Detect ***sales drops****, gaps in service, or payment delays.*

9.4 DATE and TIME Functions

SQL Server provides many functions for working with dates and times.

Function	Description
`GETDATE()`	Current date and time

Function	Description
`SYSDATETIME()`	Higher precision than `GETDATE()`
`DATEPART()`	Extracts part of a date (e.g., year)
`DATEDIFF()`	Calculates the difference between dates
`EOMONTH()`	End of month for a given date
`FORMAT()`	Format a date/time value as text

For example, to get the current date and time:

```
SELECT
    OrderID, OrderDate,
    YEAR(OrderDate) AS OrderYear,
    MONTH(OrderDate) AS OrderMonth
FROM Sales.Orders;
```

For example, to find the number of days between orders for each customer:

```
SELECT
    CustomerID, OrderID, OrderDate,
    LAG(OrderDate) OVER (PARTITION BY CustomerID ORDER BY OrderDate) AS PrevOrderDate,
    DATEDIFF(day, LAG(OrderDate) OVER (PARTITION BY CustomerID ORDER BY OrderDate), OrderDa
FROM Sales.Orders;
```

To format sales data by month and year, you can use the `FORMAT()` function:

```
SELECT
    FORMAT(OrderDate, 'yyyy-MM') AS OrderPeriod,
    SUM(TotalAmount) AS TotalSales
FROM Sales.Orders
GROUP BY FORMAT(OrderDate, 'yyyy-MM');
```

9.5 Combining Window + Time Analysis

You can combine window functions with date/time functions to analyze trends over time.

For example, to calculate the monthly revenue and compare it with the previous month:

```
SELECT
    FORMAT(OrderDate, 'yyyy-MM') AS Month,
    SUM(TotalAmount) AS MonthlyRevenue,
    LAG(SUM(TotalAmount)) OVER (ORDER BY FORMAT(OrderDate, 'yyyy-MM')) AS PrevMonthRevenue
    LAG(SUM(TotalAmount)) OVER (ORDER BY FORMAT(OrderDate, 'yyyy-MM')) AS PrevMonthRevenue
FROM Sales.Orders
GROUP BY FORMAT(OrderDate, 'yyyy-MM');
```

> *Use `Common Table Expressions (CTE)` if needed for complex aggregations with window functions.*

9.6 Exercise 20: Rank Top Customers Monthly

9.6.1 Description

In this exercise, you will use SQL Server 2025's **window functions**, specifically `RANK()` with `PARTITION BY`, to assign **monthly rankings** to customers based on their total purchases. This approach enables insights like "Top 3 Customers Each Month" and supports leaderboard-style reporting, trend tracking, and KPI analysis.

9.6.2 Objectives

By the end of this exercise, you will be able to:

- Use `RANK()` to assign rank values based on aggregated totals.
- Use `PARTITION BY` to segment ranking per month.
- Combine grouping, aggregation, and windowing in a single query.

9.6.3 Prerequisites

- SQL Server 2025 and SSMS 21.x.
- `AdventureWorks2022` database restored.
- Familiarity with `GROUP BY`, `SUM()`, and subqueries or CTEs.
- Some experience with window functions (`OVER`, `PARTITION BY`, `ORDER BY`).

9.6.4 Steps

Here's a step-by-step guide to complete this exercise:

9.6.4.1 Step 1: Connect to SQL Server and Set the Database

Open SSMS and set the working database to `AdventureWorks2022`.

```
USE AdventureWorks2022;
GO
```

📌 *This ensures all queries run against the correct environment.*

9.6.4.2 Step 2: Understand the Sales Data Structure

We’ll use these tables:

- `Sales.SalesOrderHeader`: contains `CustomerID`, `OrderDate`, `TotalDue`.
- `Sales.Customer`: additional customer info (optional for labeling).

Preview the order data:

```
SELECT TOP 5 SalesOrderID, CustomerID, OrderDate, TotalDue
FROM Sales.SalesOrderHeader
ORDER BY OrderDate DESC;
```

📌 *We’ll rank customers by `SUM(TotalDue)` per month.*

9.6.4.3 Step 3: Aggregate Sales Monthly Per Customer

To prepare for ranking, we first calculate **monthly revenue per customer**.

```
SELECT
    CustomerID,
    FORMAT(OrderDate, 'yyyy-MM') AS OrderMonth,
    SUM(TotalDue) AS MonthlyTotal
FROM Sales.SalesOrderHeader
GROUP BY CustomerID, FORMAT(OrderDate, 'yyyy-MM');
```

📌 *This shows each customer’s total purchases for each month. The `FORMAT()` function helps extract and label the month.*

9.6.4.4 Step 4: Apply RANK() to Determine Top Customers per Month

Now we’ll wrap the aggregation inside a **Common Table Expression (CTE)** and apply `RANK()`:

```
WITH MonthlyCustomerTotals AS (
    SELECT
        CustomerID,
        FORMAT(OrderDate, 'yyyy-MM') AS OrderMonth,
        SUM(TotalDue) AS MonthlyTotal
    FROM Sales.SalesOrderHeader
    GROUP BY CustomerID, FORMAT(OrderDate, 'yyyy-MM')
)
```

```
SELECT
    CustomerID,
    OrderMonth,
    MonthlyTotal,
    RANK() OVER (PARTITION BY OrderMonth ORDER BY MonthlyTotal DESC) AS RevenueRank
FROM MonthlyCustomerTotals
ORDER BY OrderMonth, RevenueRank;
```

📌 *Here's what happens:*

- `PARTITION BY OrderMonth`: resets ranking per month.
- `ORDER BY MonthlyTotal DESC`: highest spenders get lowest rank (1 = top).
- `RANK()` handles ties by skipping numbers if there's a tie.

9.6.4.5 Step 5: Optional – Filter to Top 3 Customers Per Month

To focus only on top performers, wrap the query and filter where `RevenueRank <= 3`.

```
WITH MonthlyCustomerTotals AS (
    SELECT
        CustomerID,
        FORMAT(OrderDate, 'yyyy-MM') AS OrderMonth,
        SUM(TotalDue) AS MonthlyTotal
    FROM Sales.SalesOrderHeader
    GROUP BY CustomerID, FORMAT(OrderDate, 'yyyy-MM')
)
SELECT *
FROM (
    SELECT
        CustomerID,
        OrderMonth,
        MonthlyTotal,
        RANK() OVER (PARTITION BY OrderMonth ORDER BY MonthlyTotal DESC) AS RevenueRank
    FROM MonthlyCustomerTotals
) AS RankedData
WHERE RevenueRank <= 3
ORDER BY OrderMonth, RevenueRank;
```

📌 *This final result gives a leaderboard of the* ***top 3 customers per month****, which is useful for CRM, loyalty programs, or sales contests.*

9.6.5 Summary

In this exercise, you:

- Grouped customer revenue monthly using `FORMAT()` and `SUM()`.
- Applied `RANK()` with `PARTITION BY` to rank customers within each month.
- Filtered to identify the top performers each month.

✅ *Window functions like `RANK()` allow you to build dynamic, trend-based reports with ease—perfect for competitive analysis, top-N reporting, and business dashboards.*

9.7 Exercise 21: Compare Customer Revenue Month-over-Month

9.7.1 Description

This exercise focuses on comparing each customer’s **monthly revenue** to their **previous and next month’s performance** using `LAG()` and `LEAD()` window functions. These functions are ideal for detecting trends like **sales drops**, **growth**, or **inactivity gaps**—all within SQL Server 2025 using built-in analytics capabilities.

9.7.2 Objectives

By the end of this exercise, you will be able to:

- Use `LAG()` and `LEAD()` to reference adjacent rows in partitioned data.
- Track month-to-month performance per customer.
- Calculate growth or decline in revenue over time.

9.7.3 Prerequisites

- SQL Server 2025 and SSMS 21.x.
- `AdventureWorks2022` database restored (as in Exercise 1).
- Familiarity with `GROUP BY`, `SUM()`, and basic window functions like `RANK()`.

9.7.4 Steps

Here’s a step-by-step guide to complete this exercise:

9.7.4.1 Step 1: Connect to SQL Server and Use the Correct Database

Begin by opening SSMS and setting the working database context:

```
USE AdventureWorks2022;
GO
```

📌 *Ensure you are querying within the correct environment.*

9.7.4.2 Step 2: Aggregate Monthly Revenue per Customer

Before applying window functions, let’s aggregate revenue per customer for each month.

```sql
SELECT
    CustomerID,
    FORMAT(OrderDate, 'yyyy-MM') AS OrderMonth,
    SUM(TotalDue) AS MonthlyTotal
FROM Sales.SalesOrderHeader
GROUP BY CustomerID, FORMAT(OrderDate, 'yyyy-MM')
ORDER BY CustomerID, OrderMonth;
```

📌 *We’re preparing a dataset where each row shows how much a customer spent each month.*

9.7.4.3 Step 3: Apply LAG() and LEAD() to Track Adjacent Revenue

Now, use a **CTE** to build the base, then apply `LAG()` and `LEAD()` to compare month-over-month revenue.

```sql
WITH MonthlyRevenue AS (
    SELECT
        CustomerID,
        FORMAT(OrderDate, 'yyyy-MM') AS OrderMonth,
        SUM(TotalDue) AS MonthlyTotal
    FROM Sales.SalesOrderHeader
    GROUP BY CustomerID, FORMAT(OrderDate, 'yyyy-MM')
)
SELECT
    CustomerID,
    OrderMonth,
    MonthlyTotal,
    LAG(MonthlyTotal) OVER (PARTITION BY CustomerID ORDER BY OrderMonth) AS PreviousMonthRe
    LEAD(MonthlyTotal) OVER (PARTITION BY CustomerID ORDER BY OrderMonth) AS NextMonthRever
FROM MonthlyRevenue
ORDER BY CustomerID, OrderMonth;
```

📌 *This allows you to see revenue values from the previous and next month per customer. If `NULL` appears, it means no activity for that customer in that adjacent month.*

9.7.4.4 Step 4: Calculate Month-over-Month Change (Optional)

You can now compute revenue growth or decline using arithmetic on the window values.

```sql
WITH MonthlyRevenue AS (
    SELECT
        CustomerID,
        FORMAT(OrderDate, 'yyyy-MM') AS OrderMonth,
        SUM(TotalDue) AS MonthlyTotal
    FROM Sales.SalesOrderHeader
    GROUP BY CustomerID, FORMAT(OrderDate, 'yyyy-MM')
)
SELECT
    CustomerID,
    OrderMonth,
    MonthlyTotal,
    LAG(MonthlyTotal) OVER (PARTITION BY CustomerID ORDER BY OrderMonth) AS PreviousMonthRe
    MonthlyTotal - LAG(MonthlyTotal) OVER (PARTITION BY CustomerID ORDER BY OrderMonth) AS
FROM MonthlyRevenue
ORDER BY CustomerID, OrderMonth;
```

📌 *This version shows how much each customer's spending changed from the previous month—helpful for detecting retention or drop-off trends.*

9.7.4.5 Step 5: Filter Specific Customer (Optional)

To zoom in on a specific customer's history, use a `WHERE` clause.

```sql
-- Example for CustomerID 11000
WITH MonthlyRevenue AS (
    SELECT
        CustomerID,
        FORMAT(OrderDate, 'yyyy-MM') AS OrderMonth,
        SUM(TotalDue) AS MonthlyTotal
    FROM Sales.SalesOrderHeader
    GROUP BY CustomerID, FORMAT(OrderDate, 'yyyy-MM')
)
SELECT
    CustomerID,
    OrderMonth,
    MonthlyTotal,
    LAG(MonthlyTotal) OVER (PARTITION BY CustomerID ORDER BY OrderMonth) AS PreviousMonthRe
    MonthlyTotal - LAG(MonthlyTotal) OVER (PARTITION BY CustomerID ORDER BY OrderMonth) AS
FROM MonthlyRevenue
WHERE CustomerID = 11000
ORDER BY OrderMonth;
```

📌 *This is useful for account managers or CRM analytics dashboards.*

9.7.5 Summary

In this lab, you:

- Aggregated monthly customer revenue.
- Used `LAG()` and `LEAD()` to compare current revenue to adjacent months.
- Calculated revenue changes to detect growth or decline trends.

✅ *These techniques are critical for trend analysis, performance monitoring, and customer behavior prediction in modern BI systems.*

9.8 Exercise 22: Calculate Moving Averages on Sales

9.8.1 Description

In this hands-on lab, you will calculate **moving averages** for customer sales using SQL Server 2025's windowing capabilities. Moving averages help identify **short-term trends** and **smooth out fluctuations** in revenue over time. You'll use the `AVG()` function with a **window frame** (`ROWS BETWEEN`) to compute rolling revenue insights.

9.8.2 Objectives

By the end of this exercise, you will be able to:

- Use `AVG()` as a window function.
- Define custom **window frames** to calculate moving averages.
- Apply time-based logic using `PARTITION BY` and `ORDER BY`.

9.8.3 Prerequisites

- SQL Server 2025 and SSMS 21.x installed.
- `AdventureWorks2022` database restored (from Exercise 1).
- Familiarity with `OVER`, `PARTITION BY`, and basic aggregate functions.

9.8.4 Steps

Here's a step-by-step guide to complete this exercise:

9.8.4.1 Step 1: Set Up the Database Context

Start your session by selecting the appropriate database:

```
USE AdventureWorks2022;
GO
```

📌 *This ensures you’re working with the correct sample data.*

9.8.4.2 Step 2: Prepare Monthly Customer Revenue

Start by computing **monthly totals** for each customer. You’ll need this base to compute moving averages over months.

```
SELECT
    CustomerID,
    FORMAT(OrderDate, 'yyyy-MM') AS OrderMonth,
    SUM(TotalDue) AS MonthlyTotal
FROM Sales.SalesOrderHeader
GROUP BY CustomerID, FORMAT(OrderDate, 'yyyy-MM');
```

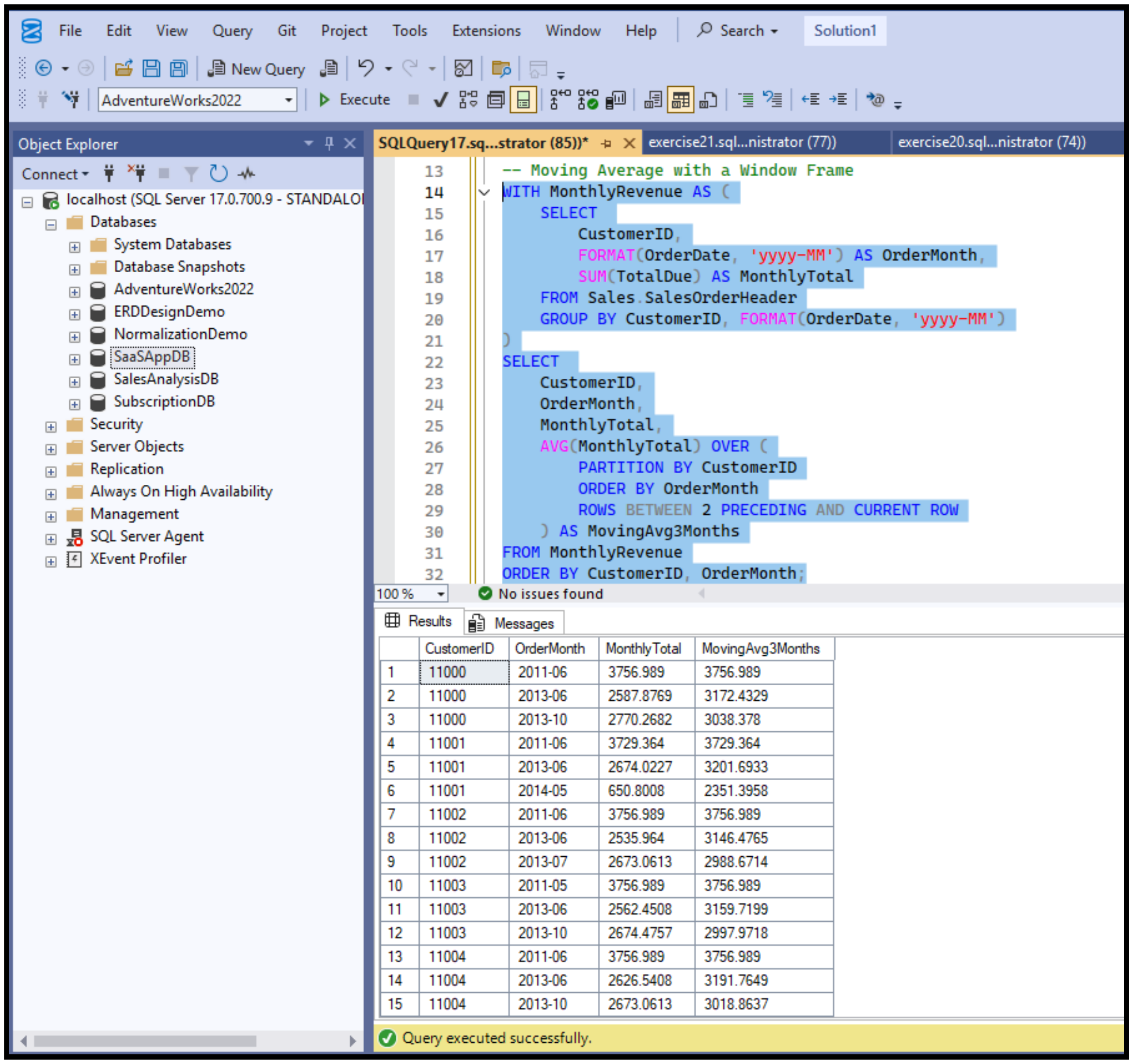
📌 *Each row now represents the revenue a customer generated in a particular month.*

9.8.4.3 Step 3: Apply Moving Average with a Window Frame

Now calculate a **3-month moving average** using `AVG()` with a **sliding window** of the current and previous 2 months.

```
WITH MonthlyRevenue AS (
    SELECT
        CustomerID,
        FORMAT(OrderDate, 'yyyy-MM') AS OrderMonth,
        SUM(TotalDue) AS MonthlyTotal
    FROM Sales.SalesOrderHeader
    GROUP BY CustomerID, FORMAT(OrderDate, 'yyyy-MM')
)
SELECT
    CustomerID,
    OrderMonth,
    MonthlyTotal,
    AVG(MonthlyTotal) OVER (
        PARTITION BY CustomerID
        ORDER BY OrderMonth
        ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
    ) AS MovingAvg3Months
FROM MonthlyRevenue
ORDER BY CustomerID, OrderMonth;
```

📌 *This query calculates the average of the current month and the two preceding months—similar to a 3-month financial moving average.*



```sql
-- Moving Average with a Window Frame
WITH MonthlyRevenue AS (
    SELECT
        CustomerID,
        FORMAT(OrderDate, 'yyyy-MM') AS OrderMonth,
        SUM(TotalDue) AS MonthlyTotal
    FROM Sales.SalesOrderHeader
    GROUP BY CustomerID, FORMAT(OrderDate, 'yyyy-MM')
)
SELECT
    CustomerID,
    OrderMonth,
    MonthlyTotal,
    AVG(MonthlyTotal) OVER (
        PARTITION BY CustomerID
        ORDER BY OrderMonth
        ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
    ) AS MovingAvg3Months
FROM MonthlyRevenue
ORDER BY CustomerID, OrderMonth;
```

	CustomerID	OrderMonth	MonthlyTotal	MovingAvg3Months
1	11000	2011-06	3756.989	3756.989
2	11000	2013-06	2587.8769	3172.4329
3	11000	2013-10	2770.2682	3038.378
4	11001	2011-06	3729.364	3729.364
5	11001	2013-06	2674.0227	3201.6933
6	11001	2014-05	650.8008	2351.3958
7	11002	2011-06	3756.989	3756.989
8	11002	2013-06	2535.964	3146.4765
9	11002	2013-07	2673.0613	2988.6714
10	11003	2011-05	3756.989	3756.989
11	11003	2013-06	2562.4508	3159.7199
12	11003	2013-10	2674.4757	2997.9718
13	11004	2011-06	3756.989	3756.989
14	11004	2013-06	2626.5408	3191.7649
15	11004	2013-10	2673.0613	3018.8637

Query executed successfully.

Figure 9.1: Applying moving average with a window frame.

9.8.4.4 Step 4: Interpret the Moving Average

Look at how the `MovingAvg3Months` column, Figure 9.1, smooths out spikes or dips. For the **first two months** per customer, the average may only include 1 or 2 values (not a full 3-month window), but SQL Server handles this automatically.

9.8.4.5 Step 5: Filter a Specific Customer (Optional)

To see a clean trendline for one customer:

```sql
WITH MonthlyRevenue AS (
    SELECT
        CustomerID,
        FORMAT(OrderDate, 'yyyy-MM') AS OrderMonth,
        SUM(TotalDue) AS MonthlyTotal
    FROM Sales.SalesOrderHeader
    GROUP BY CustomerID, FORMAT(OrderDate, 'yyyy-MM')
)
SELECT
    CustomerID,
    OrderMonth,
    MonthlyTotal,
    AVG(MonthlyTotal) OVER (
        PARTITION BY CustomerID
        ORDER BY OrderMonth
        ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
    ) AS MovingAvg3Months
FROM MonthlyRevenue
WHERE CustomerID = 11000
ORDER BY OrderMonth;
```

📌 *This is useful for visual analysis, business health checks, or feeding into BI dashboards.*

9.8.5 Summary

In this exercise, you:

- Aggregated monthly revenue per customer.
- Applied the `AVG()` function with a rolling window to compute 3-month moving averages.
- Used `ROWS BETWEEN` to create a sliding analytic window.

✅ *This pattern is valuable for understanding revenue stability, seasonality, and forecasting performance over time.*

9.9 Exercise 23: Analyze Customer Sales Percentiles

9.9.1 Description

This hands-on lab explores how to use **percentile-based window functions** in SQL Server 2025. You'll learn how to apply `CUME_DIST()` and `PERCENT_RANK()` to rank customers based on their **total purchases**, providing insight into **how each customer performs relative to others**. These functions are essential in loyalty programs, revenue segmentation, and targeted marketing strategies.

9.9.2 Objectives

By the end of this exercise, you will be able to:

- Calculate **cumulative distribution** and **percentile rank** over a dataset.
- Apply percentile logic using window functions in SQL.
- Identify customers in the top or bottom percentiles of revenue.

9.9.3 Prerequisites

- SQL Server 2025 and SSMS 21.x.
- `AdventureWorks2022` database restored.
- Basic familiarity with `OVER(...)`, `ORDER BY`, and `window functions`.

9.9.4 Steps

Here's a step-by-step guide to complete this exercise:

9.9.4.1 Step 1: Set the Working Database

Begin by setting your database context to `AdventureWorks2022`:

```
USE AdventureWorks2022;
GO
```

📌 *This ensures all subsequent queries run in the correct environment.*

9.9.4.2 Step 2: Compute Total Revenue per Customer

Let's first aggregate total revenue for each customer.

```
SELECT
    CustomerID,
    SUM(TotalDue) AS TotalRevenue
FROM Sales.SalesOrderHeader
GROUP BY CustomerID;
```

📌 *This forms the base dataset used for percentile calculations.*

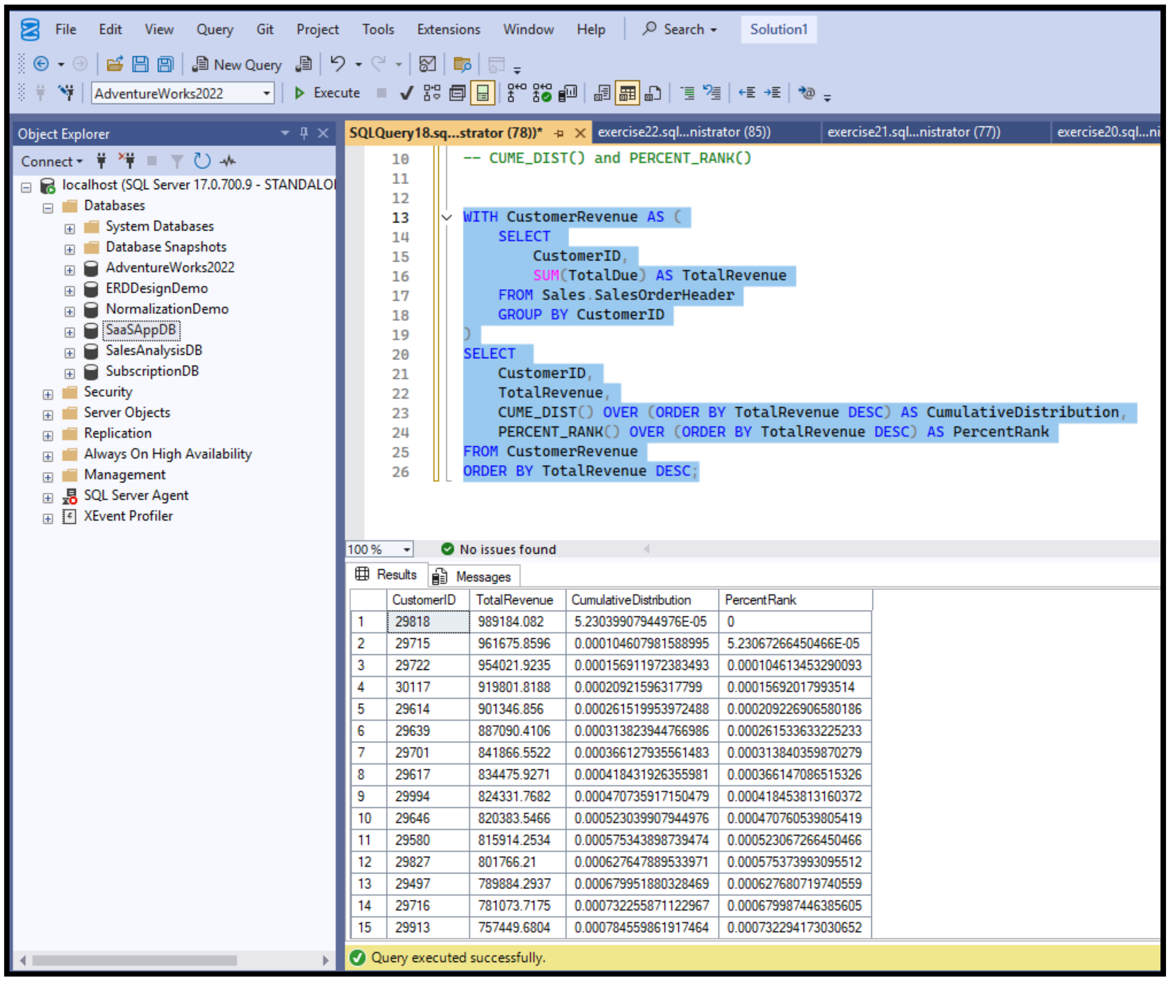
9.9.4.3 Step 3: Apply CUME_DIST() and PERCENT_RANK()

Wrap the previous aggregation in a **Common Table Expression (CTE)** and apply both percentile functions.

```
WITH CustomerRevenue AS (
    SELECT
        CustomerID,
        SUM(TotalDue) AS TotalRevenue
    FROM Sales.SalesOrderHeader
    GROUP BY CustomerID
)
SELECT
    CustomerID,
    TotalRevenue,
    CUME_DIST() OVER (ORDER BY TotalRevenue DESC) AS CumulativeDistribution,
    PERCENT_RANK() OVER (ORDER BY TotalRevenue DESC) AS PercentRank
FROM CustomerRevenue
ORDER BY TotalRevenue DESC;
```

📌 *This returns each customer's revenue along with two metrics:*

- `CUME_DIST()` → shows the **proportion** of customers with **equal or lower** revenue.
- `PERCENT_RANK()` → shows **rank percentile**, with **lowest revenue = 0.0** and **highest = 1.0**.



```
-- CUME_DIST() and PERCENT_RANK()


WITH CustomerRevenue AS (
    SELECT
        CustomerID,
        SUM(TotalDue) AS TotalRevenue
    FROM Sales.SalesOrderHeader
    GROUP BY CustomerID
)
SELECT
    CustomerID,
    TotalRevenue,
    CUME_DIST() OVER (ORDER BY TotalRevenue DESC) AS CumulativeDistribution,
    PERCENT_RANK() OVER (ORDER BY TotalRevenue DESC) AS PercentRank
FROM CustomerRevenue
ORDER BY TotalRevenue DESC;
```

	CustomerID	TotalRevenue	CumulativeDistribution	PercentRank
1	29818	989184.082	5.23039907944976E-05	0
2	29715	961675.8596	0.000104607981588995	5.23067266450466E-05
3	29722	954021.9235	0.000156911972383493	0.000104613453290093
4	30117	919801.8188	0.00020921596317799	0.00015692017993514
5	29614	901346.856	0.000261519953972488	0.000209226906580186
6	29639	887090.4106	0.000313823944766986	0.000261533633225233
7	29701	841866.5522	0.000366127935561483	0.000313840359870279
8	29617	834475.9271	0.000418431926355981	0.000366147086515326
9	29994	824331.7682	0.000470735917150479	0.000418453813160372
10	29646	820383.5466	0.000523039907944976	0.000470760539805419
11	29580	815914.2534	0.000575343898739474	0.000523067266450466
12	29827	801766.21	0.000627647889533971	0.000575373993095512
13	29497	789884.2937	0.000679951880328469	0.000627680719740559
14	29716	781073.7175	0.000732255871122967	0.000679987446385605
15	29913	757449.6804	0.000784559861917464	0.000732294173030652

Query executed successfully.

Figure 9.2: Applying CUME_DIST() and PERCENT_RANK().

9.9.4.4 Step 4: Interpret the Results

- A customer with `CUME_DIST() = 0.95` is among the **top 5%** by revenue.
- A `PERCENT_RANK()` of `0.0` means the **lowest-ranked customer**, while `1.0` is the **highest**.

> *Note: `PERCENT_RANK()` has one fewer steps than total rows, while `CUME_DIST()` includes ties and equals 1 for the topmost record.*

9.9.4.5 Step 5: Filter for High-Performing Customers (Optional)

You can now use a `WHERE` clause to get the **top 10%** customers by revenue:

```
WITH CustomerRevenue AS (
    SELECT
        CustomerID,
        SUM(TotalDue) AS TotalRevenue
    FROM Sales.SalesOrderHeader
    GROUP BY CustomerID
)
SELECT *
FROM (
    SELECT
        CustomerID,
        TotalRevenue,
        CUME_DIST() OVER (ORDER BY TotalRevenue DESC) AS CumulativeDistribution
    FROM CustomerRevenue
) AS RankedCustomers
WHERE CumulativeDistribution <= 0.10
ORDER BY TotalRevenue DESC;
```

📌 *This is powerful for generating targeted sales reports or tiered loyalty segments.*

9.9.5 Summary

In this lab, you:

- Aggregated customer revenue.
- Used `CUME_DIST()` and `PERCENT_RANK()` to evaluate each customer's percentile.
- Identified top-performing customers for targeted analysis.

✅ *Percentile functions allow businesses to contextualize performance within the broader customer base—ideal for ranking, segmentation, and performance* *targeting.*

9.10 Conclusion

In this chapter, you learned how to leverage SQL Server 2025’s powerful window functions and date/time capabilities to analyze trends, compute rankings, and perform time-based analyses. These skills are essential for building advanced analytics solutions, enabling you to derive insights from your data that drive business decisions.

Section 5: Security, Access, and Compliance

10 User Management and Access Control

This chapter introduces essential SQL Server 2025 concepts for **user management** and **access control**, including **logins**, **users**, **roles**, and **schema-level security**. Understanding these concepts is critical for safeguarding sensitive business data and ensuring that access aligns with organizational policies and compliance standards.

10.1 Authentication vs Authorization

Authentication and authorization are two fundamental concepts in SQL Server security.

Concept	Description
Authentication	Confirms **who** the user is (login identity)
Authorization	Defines **what** they are allowed to do (permissions)

SQL Server uses **logins** for authentication and **users/roles/permissions** for authorization.

10.2 Logins and Users

A **login** is a server-level identity that allows a user to connect to SQL Server. A **user** is a database-level identity that maps to a login and defines what the user can do within a specific database.

To create a login, you can use the `CREATE LOGIN` statement:

```sql
CREATE LOGIN reporting_user WITH PASSWORD = 'StrongPassword123!';
```

To create a user in a specific database that maps to the login:

```sql
USE SalesDB;
CREATE USER reporting_user FOR LOGIN reporting_user;
```

Now `reporting_user` can connect to the `SalesDB` database.

10.3 Fixed Server and Database Roles

SQL Server provides **fixed server roles** and **fixed database roles** that grant predefined sets of permissions. These roles simplify user management by grouping common permissions.

The following are some of the key fixed server roles:

Role	Purpose
`sysadmin`	Full control of SQL Server
`securityadmin`	Manage logins and permissions
`serveradmin`	Configure server-wide settings

These roles are specific to a database and control access to its objects:

Role	Description
`db_owner`	Full control over the database
`db_datareader`	Can read all data from all user tables
`db_datawriter`	Can modify (insert/update/delete) data
`db_ddladmin`	Can run `CREATE`, `ALTER`, and `DROP` commands

To assign a role to a user, you can use the `sp_addrolemember` stored procedure:

```
-- Assign reader role
EXEC sp_addrolemember 'db_datareader', 'reporting_user';
```

You can combine roles to tailor permission sets.

10.4 Custom Roles and Role-Based Access Control (RBAC)

You can create **custom roles** to define more granular access control.

We can create a custom role for business analysts that allows them to read data from the `Sales` schema:

```
CREATE ROLE SalesAnalyst;

GRANT SELECT ON SCHEMA::Sales TO SalesAnalyst;

EXEC sp_addrolemember 'SalesAnalyst', 'reporting_user';
```

> *This allows all members of `SalesAnalyst` role to query all tables in the `Sales` schema.*

10.5 Schema-Level Security

Schemas in SQL Server are **containers for database objects**. Permissions can be applied **at the schema level**, which simplifies access management.

To grant a user access to all objects in a schema, you can use the `GRANT` statement at the schema level:

```
GRANT SELECT, INSERT, UPDATE ON SCHEMA::Sales TO SalesAnalyst;
```

> *Rather than granting table-level permissions one by one, this approach simplifies access for business analysts.*

10.6 Security Best Practices

Well-defined security practices are crucial for protecting sensitive data and ensuring compliance with regulations. Here are some best practices:

Practice	Reason

Practice	Reason
Use **least privilege** principle	Prevents accidental or malicious access
Use **roles** instead of user-specific grants	Easier to audit and manage
Separate **data readers** from **data writers**	Ensures role clarity and auditability
Regularly **review user access**	Keep access compliant with org policies
Avoid using `sa` **or sysadmin** for daily tasks	Reduces exposure and risks

10.7 Auditing Access

Auditing is a critical component of a secure SQL Server environment. By monitoring and recording access and activity, organizations can detect unauthorized actions, investigate incidents, and demonstrate compliance with regulatory requirements. SQL Server provides several built-in tools and system views to help administrators track who accessed what data and when.

To track and audit access:

- Use **SQL Server Audit** (Enterprise)
- Enable **login auditing** at server level
- Query system views like `sys.database_principals`, `sys.database_permissions`, `sys.server_principals`

10.8 Exercise 24: Create Analyst Role and Grant Access

10.8.1 Description

In this exercise, you will practice implementing **role-based access control (RBAC)** in SQL Server 2025 using T-SQL. You will create a **new database**, define a **custom database role** named `Analyst`, and grant it **read-only access** to selected tables. This is a fundamental step in enforcing **principle of least privilege** and **securing data access** within business systems.

10.8.2 Objectives

By the end of this exercise, you will be able to:

- Create a new SQL Server database.
- Create a user and assign it to a custom database role.
- Create a role and grant it appropriate read-only permissions.
- Validate role-based access control using a login or impersonation.

10.8.3 Prerequisites

- SQL Server 2025 and SSMS 21.x installed.
- Permission to create databases and logins.
- Basic familiarity with `CREATE DATABASE`, `CREATE LOGIN`, and `GRANT`.

10.8.4 Steps

Here's a step-by-step guide to complete this exercise:

10.8.4.1 Step 1: Create a New Database for Testing

Start by creating a clean database named `SecureDataLab` for this exercise.

```
CREATE DATABASE SecureDataLab;
GO

USE SecureDataLab;
GO
```

📌 *This isolates our RBAC test environment and avoids changes to production databases.*

10.8.4.2 Step 2: Create Sample Tables and Seed Data

Create a few tables that an analyst would typically query—like customers and sales.

```
CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY,
    FullName NVARCHAR(100),
    Email NVARCHAR(100),
    Region NVARCHAR(50)
);

CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    CustomerID INT FOREIGN KEY REFERENCES Customers(CustomerID),
    OrderDate DATE,
    Amount DECIMAL(10,2)
);

-- Insert sample data
INSERT INTO Customers VALUES
(1, 'Ahmad Smith', 'smith@ilmudata.id', 'East'),
(2, 'Laura Ave', 'ave@ilmudata.id', 'West');

INSERT INTO Orders VALUES
(101, 1, '2025-01-15', 1500.00),
(102, 2, '2025-02-10', 2300.00);
```

📌 *These simple tables simulate real business data an analyst might need.*

10.8.4.3 Step 3: Create a Login and Map It to a Database User

Create a SQL Server login and map it to the `SecureDataLab` database as a user.

```
-- Create SQL login
CREATE LOGIN analyst_user WITH PASSWORD = 'StrongP@ssword123!';
GO

-- Map login to database user
USE SecureDataLab;
GO
CREATE USER analyst_user FOR LOGIN analyst_user;
```

📌 *This sets up the user that will be granted controlled access via a role.*

10.8.4.4 Step 4: Create the Analyst Role and Assign Permissions

Now define the custom role `Analyst` and grant it **read-only access** to the tables.

```
-- Create custom role
CREATE ROLE Analyst;
GO

-- Grant SELECT permissions to role
GRANT SELECT ON Customers TO Analyst;
GRANT SELECT ON Orders TO Analyst;
GO

-- Add user to the role
EXEC sp_addrolemember 'Analyst', 'analyst_user';
```

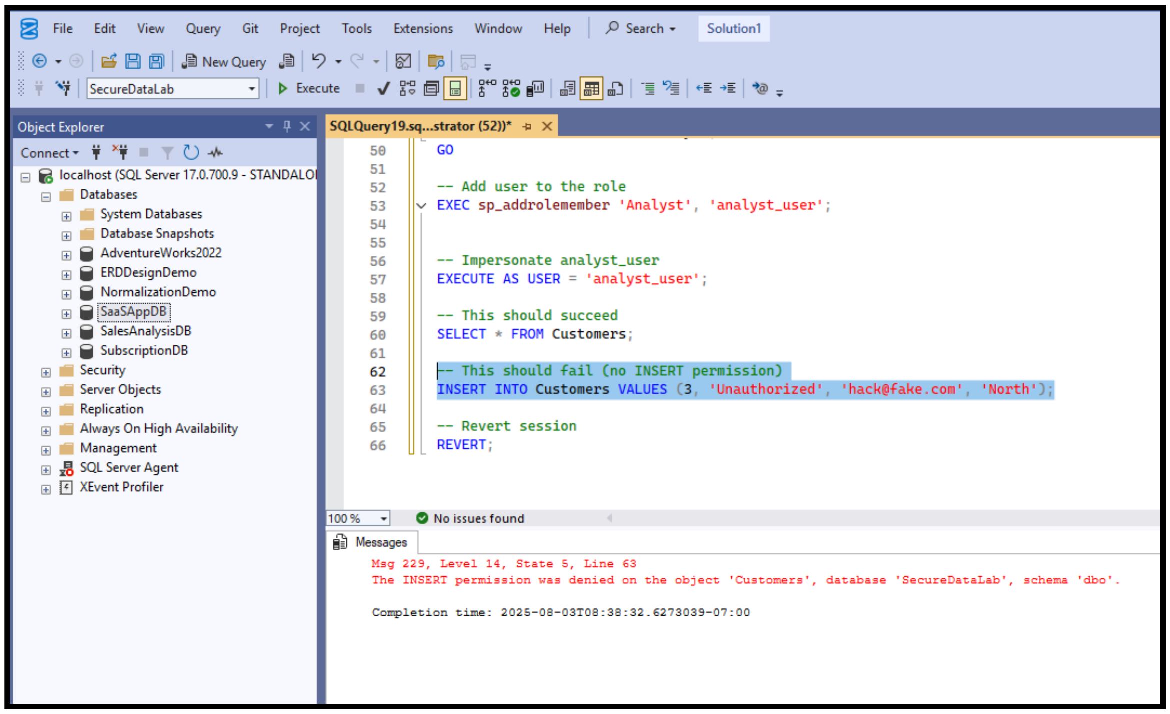
📌 *This ensures the analyst can read but not modify data. You can reuse this role for other analysts in the future.*

10.8.4.5 Step 5: Test Role Access (Optional via EXECUTE AS)

To test permissions from the analyst's perspective:

```sql
-- Impersonate analyst_user
EXECUTE AS USER = 'analyst_user';

-- This should succeed
SELECT * FROM Customers;

-- This should fail (no INSERT permission)
INSERT INTO Customers VALUES (3, 'Unauthorized', 'hack@fake.com', 'North');

-- Revert session
REVERT;
```

📌 *This helps verify that the user has only SELECT access as intended.*



```sql
GO

-- Add user to the role
EXEC sp_addrolemember 'Analyst', 'analyst_user';


-- Impersonate analyst_user
EXECUTE AS USER = 'analyst_user';

-- This should succeed
SELECT * FROM Customers;

-- This should fail (no INSERT permission)
INSERT INTO Customers VALUES (3, 'Unauthorized', 'hack@fake.com', 'North');

-- Revert session
REVERT;
```

```
Msg 229, Level 14, State 5, Line 63
The INSERT permission was denied on the object 'Customers', database 'SecureDataLab', schema 'dbo'.

Completion time: 2025-08-03T08:38:32.6273039-07:00
```

Figure 10.1: Error on inserting data.

10.8.5 Summary

In this hands-on lab, you:

- Created a new database (`SecureDataLab`) for secure access testing.
- Built customer and order tables with sample data.
- Created a SQL login and mapped it to a database user.
- Defined a custom `Analyst` role and granted it read-only access.
- Tested permissions to ensure proper role enforcement.

✅ *This exercise illustrates how to implement practical, secure, and reusable role-based access control strategies using SQL Server 2025.*

10.9 Exercise 25: Restrict Access by Schema

10.9.1 Description

In this exercise, you will practice managing access control at the **schema level** in SQL Server 2025. You'll create **separate schemas** for sensitive and general data, assign different permissions, and restrict access to specific schemas using `GRANT` and `DENY`. Schema-based access control helps manage **logical separation of data** and simplifies security management in multi-user environments.

10.9.2 Objectives

By the end of this exercise, you will be able to:

- Create and use custom schemas.
- Assign ownership to schemas and manage access using `GRANT` and `DENY`.
- Restrict user access based on schema-level permissions.

10.9.3 Prerequisites

- SQL Server 2025 and SSMS 21.x installed.
- `SecureDataLab` database created (see Exercise 24).
- `analyst_user` login and user mapped to the database.
- Familiarity with roles, users, and `GRANT`/`DENY`.

10.9.4 Steps

Here's a step-by-step guide to complete this exercise:

10.9.4.1 Step 1: Use the Existing Database

Ensure you are working within the `SecureDataLab` database:

```
USE SecureDataLab;
GO
```

📌 *This keeps all security exercises contained in one place.*

10.9.4.2 Step 2: Create New Schemas

Let’s define two schemas:

- `PublicData` – for general-purpose access.
- `SensitiveData` – for restricted access.

```
CREATE SCHEMA PublicData;
GO
CREATE SCHEMA SensitiveData;
GO
```

📌 *Schemas act like containers or namespaces for tables, allowing grouped permission management.*

10.9.4.3 Step 3: Create Tables in Each Schema

Now add tables to the new schemas:

```
-- General access table
CREATE TABLE PublicData.SalesSummary (
    Year INT,
    Region NVARCHAR(50),
    TotalSales DECIMAL(12,2)
);

-- Restricted access table
CREATE TABLE SensitiveData.EmployeeSalaries (
    EmployeeID INT,
    FullName NVARCHAR(100),
    Salary DECIMAL(10,2)
);

-- Insert example data
INSERT INTO PublicData.SalesSummary VALUES (2025, 'East', 150000.00);
INSERT INTO SensitiveData.EmployeeSalaries VALUES (1, 'Ahmad Smith', 90000.00);
```

📌 *This simulates a typical setup with public and sensitive business data.*

10.9.4.4 Step 4: Grant Access to Public Schema Only

Grant read access on the `PublicData` schema to the `Analyst` role:

```
GRANT SELECT ON SCHEMA::PublicData TO Analyst;
```

📌 *This enables users in the `Analyst` role (e.g., `analyst_user`) to read all tables in the `PublicData` schema.*

10.9.4.5 Step 5: Explicitly Deny Access to Sensitive Schema

Deny access to the `SensitiveData` schema for the `Analyst` role:

```
DENY SELECT ON SCHEMA::SensitiveData TO Analyst;
```

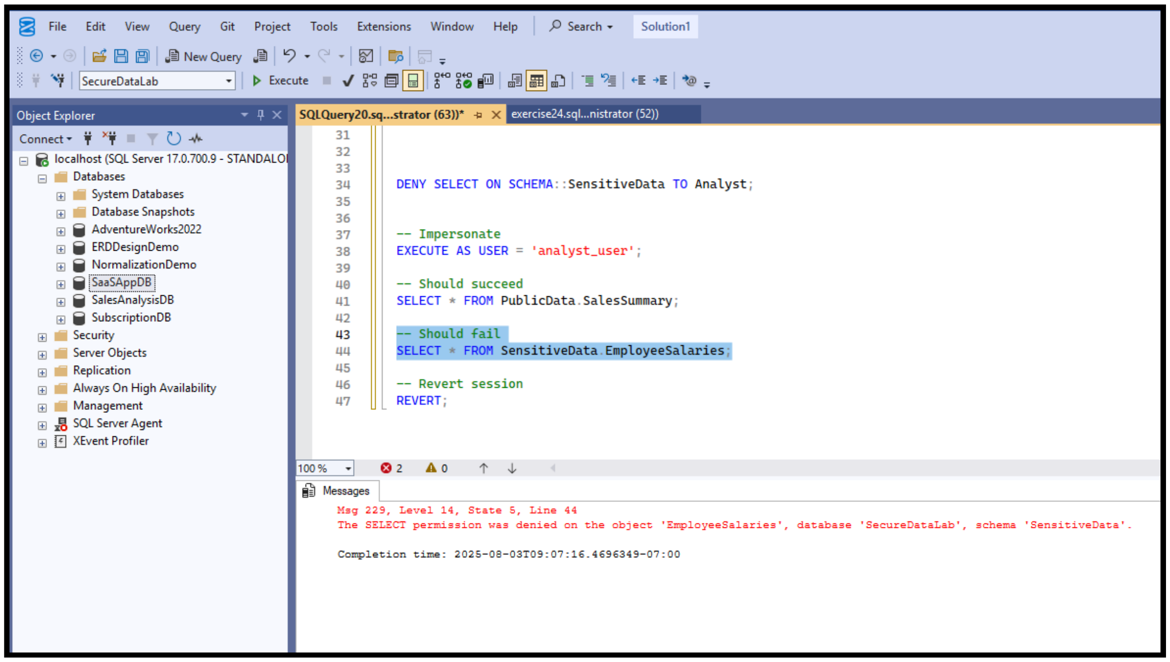
📌 *This ensures that even if a user has broad SELECT rights, they cannot access sensitive tables.*

10.9.4.6 Step 6: Test Access as Analyst User (Optional)

Impersonate the analyst to test access control:

```
-- Impersonate
EXECUTE AS USER = 'analyst_user';

-- Should succeed
SELECT * FROM PublicData.SalesSummary;

-- Should fail
SELECT * FROM SensitiveData.EmployeeSalaries;

-- Revert session
REVERT;
```

📌 *The `SELECT` on `PublicData` works, but the attempt on `SensitiveData` is denied—even though both exist in the same database.*



```sql
DENY SELECT ON SCHEMA::SensitiveData TO Analyst;


-- Impersonate
EXECUTE AS USER = 'analyst_user';

-- Should succeed
SELECT * FROM PublicData.SalesSummary;

-- Should fail
SELECT * FROM SensitiveData.EmployeeSalaries;

-- Revert session
REVERT;
```

```
Msg 229, Level 14, State 5, Line 44
The SELECT permission was denied on the object 'EmployeeSalaries', database 'SecureDataLab', schema 'SensitiveData'.

Completion time: 2025-08-03T09:07:16.4696349-07:00
```

Figure 10.2: Error on selecting data.

10.9.5 Summary

In this exercise, you:

- Created logical schemas for public and sensitive data.
- Used `GRANT` and `DENY` to manage access at the schema level.
- Enforced security boundaries using container-level permission control.

✅ *Managing access by schema allows organizations to apply role-based access rules with clarity and precision, reducing risks of unauthorized access and simplifying permission maintenance.*

10.10 Exercise 26: Revoke Permissions and Audit Role Membership

10.10.1 Description

Security is not just about granting access — it's equally important to *revoke* access when no longer needed and *audit* who has which permissions. This hands-on lab teaches how to revoke object-level permissions from a user or role and how to inspect existing role memberships and permission assignments in SQL Server.

These skills are crucial for implementing the **Principle of Least Privilege** and ensuring data security.

10.10.2 Objectives

By the end of this lab, you will be able to:

- Revoke permissions from users and roles
- Audit role membership using system views
- Identify permission grants on objects
- Use queries to document access control configuration

10.10.3 Prerequisites

- SQL Server 2025
- SQL Server Management Studio (SSMS) 21.x
- A dedicated database for this exercise

10.10.4 Steps

Here’s a step-by-step guide to complete this exercise:

10.10.4.1 Step 1: Create a Dedicated Security Demo Database

We’ll start by creating a new database to isolate our permission configuration and auditing.

```
CREATE DATABASE SecurityAuditDemo;
GO
USE SecurityAuditDemo;
GO
```

✅ This ensures your practice won’t interfere with production or existing configurations.

10.10.4.2 Step 2: Create Tables and Seed Data

Create a simple table to simulate sensitive data access.

```
CREATE TABLE Sales (
    SaleId INT IDENTITY PRIMARY KEY,
    ProductName NVARCHAR(100),
    Amount MONEY
);

INSERT INTO Sales (ProductName, Amount)
VALUES ('Widget A', 1500), ('Widget B', 2400), ('Widget C', 720);
GO
```

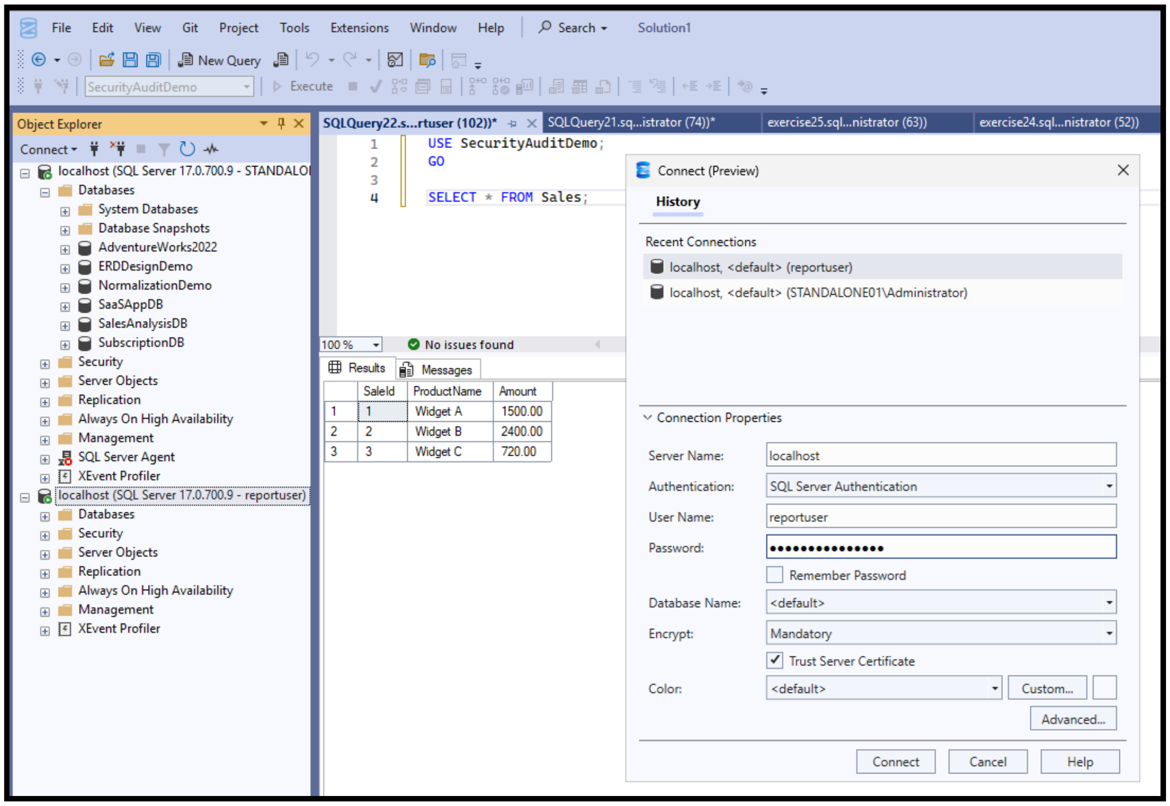
✅ This is our target object for permission testing.

10.10.4.3 Step 3: Create Users and Assign Permissions

We’ll create a test login, database user, and assign SELECT permission.

```
CREATE LOGIN reportuser WITH PASSWORD = 'ComplexPass123!';
CREATE USER reportuser FOR LOGIN reportuser;

GRANT SELECT ON dbo.Sales TO reportuser;
GO
```

✅ At this point, the `reportuser` has access to query the Sales table.



```
USE SecurityAuditDemo;
GO

SELECT * FROM Sales;
```

	SaleId	ProductName	Amount
1	1	Widget A	1500.00
2	2	Widget B	2400.00
3	3	Widget C	720.00

Server Name:	localhost
Authentication:	SQL Server Authentication
User Name:	reportuser
Password:	•••••••••••••••
	☐ Remember Password
Database Name:	<default>
Encrypt:	Mandatory
	✓ Trust Server Certificate
Color:	<default>

Figure 10.3: Login as reportuser to SQL Server.

Try to login as `reportuser` and run:

```
SELECT * FROM dbo.Sales;
```

10.10.4.4 Step 4: Revoke the SELECT Permission

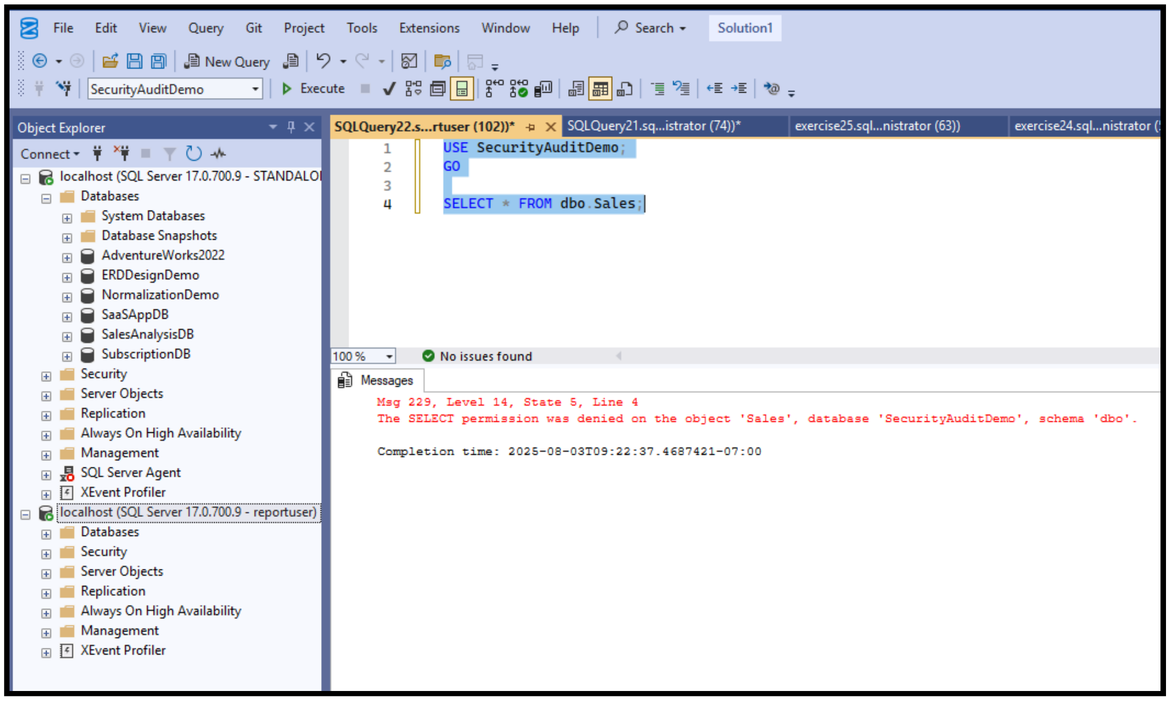
Now simulate revoking access, such as when a user changes role or leaves the team.

```
REVOKE SELECT ON dbo.Sales FROM reportuser;
GO
```

✅ This removes the permission without deleting the user.

Try running the same `SELECT` query again as `reportuser`:

```
SELECT * FROM dbo.Sales;
```



```
USE SecurityAuditDemo;
GO

SELECT * FROM dbo.Sales;
```

```
Msg 229, Level 14, State 5, Line 4
The SELECT permission was denied on the object 'Sales', database 'SecurityAuditDemo', schema 'dbo'.

Completion time: 2025-08-03T09:22:37.4687421-07:00
```

Figure 10.4: Error on selecting data while signing as reportuser to SQL Server.

10.10.4.5 Step 5: Create a Custom Role and Grant Permission

Create a role for analysts and assign permission at the role level.

```
CREATE ROLE analyst_role;
EXEC sp_addrolemember 'analyst_role', 'reportuser';

GRANT SELECT ON dbo.Sales TO analyst_role;
GO
```

✅ This demonstrates permission via roles rather than direct user-level grants.

10.10.4.6 Step 6: Audit Role Membership

Let’s now check which users belong to which roles.

```
SELECT
    r.name AS RoleName,
    m.name AS MemberName
FROM
    sys.database_role_members drm
JOIN
    sys.database_principals r ON drm.role_principal_id = r.principal_id
JOIN
    sys.database_principals m ON drm.member_principal_id = m.principal_id;
```

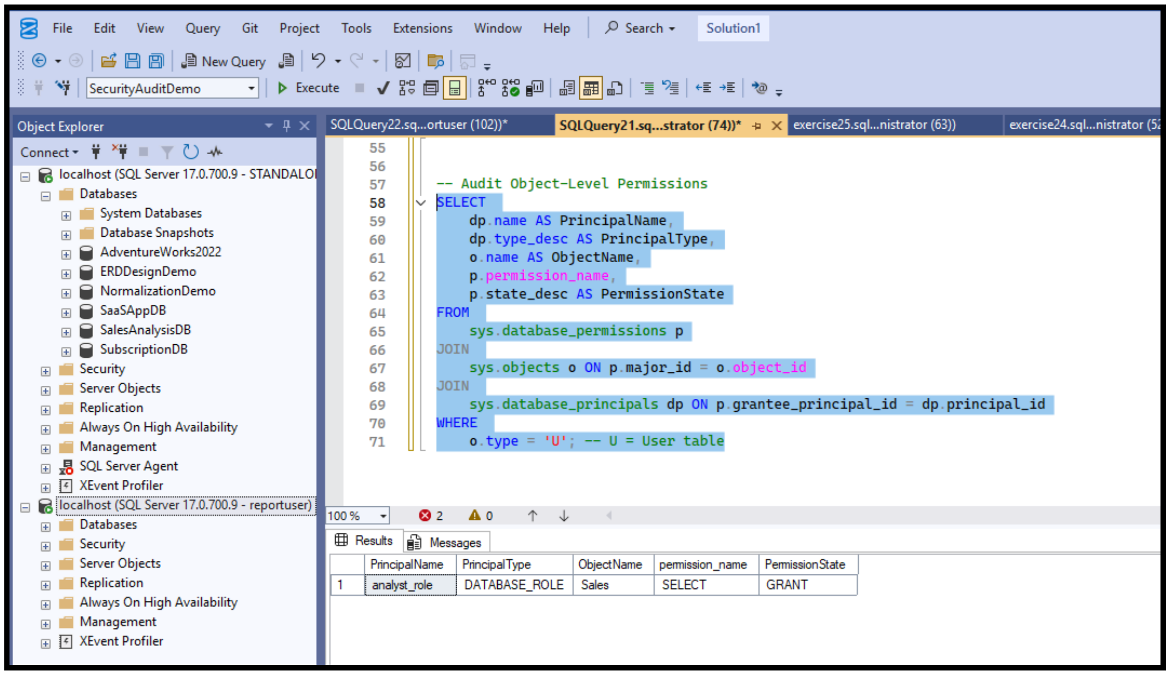
✅ This is useful for documenting role-based access control (RBAC).

10.10.4.7 Step 7: Audit Object-Level Permissions

You can check what permissions each user or role has using this query:

```
SELECT
    dp.name AS PrincipalName,
    dp.type_desc AS PrincipalType,
    o.name AS ObjectName,
    p.permission_name,
    p.state_desc AS PermissionState
FROM
    sys.database_permissions p
JOIN
    sys.objects o ON p.major_id = o.object_id
JOIN
    sys.database_principals dp ON p.grantee_principal_id = dp.principal_id
WHERE
    o.type = 'U'; -- U = User table
```

✅ This helps you monitor what access is currently in place.



```
-- Audit Object-Level Permissions
SELECT
    dp.name AS PrincipalName,
    dp.type_desc AS PrincipalType,
    o.name AS ObjectName,
    p.permission_name,
    p.state_desc AS PermissionState
FROM
    sys.database_permissions p
JOIN
    sys.objects o ON p.major_id = o.object_id
JOIN
    sys.database_principals dp ON p.grantee_principal_id = dp.principal_id
WHERE
    o.type = 'U'; -- U = User table
```

	PrincipalName	PrincipalType	ObjectName	permission_name	PermissionState
1	analyst_role	DATABASE_ROLE	Sales	SELECT	GRANT

Figure 10.5: Checking permissions for each user.

10.10.5 Summary

In this hands-on lab, you:

- Created a security sandbox with a simple table
- Granted and revoked permissions using `GRANT` and `REVOKE`
- Created a custom role and added a user to it
- Audited role membership and permission assignments using system views

✅ These are foundational skills for securing a SQL Server environment, maintaining compliance, and managing user access lifecycles.

10.11 Conclusion

In this chapter, you learned how to manage user access and permissions in SQL Server 2025. You explored concepts such as logins, users, roles, and schema-level security. You also practiced creating custom roles, granting permissions, and auditing access control configurations. These skills are essential for ensuring data security, compliance with organizational policies, and effective role-based access control in SQL Server environments.

11 Row-Level Security and Tenant Isolation

In a multi-user or multi-tenant system, data isolation is critical. SQL Server 2025 supports **Row-Level Security (RLS)**—a powerful feature that filters records dynamically based on the **user's identity** or **session context**. This chapter covers how to implement RLS for **tenant isolation** and **fine-grained access control**, ensuring each user only sees the data they are authorized to access.

11.1 What Is Row-Level Security (RLS)?

Row-Level Security enables **automatic filtering of rows** based on **rules tied to the current user's context**—without rewriting the queries. It adds a security layer directly at the database level.

Why is RLS important?

- Enforces **data privacy** at the lowest level.
- Centralizes filtering logic, reducing app-side complexity.
- Supports **multi-tenant SaaS** models with shared tables.
- Helps meet **compliance** and audit requirements (e.g., GDPR, HIPAA).

11.2 RLS Architecture in SQL Server

RLS works by defining a **security predicate function** that returns a table of rows the user is allowed to see. This function is then applied through a **security policy** to one or more tables.

11.2.1 How RLS Works Internally

When a query is executed against a table with RLS enabled, SQL Server automatically appends the security predicate to the query's WHERE clause. This happens transparently—users and applications do not need to change their queries. The filtering logic is enforced at the storage engine level, ensuring that unauthorized rows are never returned, even if users attempt to bypass application logic.

11.2.2 Types of Security Predicates

There are two main types of security predicates in SQL Server RLS:

- **FILTER Predicate:** Restricts which rows are visible to users. This is the most common use case for tenant isolation.
- **BLOCK Predicate:** Prevents unauthorized users from performing certain actions (like UPDATE or DELETE) on rows they should not access.

You can combine both predicate types for more granular control, such as allowing users to see some rows but not modify them.

11.2.3 Security Policy Management

A security policy is a database object that binds one or more predicate functions to tables. Policies can be enabled or disabled without dropping them, making it easy to test or roll back RLS configurations. Policies can also be applied to multiple tables, allowing for consistent enforcement across your data model.

11.2.4 Auditing RLS Activity

Because RLS operates at the database level, all access attempts—successful or blocked—are logged in SQL Server's audit logs. This provides a reliable audit trail for compliance and security reviews. You can further enhance auditing by combining RLS with SQL Server Audit or Extended Events to track access patterns and detect suspicious activity.

11.3 Example Scenario: Tenant-Based Filtering

Assuming we have a shared `Orders` table that contains orders from multiple tenants, we can use RLS to ensure that each tenant only sees their own orders.

```sql
CREATE TABLE Sales.Orders (
    OrderID INT PRIMARY KEY,
    TenantID INT,
    CustomerID INT,
    TotalAmount DECIMAL(10,2),
    OrderDate DATE
);
```

Each tenant's data is distinguished by the `TenantID` column.

11.4 Step-by-Step: Implementing RLS for Tenant Isolation

Follow these steps to implement Row-Level Security for tenant isolation in SQL Server:

1. Step 1: Enable SESSION_CONTEXT for Tenant ID

The application must set the current tenant using:

```
EXEC sp_set_session_context @key = N'TenantID', @value = 101;
```

2. Step 2: Create Predicate Function

Create a security predicate function that checks the `TenantID` against the session context:

```
CREATE FUNCTION Security.fn_TenantPredicate(@TenantID INT)
RETURNS TABLE
WITH SCHEMABINDING
AS
RETURN
    SELECT 1 AS Result
    WHERE @TenantID = CAST(SESSION_CONTEXT(N'TenantID') AS INT);
```

This function compares the current row's `TenantID` to the session's context value.

3. Step 3: Apply Security Policy to Table

We need to create a security policy that uses this predicate function:

```
CREATE SECURITY POLICY Security.TenantPolicy
ADD FILTER PREDICATE Security.fn_TenantPredicate(TenantID)
ON Sales.Orders
WITH (STATE = ON);
```

This policy ensures that any query against the `Sales.Orders` table will automatically filter rows based on the current tenant's context.

4. Step 4: Test the Filtering

We can now test the RLS implementation by querying the `Orders` table:

```
-- Set current tenant to 101
EXEC sp_set_session_context @key = N'TenantID', @value = 101;

-- Run unrestricted query
SELECT * FROM Sales.Orders;
```

This query will only return rows where `TenantID = 101`, effectively isolating the tenant's data.

11.5 RLS for User-Specific Access

RLS can also be used for **user identity-based filtering**, such as allowing users to only see their own records.

For instance, consider an `EmployeeDocuments` table where each employee can only access their own documents:

```
CREATE TABLE HR.EmployeeDocuments (
    DocID INT,
    EmployeeID INT,
    DocumentName NVARCHAR(100)
);
```

To filter documents based on the logged-in user, we can create a predicate function that checks the `EmployeeID` against the current user's login:

```
CREATE FUNCTION HR.fn_UserAccessPredicate(@EmployeeID INT)
RETURNS TABLE
WITH SCHEMABINDING
AS
RETURN
    SELECT 1 AS Result
    WHERE @EmployeeID = (
        SELECT EmployeeID FROM HR.Users WHERE LoginName = SYSTEM_USER
    );
```

Use `SYSTEM_USER` or `ORIGINAL_LOGIN()` to get current login name.

11.6 Best Practices for RLS

When implementing Row-Level Security, consider these best practices:

Practice	Benefit

Practice	Benefit
Use `SESSION_CONTEXT()` for multi-tenant apps	Better than parsing usernames
Avoid hardcoding user logic in app	Centralizes logic in DB
Keep security predicate functions simple	Required for performance and indexing
Use separate **security schemas** for RLS logic	Improves clarity and modularity
Combine with **views or stored procedures** for added control	Simplifies reporting and access

11.7 RLS Limitations to Note

For all its power, RLS has some limitations:

Limitation	Notes
No support for `TEXT`, `NTEXT`, `IMAGE` columns	Must be excluded
Not compatible with `INSTEAD OF` triggers	Will be blocked
Applies only to **SELECT**, **UPDATE**, **DELETE**	Not enforced on **INSERT** logic
Admins (`sysadmin`) are exempt	They bypass RLS automatically

11.8 Exercise 27: Enforce Tenant Filtering with RLS

11.8.1 Description

Row-Level Security (RLS) allows SQL Server to restrict access to rows in a table based on the executing user's context. In this exercise, you'll simulate a **multi-tenant SaaS environment** by applying RLS so each tenant can access only their own data. This is especially important for applications requiring **strict data isolation** without duplicating schema or database logic.

11.8.2 Objectives

By completing this exercise, you will:

- Understand the purpose and implementation of Row-Level Security.
- Learn to create and bind a **security policy** to filter data.
- Implement `SESSION_CONTEXT()` to set the current tenant at runtime.

11.8.3 Prerequisites

- SQL Server 2025 and SSMS 21.x installed.
- User has `sysadmin` or `db_owner` rights on the server.
- Basic familiarity with SQL Server logins, users, and `SESSION_CONTEXT()`.

11.8.4 Steps

Here’s a step-by-step guide to complete this exercise:

11.8.4.1 Step 1: Create a New Database for RLS Demo

Create a clean database named `RlsTenantLab`.

```
CREATE DATABASE RlsTenantLab;
GO

USE RlsTenantLab;
GO
```

📌 *This isolates our RLS demonstration from other labs and databases.*

11.8.4.2 Step 2: Create a Table with Tenant Data

Create a `CustomerOrders` table that includes a `TenantId` column used to filter data.

```
CREATE TABLE CustomerOrders (
    OrderID INT IDENTITY PRIMARY KEY,
    TenantId INT NOT NULL,
    CustomerName NVARCHAR(100),
    OrderDate DATE,
    Amount DECIMAL(10,2)
);
```

📌 *Each row is tagged with a `TenantId`, which determines which tenant “owns” the record.*

11.8.4.3 Step 3: Insert Sample Tenant Data

Insert rows for two different tenants:

```
INSERT INTO CustomerOrders (TenantId, CustomerName, OrderDate, Amount)
VALUES
(1, 'Tata Industries', '2025-07-01', 1200.00),
(1, 'Tata Industries', '2025-07-15', 3000.00),
(2, 'Infosys Ltd', '2025-07-02', 1500.00),
(2, 'Infosys Ltd', '2025-07-20', 500.00);
```

📌 *This simulates two tenants: Tenant 1 and Tenant 2.*

11.8.4.4 Step 4: Create a Predicate Function for RLS

Create an inline table-valued function to check if the current row’s `TenantId` matches the user’s context.

```
CREATE FUNCTION fn_tenant_filter(@TenantId INT)
RETURNS TABLE
WITH SCHEMABINDING
AS
RETURN SELECT 1 AS result
WHERE @TenantId = CAST(SESSION_CONTEXT(N'TenantId') AS INT);
```

📌 *This function uses `SESSION_CONTEXT('TenantId')` which you’ll set dynamically in your app or session.*

11.8.4.5 Step 5: Create a Security Policy Using the Predicate

Bind the filter function to the `CustomerOrders` table:

```
CREATE SECURITY POLICY TenantFilterPolicy
ADD FILTER PREDICATE dbo.fn_tenant_filter(TenantId)
ON dbo.CustomerOrders
WITH (STATE = ON);
```

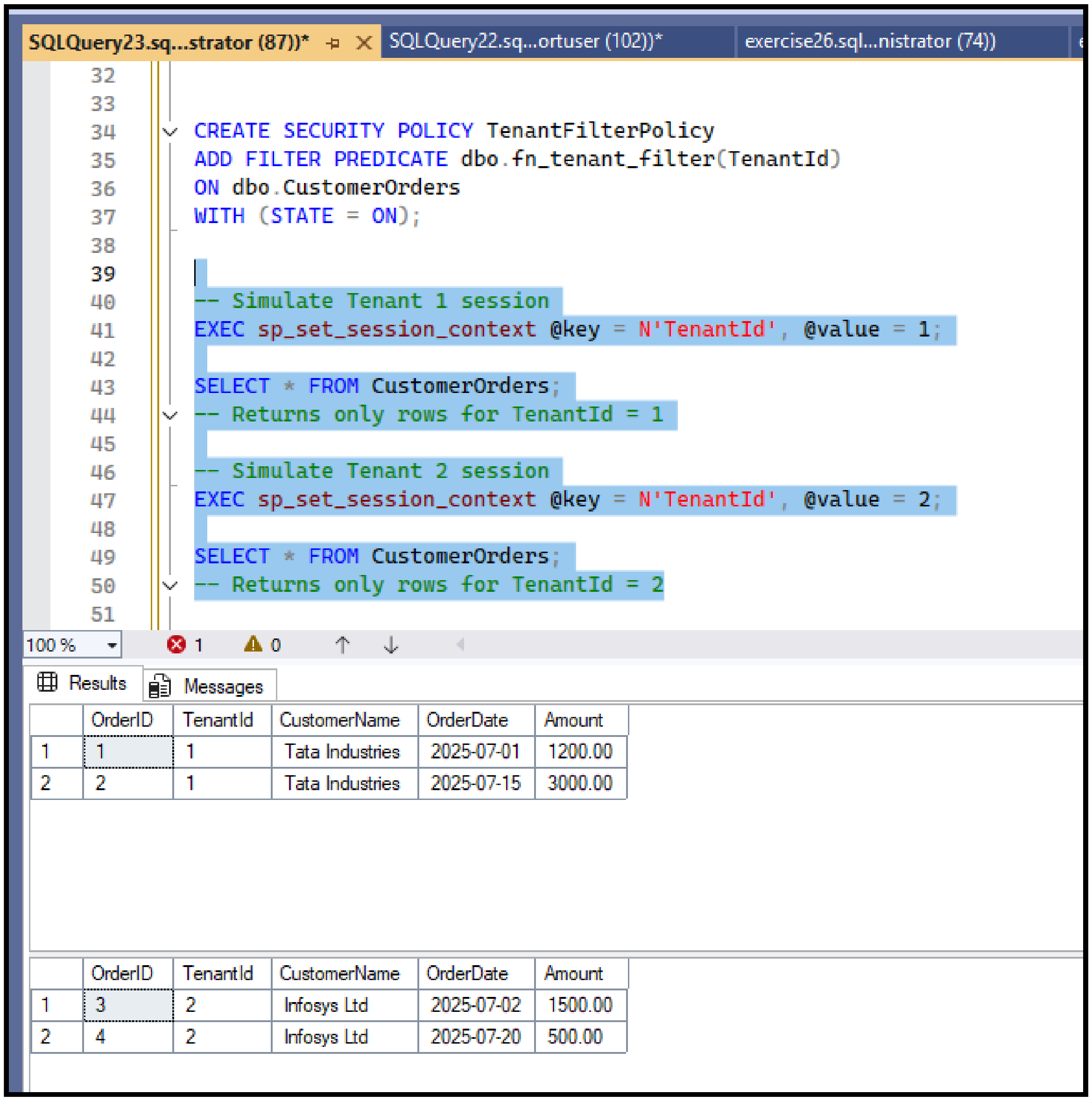
📌 *This policy ensures that whenever a query is run against `CustomerOrders`, only matching rows for the current `TenantId` are returned.*

11.8.4.6 Step 6: Test the Policy with Different Tenant Contexts

Set the tenant context using `SESSION_CONTEXT`, then query the data.

```sql
-- Simulate Tenant 1 session
EXEC sp_set_session_context @key = N'TenantId', @value = 1;

SELECT * FROM CustomerOrders;
-- Returns only rows for TenantId = 1

-- Simulate Tenant 2 session
EXEC sp_set_session_context @key = N'TenantId', @value = 2;

SELECT * FROM CustomerOrders;
-- Returns only rows for TenantId = 2
```

📌 *The RLS filter activates automatically. If the context is not set or doesn’t match, zero rows are returned.*


SQLQuery23.sq...strator (87))*
SQLQuery22.sq...ortuser (102))*
exercise26.sql...nistrator (74))
100 %
Results
Messages

```
CREATE SECURITY POLICY TenantFilterPolicy
ADD FILTER PREDICATE dbo.fn_tenant_filter(TenantId)
ON dbo.CustomerOrders
WITH (STATE = ON);


-- Simulate Tenant 1 session
EXEC sp_set_session_context @key = N'TenantId', @value = 1;

SELECT * FROM CustomerOrders;
-- Returns only rows for TenantId = 1

-- Simulate Tenant 2 session
EXEC sp_set_session_context @key = N'TenantId', @value = 2;

SELECT * FROM CustomerOrders;
-- Returns only rows for TenantId = 2
```

	OrderID	TenantId	CustomerName	OrderDate	Amount
1	1	1	Tata Industries	2025-07-01	1200.00
2	2	1	Tata Industries	2025-07-15	3000.00

	OrderID	TenantId	CustomerName	OrderDate	Amount
1	3	2	Infosys Ltd	2025-07-02	1500.00
2	4	2	Infosys Ltd	2025-07-20	500.00

Figure 11.1: Quering by tenant Id.

11.8.4.7 Step 7: Optional – Add a Role and User for Tenant Access

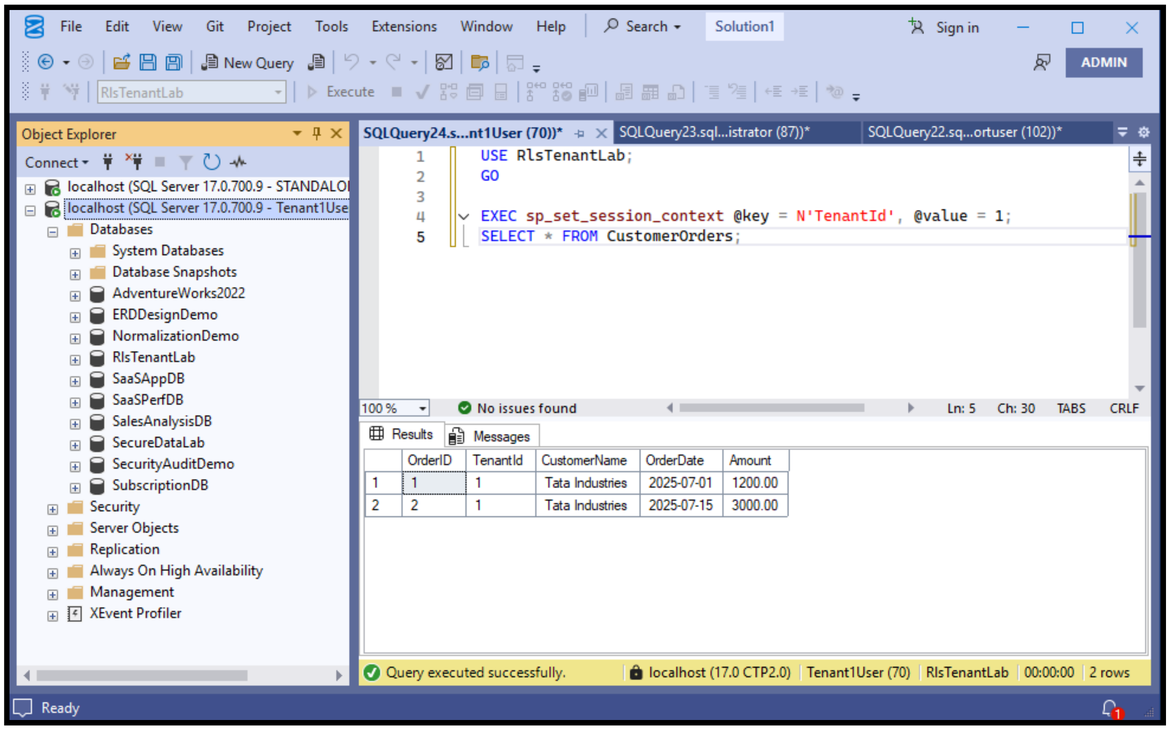
You can simulate an app login per tenant and use a login trigger or app logic to set the `SESSION_CONTEXT`.

```
-- Create login and user for Tenant1
CREATE LOGIN Tenant1User WITH PASSWORD = 'StrongP@ssword123!';
CREATE USER Tenant1User FOR LOGIN Tenant1User;

GRANT SELECT ON CustomerOrders TO Tenant1User;
```

Then log in as `Tenant1User`, execute:

```
EXEC sp_set_session_context @key = N'TenantId', @value = 1;
SELECT * FROM CustomerOrders;
```

📌 *This setup simulates app-level row-level access without changing application logic.*



```
USE RlsTenantLab;
GO

EXEC sp_set_session_context @key = N'TenantId', @value = 1;
SELECT * FROM CustomerOrders;
```

	OrderID	TenantId	CustomerName	OrderDate	Amount
1	1	1	Tata Industries	2025-07-01	1200.00
2	2	1	Tata Industries	2025-07-15	3000.00

Query executed successfully. | localhost (17.0 CTP2.0) | Tenant1User (70) | RlsTenantLab | 00:00:00 | 2 rows

Figure 11.2: Login as Tenant1User and then perform a query.

11.8.5 Summary

In this exercise, you:

- Created a predicate function and bound it via a security policy.
- Used `SESSION_CONTEXT()` to control visibility by tenant.
- Enforced tenant-level filtering **without altering query logic**.

✅ *Row-Level Security (RLS) is a powerful feature for tenant isolation and is critical for modern SaaS, multi-tenant apps using SQL Server.*

11.9 Exercise 28: Validate Isolation Using Test Accounts

11.9.1 Description

After applying Row-Level Security (RLS), it's essential to **verify tenant isolation** by simulating access from different users. In this lab, you'll create test user accounts that represent different tenants and validate that each user only sees their own data—even when using the same queries.

11.9.2 Objectives

By the end of this lab, you will:

- Create separate users for different tenants.
- Use login sessions to simulate application users.
- Confirm that RLS policies enforce isolation without needing to alter application code.

11.9.3 Prerequisites

- Completed Exercise 27 (*Enforce Tenant Filtering with RLS*).
- Database `RlsTenantLab` with RLS already applied.
- SQL Server 2025 and SSMS 21.x installed.
- Login privileges to create users, roles, and set session context.

11.9.4 Steps

Here's a step-by-step guide to complete this exercise:

11.9.4.1 Step 1: Reuse or Confirm Setup from Exercise

Ensure you're using the same database and that the `CustomerOrders` table, `fn_tenant_filter()` function, and `TenantFilterPolicy` are already present and active.

```
USE RlsTenantLab;
GO

-- Confirm table
SELECT * FROM sys.tables WHERE name = 'CustomerOrders';

-- Confirm security policy
SELECT * FROM sys.security_policies WHERE name = 'TenantFilterPolicy';
```

📌 *If the policy or function is missing, complete Exercise 26 before continuing.*

11.9.4.2 Step 2: Create Test Logins and Users for Tenants

Create one SQL login and user per tenant. Each will have read-only access to the table.

```
-- Tenant 1
CREATE LOGIN Tenant1User1 WITH PASSWORD = 'Tenant1Pass!';
CREATE USER Tenant1User1 FOR LOGIN Tenant1User1;
GRANT SELECT ON dbo.CustomerOrders TO Tenant1User1;

-- Tenant 2
CREATE LOGIN Tenant2User1 WITH PASSWORD = 'Tenant2Pass!';
CREATE USER Tenant2User1 FOR LOGIN Tenant2User1;
GRANT SELECT ON dbo.CustomerOrders TO Tenant2User1;
```

📌 *These users simulate tenant accounts connecting to the database from an external app or reporting tool.*

11.9.4.3 Step 3: Create a Logon Trigger to Set Tenant Context

Use a logon trigger to automatically set the correct `TenantId` in the session context based on login.

```
CREATE OR ALTER TRIGGER trg_SetTenantContext
ON ALL SERVER
FOR LOGON
AS
BEGIN
    DECLARE @tenantId INT;

    IF ORIGINAL_LOGIN() = 'Tenant1User1'
        SET @tenantId = 1;
    ELSE IF ORIGINAL_LOGIN() = 'Tenant2User1'
        SET @tenantId = 2;

    EXECUTE AS LOGIN = ORIGINAL_LOGIN();
    EXEC sp_set_session_context @key = N'TenantId', @value = @tenantId;
    REVERT;
END;
```

📌 *This ensures each tenant automatically gets the correct `TenantId` set during login.*

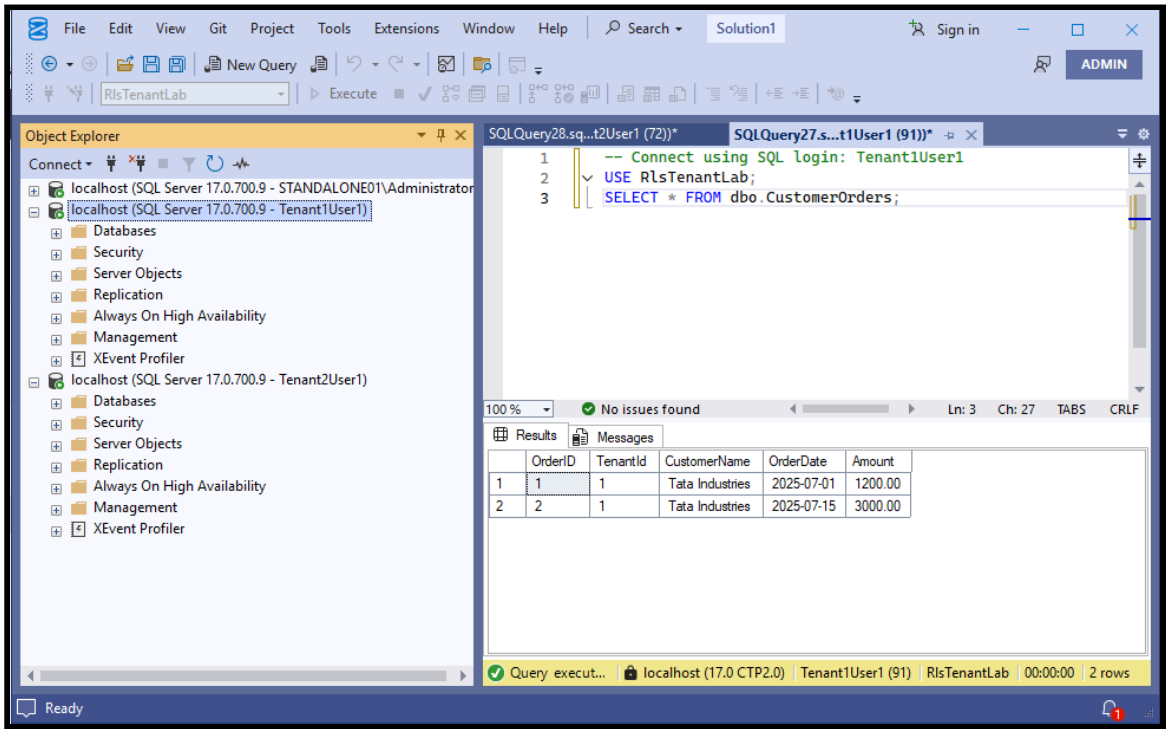
11.9.4.4 Step 4: Test Isolation via SSMS Query Windows

Open two SSMS sessions with different logins and test each account's visibility.

Session A — Login as Tenant1User1

```
-- Connect using SQL login: Tenant1User1
USE RlsTenantLab;
SELECT * FROM dbo.CustomerOrders;
```

Expected output: Only orders where `TenantId = 1`.



```
localhost (SQL Server 17.0.700.9 - STANDALONE01\Administrator
localhost (SQL Server 17.0.700.9 - Tenant1User1)
    Databases
    Security
    Server Objects
    Replication
    Always On High Availability
    Management
    XEvent Profiler
localhost (SQL Server 17.0.700.9 - Tenant2User1)
    Databases
    Security
    Server Objects
    Replication
    Always On High Availability
    Management
    XEvent Profiler
```

```sql
-- Connect using SQL login: Tenant1User1
USE RlsTenantLab;
SELECT * FROM dbo.CustomerOrders;
```

	OrderID	TenantId	CustomerName	OrderDate	Amount
1	1	1	Tata Industries	2025-07-01	1200.00
2	2	1	Tata Industries	2025-07-15	3000.00

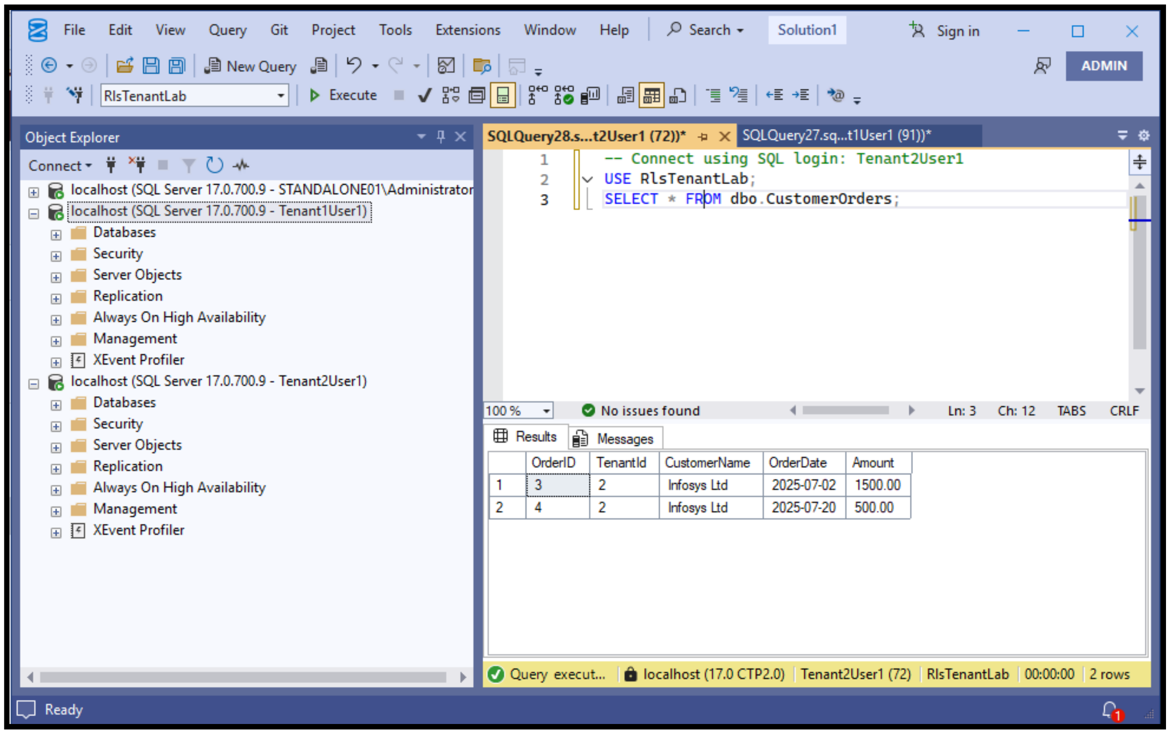
```
Query execut... | localhost (17.0 CTP2.0) | Tenant1User1 (91) | RlsTenantLab | 00:00:00 | 2 rows
```

Figure 11.3: Login as Tenant1User1 and then perform a query.

Session B — Login as Tenant2User1

```sql
-- Connect using SQL login: Tenant2User1
USE RlsTenantLab;
SELECT * FROM dbo.CustomerOrders;
```

Expected output: Only orders where `TenantId = 2`.



```
-- Connect using SQL login: Tenant2User1
USE RlsTenantLab;
SELECT * FROM dbo.CustomerOrders;
```

	OrderID	TenantId	CustomerName	OrderDate	Amount
1	3	2	Infosys Ltd	2025-07-02	1500.00
2	4	2	Infosys Ltd	2025-07-20	500.00

Figure 11.4: Login as Tenant2User1 and then perform a query.

📌 *Notice that both users run the same query but only see their own rows due to RLS.*

11.9.4.5 Step 5: Try Bypassing RLS (Readonly Context Prevents Override)

By default, `sp_set_session_context` can override the value set by the logon trigger. To prevent this, set the context value as `readonly` in the logon trigger so it cannot be changed during the session.

Update your logon trigger as follows:

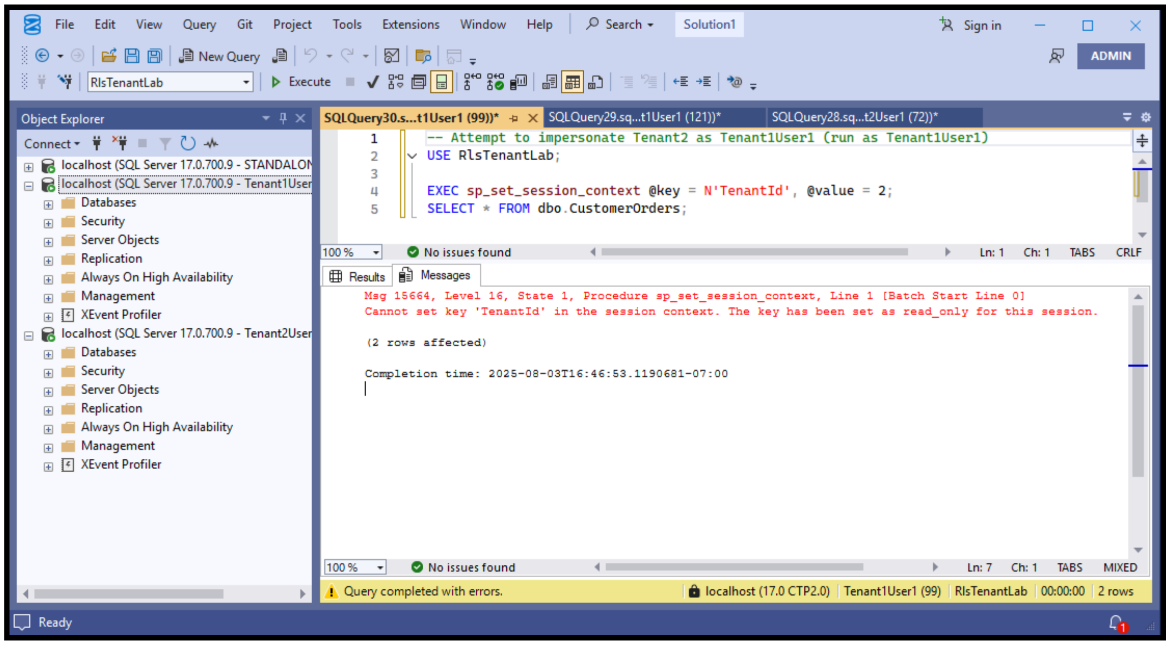
```
CREATE OR ALTER TRIGGER trg_SetTenantContext
ON ALL SERVER
FOR LOGON
AS
BEGIN
    DECLARE @tenantId INT;

    IF ORIGINAL_LOGIN() = 'Tenant1User1'
        SET @tenantId = 1;
    ELSE IF ORIGINAL_LOGIN() = 'Tenant2User1'
        SET @tenantId = 2;

    EXECUTE AS LOGIN = ORIGINAL_LOGIN();
    EXEC sp_set_session_context @key = N'TenantId', @value = @tenantId, @readonly = 1;
    REVERT;
END;
```

This ensures that once the tenant context is set, it cannot be changed during the session.

Now, attempts to override the context will fail:

```
-- Attempt to impersonate Tenant2 as Tenant1User1 (run as Tenant1User1)
EXEC sp_set_session_context @key = N'TenantId', @value = 2;
SELECT * FROM dbo.CustomerOrders;
```

Expected: The session context remains restricted to the login-defined tenant, and the override attempt will result in an error.



```
-- Attempt to impersonate Tenant2 as Tenant1User1 (run as Tenant1User1)
USE RlsTenantLab;

EXEC sp_set_session_context @key = N'TenantId', @value = 2;
SELECT * FROM dbo.CustomerOrders;
```

```
Msg 15664, Level 16, State 1, Procedure sp_set_session_context, Line 1 [Batch Start Line 0]
Cannot set key 'TenantId' in the session context. The key has been set as read_only for this session.

(2 rows affected)

Completion time: 2025-08-03T16:46:53.1190681-07:00
```

Query completed with errors. | localhost (17.0 CTP2.0) | Tenant1User1 (99) | RlsTenantLab | 00:00:00 | 2 rows

Figure 11.5: Error on overriding Tenant 2 to Tenant 1.

11.9.5 Summary

In this exercise, you:

- Created separate users representing tenants.
- Used a logon trigger to automatically assign tenant context.
- Verified that Row-Level Security restricts access correctly based on login identity.

✅ *RLS enforcement is transparent to application users and ensures secure, tenant-isolated access to shared data in SQL Server.*

11.10 Exercise 29: Audit RLS Access and Log Session Context Activity

11.10.1 Description

After implementing Row-Level Security (RLS), it’s important to verify and monitor how users access sensitive or tenant-specific rows. In this lab, you will configure a mechanism to log every access attempt to a sensitive table by capturing the session context. This simulates an audit log to review who accessed which data—useful for compliance, debugging, or multi-tenant systems.

11.10.2 Objectives

By the end of this hands-on lab, you will be able to:

- Create a logging table to capture RLS access
- Use SQL Server’s `SESSION_CONTEXT` for tenant/user tracking
- Implement triggers to log access to RLS-protected tables
- Query and interpret the access logs

11.10.3 Prerequisites

- SQL Server 2025
- SQL Server Management Studio (SSMS) 21.x
- A new dedicated database for this exercise (to avoid affecting production or shared schemas)

11.10.4 Steps

Here’s a step-by-step guide to complete this exercise:

11.10.4.1 Step 1: Create a New Database

Let’s begin by creating a clean environment for testing and logging.

```
CREATE DATABASE RLS_AuditLab;
GO
USE RLS_AuditLab;
GO
```

✅ This isolates our test data and ensures no interference with other exercises.

11.10.4.2 Step 2: Create Tenant Data Table and Sample Users

Create a basic `Invoices` table with a `TenantId`.

```
CREATE TABLE Invoices (
    InvoiceId INT PRIMARY KEY IDENTITY,
    TenantId INT,
    CustomerName NVARCHAR(100),
    Amount MONEY
);
GO

INSERT INTO Invoices (TenantId, CustomerName, Amount)
VALUES (1, 'Cecep', 1200.00),
       (1, 'Bambang', 890.00),
       (2, 'Ita', 455.50),
       (2, 'Diana', 310.75);
GO
```

✅ This simulates tenant-partitioned data.

11.10.4.3 Step 3: Add Security Predicate for RLS

Define a function and security policy for tenant isolation.

```
CREATE FUNCTION fn_tenant_filter(@TenantId INT)
RETURNS TABLE
WITH SCHEMABINDING
AS
RETURN SELECT 1 AS result
WHERE @TenantId = CAST(SESSION_CONTEXT(N'TenantId') AS INT);
GO

CREATE SECURITY POLICY InvoiceTenantFilter
ADD FILTER PREDICATE dbo.fn_tenant_filter(TenantId)
ON dbo.Invoices
WITH (STATE = ON);
GO
```

✅ This ensures users can only see data matching their session’s tenant ID.

11.10.4.4 Step 4: Create a Logging Table

Now create a table to capture access attempts.

```
CREATE TABLE InvoiceAccessLog (
    LogId INT IDENTITY PRIMARY KEY,
    TenantId INT,
    Username NVARCHAR(100),
    AccessedAt DATETIME2 DEFAULT SYSUTCDATETIME(),
    QueryType NVARCHAR(10)
);
GO
```

✅ This will be used by a trigger to log activity transparently.

11.10.4.5 Step 5: Create AFTER INSERT Trigger

We'll simulate access logging by capturing `INSERT` operations.

```
-- SQL Server does not support SELECT triggers; use an AFTER INSERT trigger for demonstrat
CREATE TRIGGER trg_log_invoice_access
ON Invoices
AFTER INSERT
AS
BEGIN
    DECLARE @tenantId INT = CAST(SESSION_CONTEXT(N'TenantId') AS INT);
    DECLARE @username NVARCHAR(100) = SYSTEM_USER;

    INSERT INTO InvoiceAccessLog (TenantId, Username, QueryType)
    SELECT DISTINCT @tenantId, @username, 'INSERT'
    FROM inserted;
END;
GO
```

✅ This will log all SELECTs made to the table.

11.10.4.6 Step 6: Simulate Tenant Access

Let's simulate different users accessing the system.

```
-- Simulate access as Tenant 1
EXEC sp_set_session_context 'TenantId', 1;
INSERT INTO Invoices (TenantId, CustomerName, Amount) VALUES (1, 'Tenant1 Customer', 100.0

-- Simulate access as Tenant 2
EXEC sp_set_session_context 'TenantId', 2;
INSERT INTO Invoices (TenantId, CustomerName, Amount) VALUES (2, 'Tenant2 Customer', 200.0
```

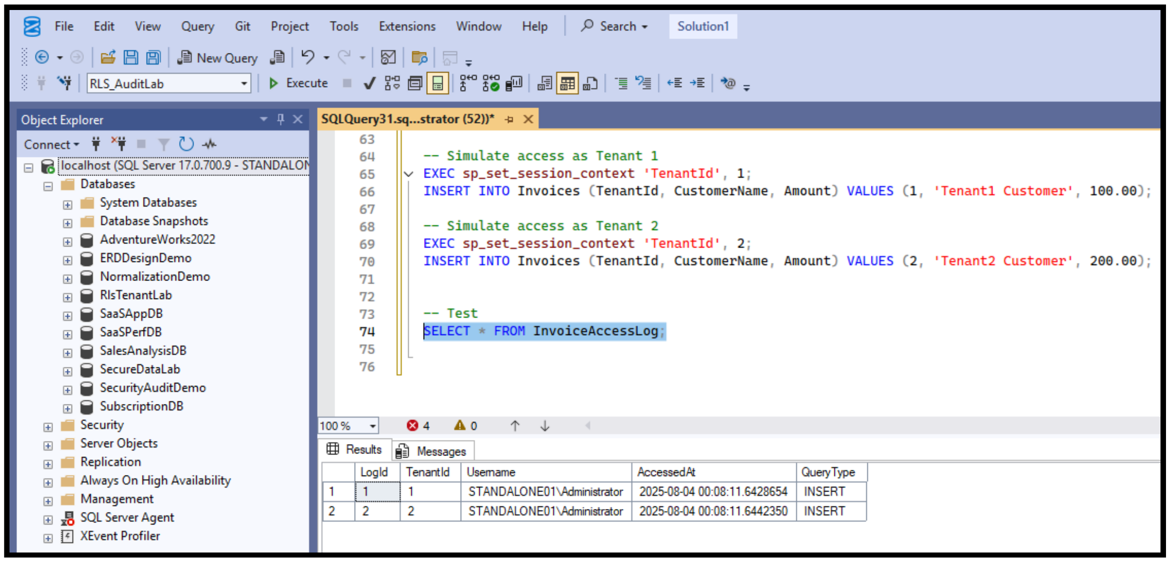
✅ Each access should insert a record into `InvoiceAccessLog`.

11.10.4.7 Step 7: Review Audit Logs

Now review the contents of the audit table.

```
SELECT * FROM InvoiceAccessLog;
```

✅ You should see which tenant accessed the data, at what time, and what operation was performed.



```
-- Simulate access as Tenant 1
EXEC sp_set_session_context 'TenantId', 1;
INSERT INTO Invoices (TenantId, CustomerName, Amount) VALUES (1, 'Tenant1 Customer', 100.00);

-- Simulate access as Tenant 2
EXEC sp_set_session_context 'TenantId', 2;
INSERT INTO Invoices (TenantId, CustomerName, Amount) VALUES (2, 'Tenant2 Customer', 200.00);


-- Test
SELECT * FROM InvoiceAccessLog;
```

	LogId	TenantId	Username	AccessedAt	QueryType
1	1	1	STANDALONE01\Administrator	2025-08-04 00:08:11.6428654	INSERT
2	2	2	STANDALONE01\Administrator	2025-08-04 00:08:11.6442350	INSERT

Figure 11.6: Showing audit log from InvoiceAccessLog table.

11.10.5 Summary

In this exercise, you:

- Created an RLS-protected table using `SESSION_CONTEXT`
- Built an audit mechanism to capture access events
- Simulated tenants accessing their own data
- Logged and reviewed access history using a custom log table

✅ This approach enhances observability and compliance, particularly for multi-tenant or regulated systems.

11.11 Conclusion

In this chapter, you learned how to implement Row-Level Security (RLS) in SQL Server 2025 to enforce tenant isolation and fine-grained access control. You explored the architecture of RLS, created security predicates, and applied them through security policies. Additionally, you practiced auditing access to ensure compliance and security in multi-tenant applications.

12 Masking, Encryption, and Auditing

In today's regulatory environment, safeguarding sensitive business data—such as personal identifiers, financials, and healthcare records—is not optional. This chapter explores key SQL Server 2025 features for **data privacy and protection**, including **Dynamic Data Masking (DDM)**, **encryption options**, and **auditing access** to sYou'll create a table with sensitive data, then use the **Always Encrypted Wizard** in SSMS 21.x to encrypt existing columns. This approach is more practical and user-friendly than manual key management.nsitive data.

12.1 Dynamic Data Masking (DDM)

Dynamic Data Masking limits sensitive data exposure by **masking it in query results**, while keeping it unchanged in storage.

> *Ideal for restricting sensitive columns (e.g., SSNs, emails) from non-privileged users.*

Here's how to apply DDM to a column:

```sql
ALTER TABLE Employees
ALTER COLUMN Email ADD MASKED WITH (FUNCTION = 'email()');
```

The following functions are commonly used for masking:

Function	Example Column	Masked Output
`default()`	`Salary` → 0	Numeric/string fallback
`email()`	`jane@corp.com` → `jXXX@XXXX.com`	
`partial()`	`partial(1,"XXXX",1)` → shows start/end only	
`random()`	For numeric data: masks with random values	

We can create a table with masked columns like this:

```
CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY,
    FullName NVARCHAR(100),
    Email NVARCHAR(100) MASKED WITH (FUNCTION = 'email()'),
    CreditCard CHAR(16) MASKED WITH (FUNCTION = 'partial(4,"XXXX-XXXX-XXXX-",4)')
);
```

*Masking is **applied at query time** for users without `UNMASK` permission.*

To allow a user to see masked data, you can grant them the `UNMASK` permission:

```
-- Allow access to see original data
GRANT UNMASK TO auditor_user;

-- Revoke access
REVOKE UNMASK FROM analyst_user;
```

Use DDM to protect sensitive data while allowing users to work with the data they need without exposing sensitive information.

12.2 Encryption Options

SQL Server supports multiple encryption models for securing data at rest, in transit, and in use.

12.2.1 Transparent Data Encryption (TDE)

Encrypts the **entire database** at the file level, including backups and transaction logs.

We can enable TDE with the following steps:

```
-- Step 1: Create a master key
CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'YourStrongPassword!';

-- Step 2: Create certificate
CREATE CERTIFICATE MyTDECert WITH SUBJECT = 'TDE Cert';

-- Step 3: Create encryption key and enable TDE
CREATE DATABASE ENCRYPTION KEY
WITH ALGORITHM = AES_256
ENCRYPTION BY SERVER CERTIFICATE MyTDECert;

ALTER DATABASE SalesDB SET ENCRYPTION ON;
```

These commands create a master key, a certificate, and then enable TDE on the `SalesDB` database.

12.2.2 Always Encrypted

Encrypts specific **columns** at the client-side, so data is never visible in plaintext to SQL Server.

> *Ideal for highly sensitive values like SSNs, credit cards.*

- Requires client drivers and key management.
- Columns must be defined as **encrypted** during creation.

12.2.3 Cell-Level Encryption (CLE)

Encrypts **individual column values** using functions like `EncryptByKey` and `DecryptByKey`.

Here’s how to encrypt a credit card number:

```
-- Create symmetric key
CREATE SYMMETRIC KEY CreditCardKey
WITH ALGORITHM = AES_256
ENCRYPTION BY PASSWORD = 'KeyPassword123!';

-- Open and use the key
OPEN SYMMETRIC KEY CreditCardKey DECRYPTION BY PASSWORD = 'KeyPassword123!';

SELECT
    CONVERT(varchar, DecryptByKey(EncryptedCreditCard)) AS CreditCard
FROM Customers;

-- Close key
CLOSE SYMMETRIC KEY CreditCardKey;
```

12.3 Auditing Access to Sensitive Data

SQL Server provides native tools to **track and log access** to sensitive tables, columns, or actions.

SQL Server Audit allows you to create server-level and database-level audits to track specific actions, such as:

- SELECT on sensitive columns
- INSERT/UPDATE on critical tables
- Login/logout events

You can create an audit and then define a database audit specification to capture specific actions:

```
-- Create server audit
CREATE SERVER AUDIT Audit_SensitiveData
TO FILE (FILEPATH = 'C:\AuditLogs\', MAXSIZE = 10 MB);

-- Enable the audit
ALTER SERVER AUDIT Audit_SensitiveData WITH (STATE = ON);

-- Create database audit specification
CREATE DATABASE AUDIT SPECIFICATION Audit_PII_Reads
FOR SERVER AUDIT Audit_SensitiveData
ADD (SELECT ON OBJECT::dbo.Customers BY public);

-- Enable it
ALTER DATABASE AUDIT SPECIFICATION Audit_PII_Reads WITH (STATE = ON);
```

This setup captures all `SELECT` operations on the `Customers` table and logs them to the specified audit file.

> *You can also filter by user roles or specific actions.*

You can query the audit logs using the `sys.fn_get_audit_file` function:

```
SELECT *
FROM sys.fn_get_audit_file('C:\AuditLogs\*.sqlaudit', DEFAULT, DEFAULT);
```

12.4 Best Practices for Data Protection

The following best practices can help ensure effective data protection in SQL Server:

Practice	**Why It Matters**
Use DDM for casual data hiding	Easy to implement, low impact
Use Always Encrypted for PII/PHI	Data never exposed to DB or admins
Use TDE for broad protection	Transparent protection of entire database

Practice	Why It Matters
Limit `UNMASK` and `CONTROL` grants	Enforce least privilege
Enable auditing on sensitive objects	For compliance and breach detection

12.5 Exercise 30: Mask Email and Phone Fields in Query Output

12.5.1 Description

In this exercise, you will explore **Dynamic Data Masking (DDM)**, a feature in SQL Server that helps protect sensitive information such as email addresses and phone numbers by obfuscating data in query results. This is especially useful in scenarios where users should not see full personal data—like junior analysts or customer support staff.

You'll define masking rules directly in the schema, control who sees unmasked data, and test the impact through different users.

12.5.2 Objectives

By the end of this lab, you will be able to:

- Create a table with sensitive columns.
- Apply built-in DDM functions (`email`, `partial`, `default`) to mask data.
- Verify masked output for standard users.
- Grant unmask privileges to view actual data.

12.5.3 Prerequisites

- SQL Server 2025 instance with SSMS 21.x.
- Permission to create databases, users, and apply masking rules.

12.5.4 Steps

Here's a step-by-step guide to complete this exercise:

12.5.4.1 Step 1: Create a New Database for the Lab

Let’s begin by creating a new test database called `DataMaskingLab`.

```
CREATE DATABASE DataMaskingLab;
GO

USE DataMaskingLab;
GO
```

📌 *This database will hold our customer table and test users.*

12.5.4.2 Step 2: Create a Table with Sensitive Fields

Now define a `Customers` table with columns for name, email, and phone. We’ll apply masks to email and phone fields using DDM.

```
CREATE TABLE Customers (
    CustomerId INT IDENTITY PRIMARY KEY,
    FullName NVARCHAR(100),
    Email NVARCHAR(100) MASKED WITH (FUNCTION = 'email()'),
    PhoneNumber NVARCHAR(20) MASKED WITH (FUNCTION = 'partial(2,"XXXXXXX",2)')
);
```

📌 *The `email()` mask hides the username part; `partial()` keeps the first and last 2 characters visible in phone numbers.*

12.5.4.3 Step 3: Insert Sample Data

Next, insert several rows with full data.

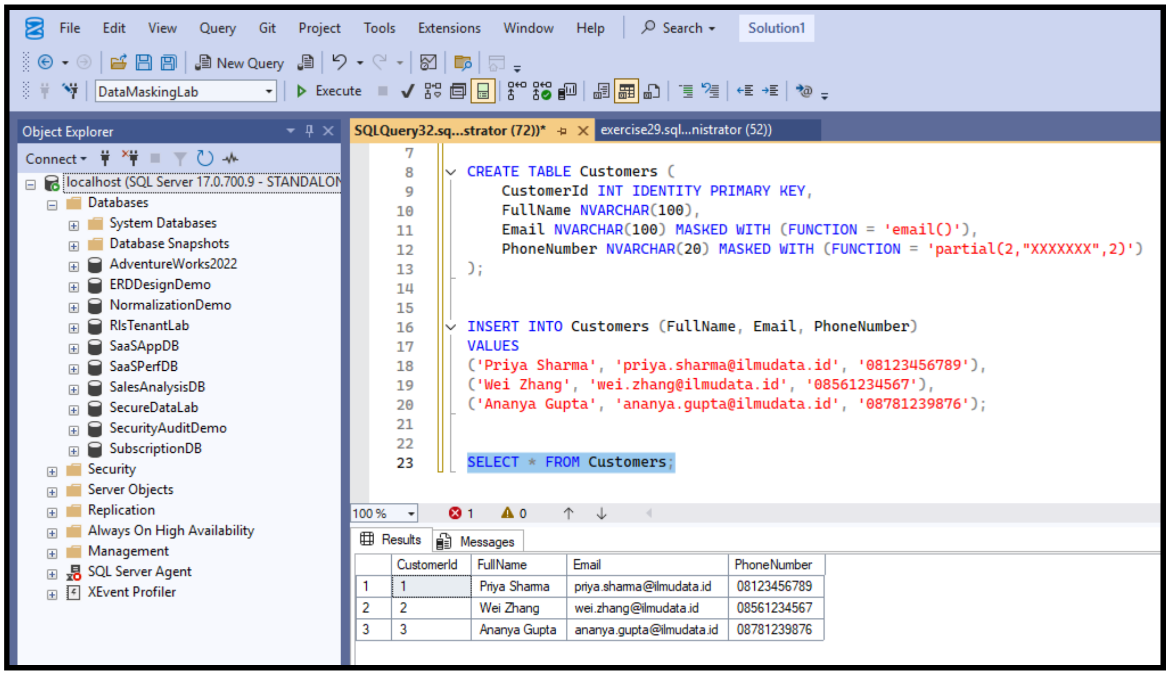
```
INSERT INTO Customers (FullName, Email, PhoneNumber)
VALUES
('Priya Sharma', 'priya.sharma@ilmudata.id', '08123456789'),
('Wei Zhang', 'wei.zhang@ilmudata.id', '08561234567'),
('Ananya Gupta', 'ananya.gupta@ilmudata.id', '08781239876');
```

12.5.4.4 Step 4: Query the Data as Admin

Check the data from your current admin context. You’ll see the full values.

```
SELECT * FROM Customers;
```

Expected result as shown below:



```sql
CREATE TABLE Customers (
    CustomerId INT IDENTITY PRIMARY KEY,
    FullName NVARCHAR(100),
    Email NVARCHAR(100) MASKED WITH (FUNCTION = 'email()'),
    PhoneNumber NVARCHAR(20) MASKED WITH (FUNCTION = 'partial(2,"XXXXXXX",2)')
);


INSERT INTO Customers (FullName, Email, PhoneNumber)
VALUES
('Priya Sharma', 'priya.sharma@ilmudata.id', '08123456789'),
('Wei Zhang', 'wei.zhang@ilmudata.id', '08561234567'),
('Ananya Gupta', 'ananya.gupta@ilmudata.id', '08781239876');


SELECT * FROM Customers;
```

	CustomerId	FullName	Email	PhoneNumber
1	1	Priya Sharma	priya.sharma@ilmudata.id	08123456789
2	2	Wei Zhang	wei.zhang@ilmudata.id	08561234567
3	3	Ananya Gupta	ananya.gupta@ilmudata.id	08781239876

Figure 12.1: Showing data from Customers table.

12.5.4.5 Step 5: Create a Limited Access User

Create a new login and user that will query the data without unmasking rights.

```sql
CREATE LOGIN ReadOnlyUser WITH PASSWORD = 'ReadOnlyPass!';
CREATE USER ReadOnlyUser FOR LOGIN ReadOnlyUser;
GRANT SELECT ON Customers TO ReadOnlyUser;
```

📌 *The new user can query the table, but sensitive fields should be masked.*

12.5.4.6 Step 6: Open a New Session as ReadOnlyUser

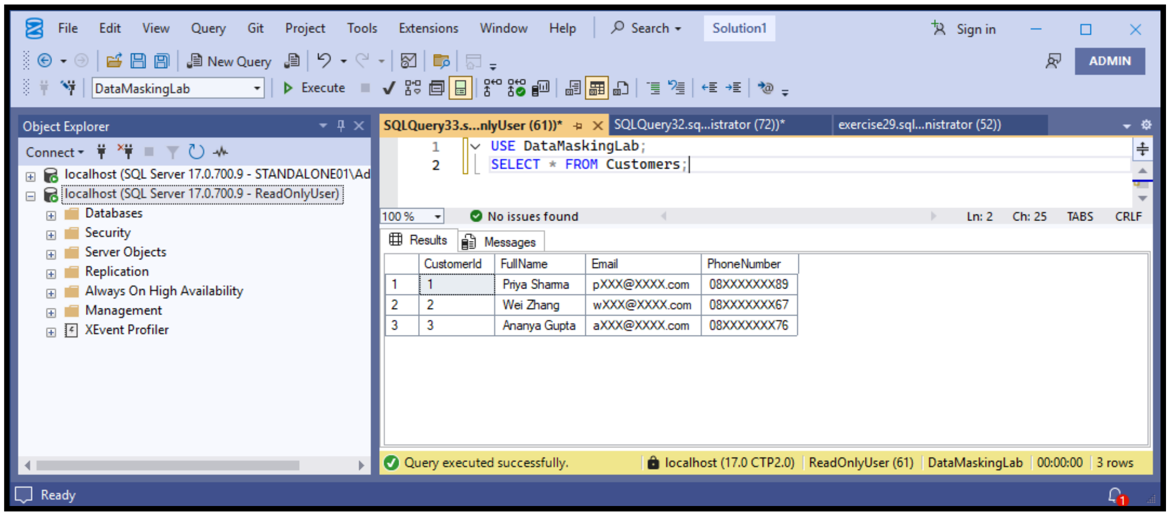
In SSMS, open a new query window using login:

- **Username:** `ReadOnlyUser`
- **Password:** `ReadOnlyPass!`

Run:

```sql
USE DataMaskingLab;
SELECT * FROM Customers;
```

Expected masked output as shown below:



```
USE DataMaskingLab;
SELECT * FROM Customers;
```

	CustomerId	FullName	Email	PhoneNumber
1	1	Priya Sharma	pXXX@XXXX.com	08XXXXXXX89
2	2	Wei Zhang	wXXX@XXXX.com	08XXXXXXX67
3	3	Ananya Gupta	aXXX@XXXX.com	08XXXXXXX76

Query executed successfully. | localhost (17.0 CTP2.0) | ReadOnlyUser (61) | DataMaskingLab | 00:00:00 | 3 rows

Figure 12.2: Showing masking data.

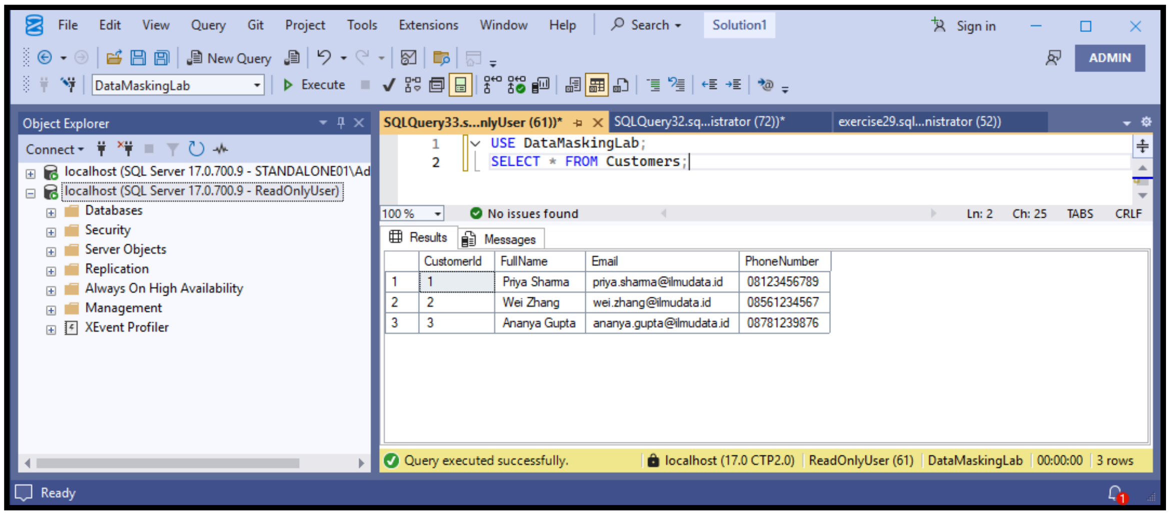
12.5.4.7 Step 7: Grant UNMASK Permission

Return to the admin session and grant the user permission to see full data.

```
GRANT UNMASK TO ReadOnlyUser;
```

Re-run the query in the other window logged in as `ReadOnlyUser`.

Now the user will see **unmasked data**, same as the admin.



```
USE DataMaskingLab;
SELECT * FROM Customers;
```

	CustomerId	FullName	Email	PhoneNumber
1	1	Priya Sharma	priya.sharma@ilmudata.id	08123456789
2	2	Wei Zhang	wei.zhang@ilmudata.id	08561234567
3	3	Ananya Gupta	ananya.gupta@ilmudata.id	08781239876

Query executed successfully. | localhost (17.0 CTP2.0) | ReadOnlyUser (61) | DataMaskingLab | 00:00:00 | 3 rows

Figure 12.3: Showing unmasked data from Customers table.

12.5.5 Summary

In this hands-on lab, you:

- Created a table with masked email and phone fields.
- Used Dynamic Data Masking (DDM) to obscure output without changing the data.
- Simulated masked and unmasked access for different users.

✅ *DDM allows for secure access control with minimal application logic and is perfect for compliance with data privacy regulations.*

12.6 Exercise 31: Encrypt Sensitive Data Using Always Encrypted

12.6.1 Description

This exercise introduces **Always Encrypted**, a feature in SQL Server that protects sensitive data like credit card numbers or national IDs *at rest, in transit, and in use*. The encryption and decryption are handled on the client side—so SQL Server itself can't view the plaintext data.

You'll create a table with sensitive data, then use the **Always Encrypted Wizard** in SSMS 21.x to encrypt existing columns. This approach is more practical and user-friendly than manual key management.

12.6.2 Objectives

By the end of this lab, you will:

- Create a table with sensitive data
- Use the Always Encrypted Wizard in SSMS 21.x to encrypt columns
- Configure Column Master Key (CMK) and Column Encryption Key (CEK) automatically
- Test querying encrypted data with column encryption enabled/disabled
- Understand limitations when querying encrypted columns

12.6.3 Prerequisites

- SQL Server 2019 or later and SSMS 21.x (Always Encrypted UI support required)
- Windows environment (for Windows Certificate Store)
- Administrative privileges on the local machine

12.6.4 Steps

Here’s a step-by-step guide to complete this exercise:

12.6.4.1 Step 1: Create a New Database and Table

Start by creating a new lab database and a table with sensitive data:

```
CREATE DATABASE AlwaysEncryptedLab;
GO
USE AlwaysEncryptedLab;
GO

-- Create table with sensitive data (unencrypted initially)
CREATE TABLE Customers (
    CustomerId INT IDENTITY PRIMARY KEY,
    FullName NVARCHAR(100),
    Email NVARCHAR(100),
    NationalID NVARCHAR(20),
    CreditCardNumber NVARCHAR(20),
    Salary DECIMAL(10,2)
);
```

12.6.4.2 Step 2: Insert Sample Data

Insert some test data that we’ll encrypt later:

```
INSERT INTO Customers (FullName, Email, NationalID, CreditCardNumber, Salary)
VALUES
('Thariq Akbar', 'thariq.akbar@ilmudata.id', '1234567890123456', '4532-1234-5678-9012', 75
('Zahra Zhafirah', 'zahra.zhafirah@ilmudata.id', '9876543210987654', '5678-9012-3456-7890',
('Ananda Putra', 'ananda.putra@ilmudata.id', '5555666677778888', '1234-5678-9012-3456', 95
```

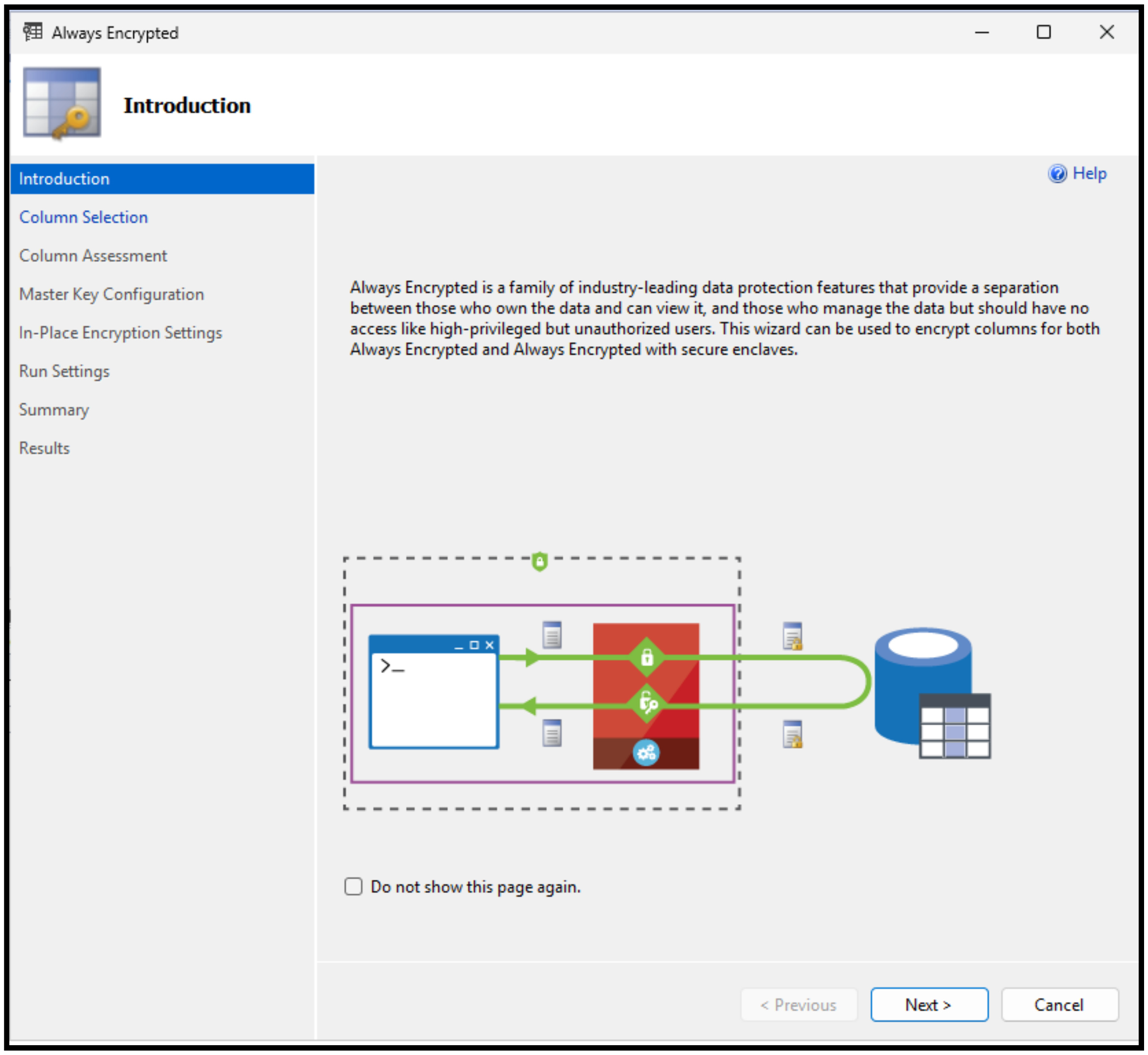
Verify the data was inserted:

```
SELECT * FROM Customers;
```

You should see the full values in the `NationalID` and `CreditCardNumber` columns.

12.6.4.3 Step 3: Launch the Always Encrypted Wizard

1. In **Object Explorer**, expand `AlwaysEncryptedLab` database so that you can see the tables
2. Right-click on the table name `Customers` and select menu `Always Encrypted Wizard`
3. The **Always Encrypted Wizard** will open



Always Encrypted is a family of industry-leading data protection features that provide a separation between those who own the data and can view it, and those who manage the data but should have no access like high-privileged but unauthorized users. This wizard can be used to encrypt columns for both Always Encrypted and Always Encrypted with secure enclaves.

☐ Do not show this page again.

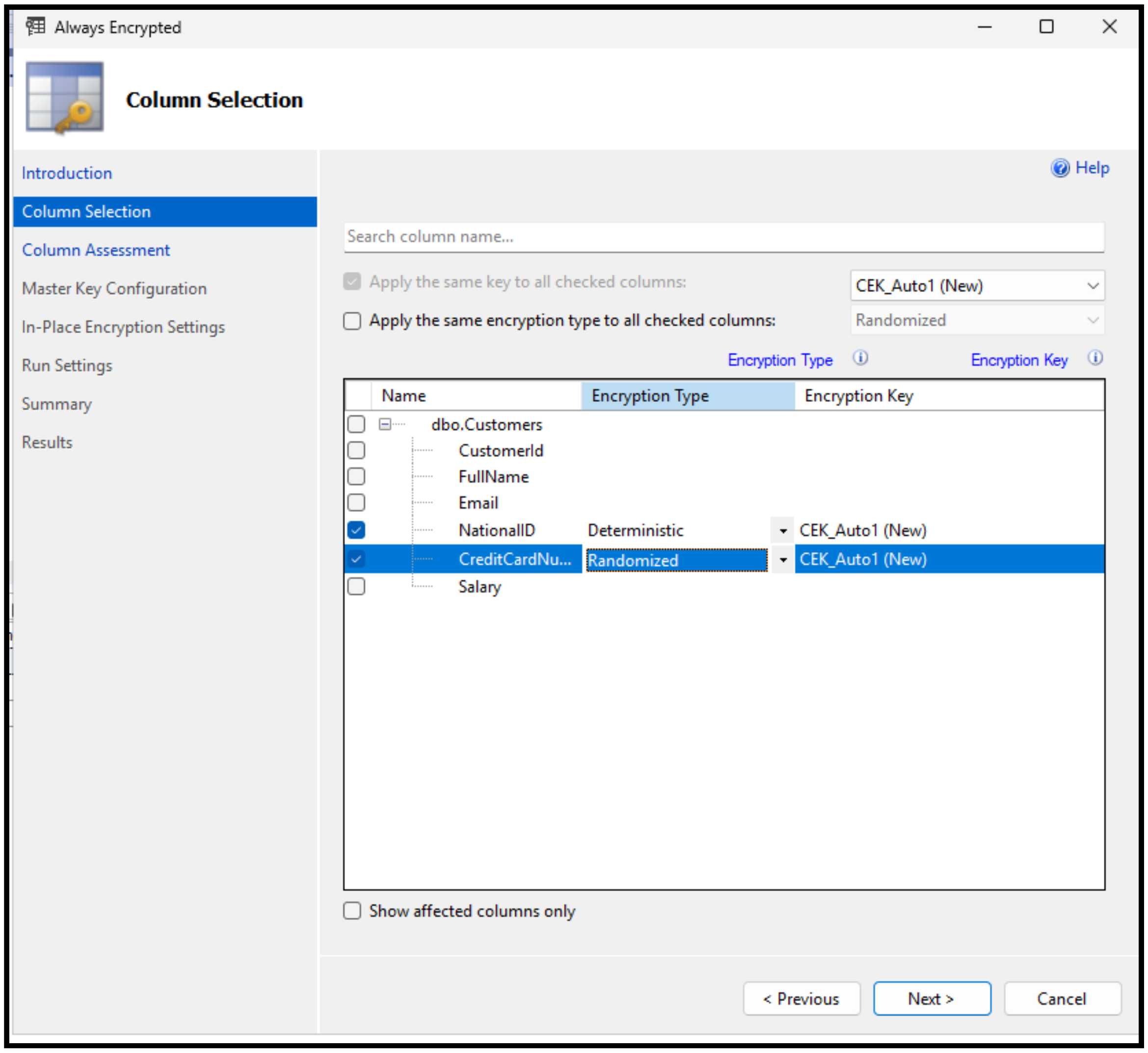
Figure 12.4: Always Encrypted Wizard.

📌 *This wizard will guide you through encrypting existing columns without manual key management.*

12.6.4.4 Step 4: Select Columns to Encrypt

In the **Column Selection** page:

1. **Select the table**: `dbo.Customers`
2. **Choose columns to encrypt**:
   - `NationalID`: Select **Deterministic** encryption (allows equality searches)
   - `CreditCardNumber`: Select **Randomized** encryption (highest security)
3. Click **Next**



Apply the same key to all checked columns: CEK_Auto1 (New)
Apply the same encryption type to all checked columns: Randomized

	Name	Encryption Type	Encryption Key
☐	dbo.Customers		
☐	CustomerId		
☐	FullName		
☐	Email		
✓	NationalID	Deterministic	CEK_Auto1 (New)
✓	CreditCardNu...	Randomized	CEK_Auto1 (New)
☐	Salary		

☐ Show affected columns only

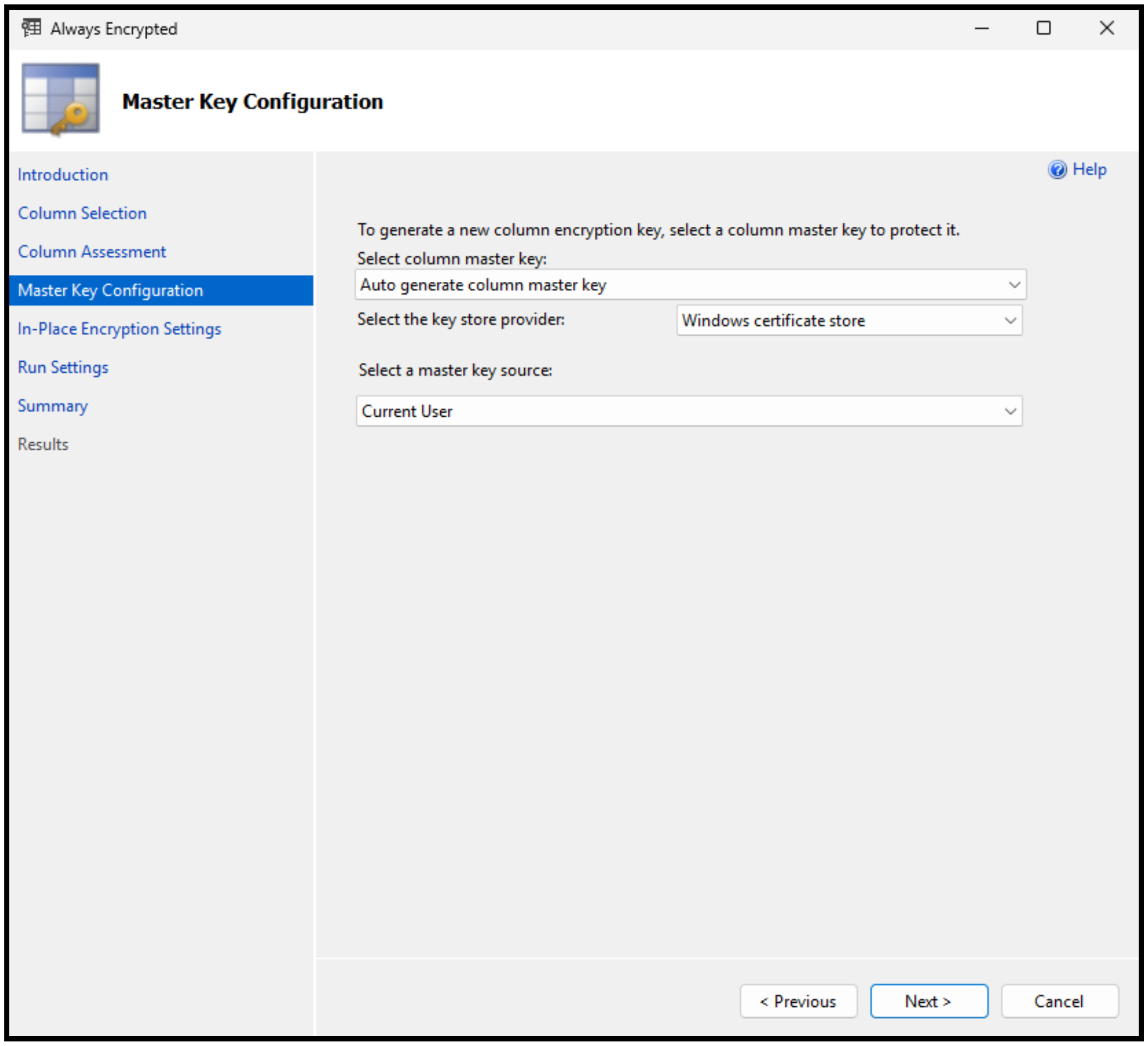
Figure 12.5: Configuring columns.

📌 *Deterministic encryption allows WHERE clauses with equality. Randomized provides better security but limits querying.*

12.6.4.5 Step 5: Configure Master Key

In the **Master Key Configuration** page:

1. **Key Store**: Select **Windows Certificate Store**
2. **Auto generate column master key**: select this option
3. Select **Current User** for a master key source
4. Click **Next**



To generate a new column encryption key, select a column master key to protect it.

Select column master key: Auto generate column master key

Select the key store provider: Windows certificate store

Select a master key source: Current User

Figure 12.6: Configuring master key.

📌 *The wizard will automatically create a certificate in your Windows Certificate Store.*

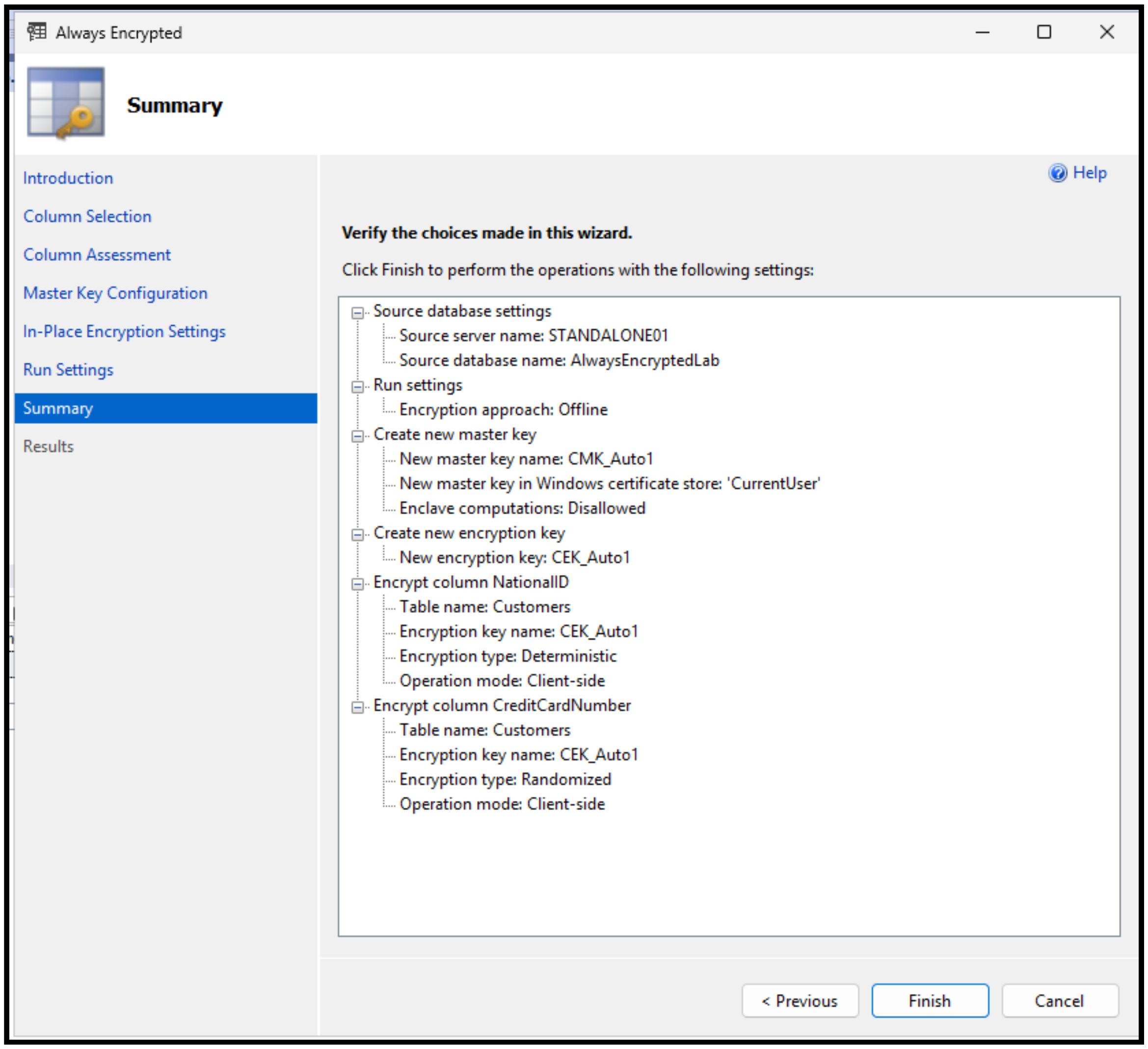
12.6.4.6 Step 6: Run Settings

1. Select **Offline (Default)** for the encryption process
2. Click **Next** to confirm the settings

12.6.4.7 Step 7: Validation and Summary

1. **Validation page**: The wizard checks prerequisites
2. **Summary page**: Review the encryption settings:
   - Column Master Key will be created

    - Column Encryption Key will be created
    - Selected columns will be encrypted
3. Click **Finish** to start the encryption process



Verify the choices made in this wizard.

Click Finish to perform the operations with the following settings:

- Source database settings
  - Source server name: STANDALONE01
  - Source database name: AlwaysEncryptedLab
- Run settings
  - Encryption approach: Offline
- Create new master key
  - New master key name: CMK_Auto1
  - New master key in Windows certificate store: 'CurrentUser'
  - Enclave computations: Disallowed
- Create new encryption key
  - New encryption key: CEK_Auto1
- Encrypt column NationalID
  - Table name: Customers
  - Encryption key name: CEK_Auto1
  - Encryption type: Deterministic
  - Operation mode: Client-side
- Encrypt column CreditCardNumber
  - Table name: Customers
  - Encryption key name: CEK_Auto1
  - Encryption type: Randomized
  - Operation mode: Client-side

Figure 12.7: Setting confirmation.

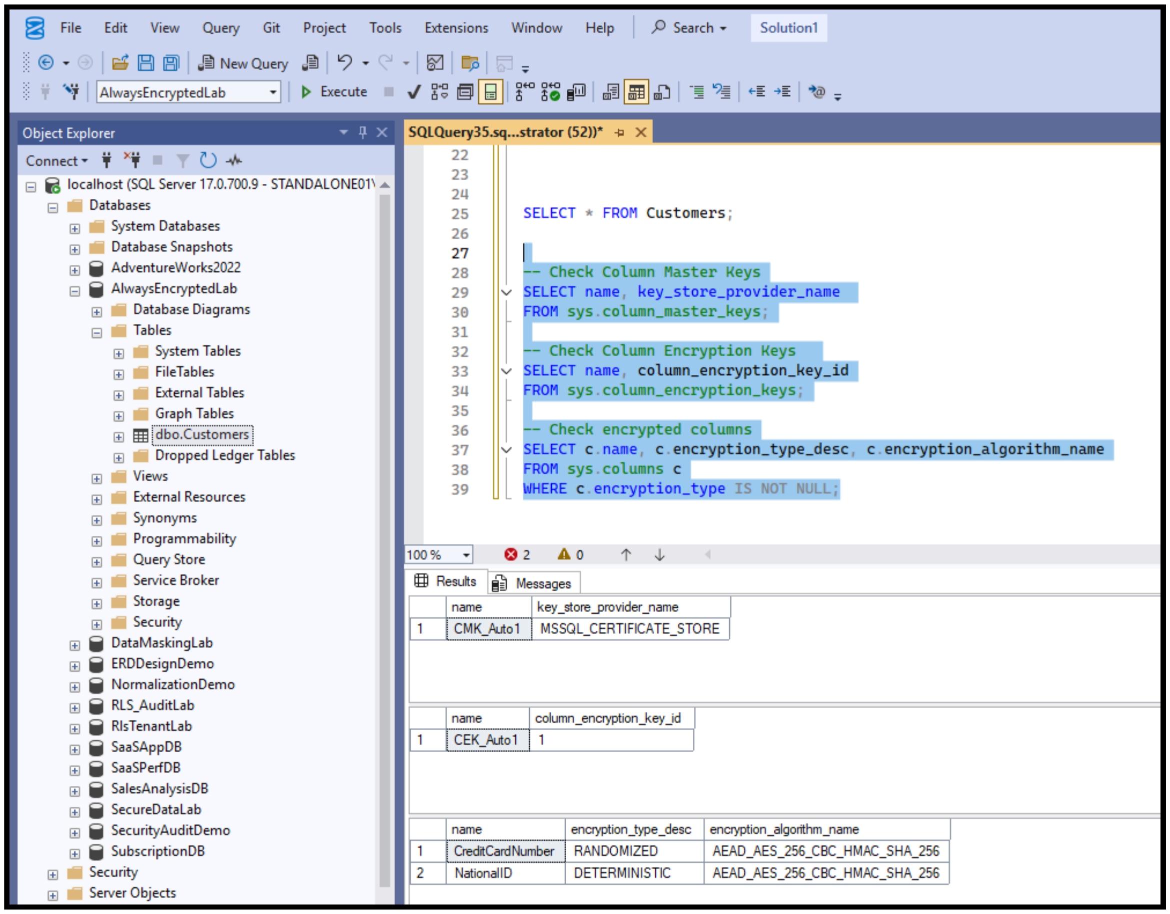
📌 *This process may take several minutes as it encrypts existing data.*

12.6.4.8 Step 8: Verify Encryption Setup

After the wizard completes, verify the encryption objects were created:

```
-- Check Column Master Keys
SELECT name, key_store_provider_name
FROM sys.column_master_keys;

-- Check Column Encryption Keys  ex
```

```sql
SELECT name, column_encryption_key_id
FROM sys.column_encryption_keys;

-- Check encrypted columns
SELECT c.name, c.encryption_type_desc, c.encryption_algorithm_name
FROM sys.columns c
WHERE c.encryption_type IS NOT NULL;
```



```sql
SELECT * FROM Customers;

-- Check Column Master Keys
SELECT name, key_store_provider_name
FROM sys.column_master_keys;

-- Check Column Encryption Keys
SELECT name, column_encryption_key_id
FROM sys.column_encryption_keys;

-- Check encrypted columns
SELECT c.name, c.encryption_type_desc, c.encryption_algorithm_name
FROM sys.columns c
WHERE c.encryption_type IS NOT NULL;
```

	name	key_store_provider_name
1	CMK_Auto1	MSSQL_CERTIFICATE_STORE

	name	column_encryption_key_id
1	CEK_Auto1	1

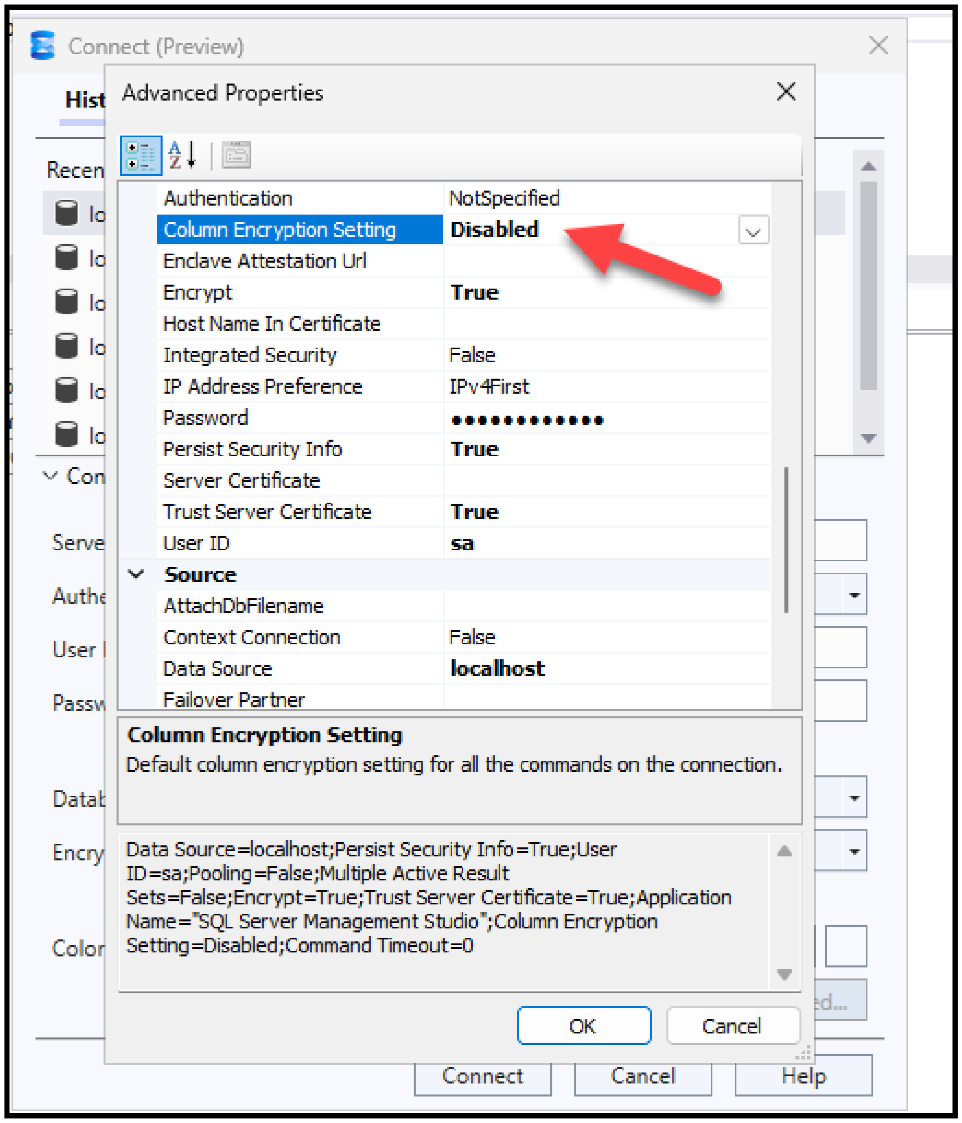
	name	encryption_type_desc	encryption_algorithm_name
1	CreditCardNumber	RANDOMIZED	AEAD_AES_256_CBC_HMAC_SHA_256
2	NationalID	DETERMINISTIC	AEAD_AES_256_CBC_HMAC_SHA_256

Figure 12.8: Verifying the encryption objects.

12.6.4.9 Step 8: Test Querying with Column Encryption Disabled

To test viewing encrypted data as binary values, we need to create a new connection with Always Encrypted disabled:

1. In **Object Explorer**, click **Connect** → **Database Engine**
2. Enter your server details
3. Click **Advanced** button
4. Select **Disabled** on column encryption settings



Connect (Preview)

Advanced Properties

Authentication	NotSpecified
Column Encryption Setting	**Disabled**
Enclave Attestation Url	
Encrypt	**True**
Host Name In Certificate	
Integrated Security	False
IP Address Preference	IPv4First
Password	●●●●●●●●●●●●
Persist Security Info	**True**
Server Certificate	
Trust Server Certificate	**True**
User ID	**sa**
Source	
AttachDbFilename	
Context Connection	False
Data Source	**localhost**
Failover Partner	

Column Encryption Setting
Default column encryption setting for all the commands on the connection.

```
Data Source=localhost;Persist Security Info=True;User
ID=sa;Pooling=False;Multiple Active Result
Sets=False;Encrypt=True;Trust Server Certificate=True;Application
Name="SQL Server Management Studio";Column Encryption
Setting=Disabled;Command Timeout=0
```

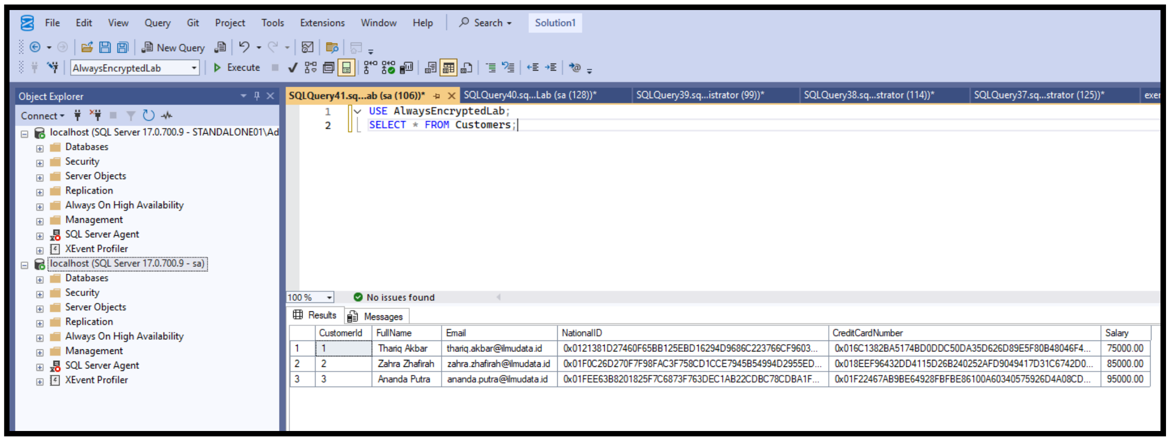
Figure 12.9: Verifying the encryption objects.

5. Click **OK** button

6. Click **Connect**

7. In the new connection, run:

```
USE AlwaysEncryptedLab;
SELECT * FROM Customers;
```



```
USE AlwaysEncryptedLab;
SELECT * FROM Customers;
```

	CustomerId	FullName	Email	NationalID	CreditCardNumber	Salary
1	1	Thariq Akbar	thariq.akbar@ilmudata.id	0x0121381D27460F65BB125EBD16294D9686C223766CF9603...	0x016C1382BA5174BD0DDC50DA35D626D89E5F80B48046F4...	75000.00
2	2	Zahra Zhafirah	zahra.zhafirah@ilmudata.id	0x01F0C26D270F7F98FAC3F758CD1CCE7945B54994D2955ED...	0x018EEF96432DD4115D26B240252AFD9049417D31C6742D0...	85000.00
3	3	Ananda Putra	ananda.putra@ilmudata.id	0x01FEE63B8201825F7C6873F763DEC1AB22CDBC78CDBA1F...	0x01F22467AB9BE64928FBFBE86100A60340575926D4A08CD...	95000.00

Figure 12.10: Showing encrypted data on query.

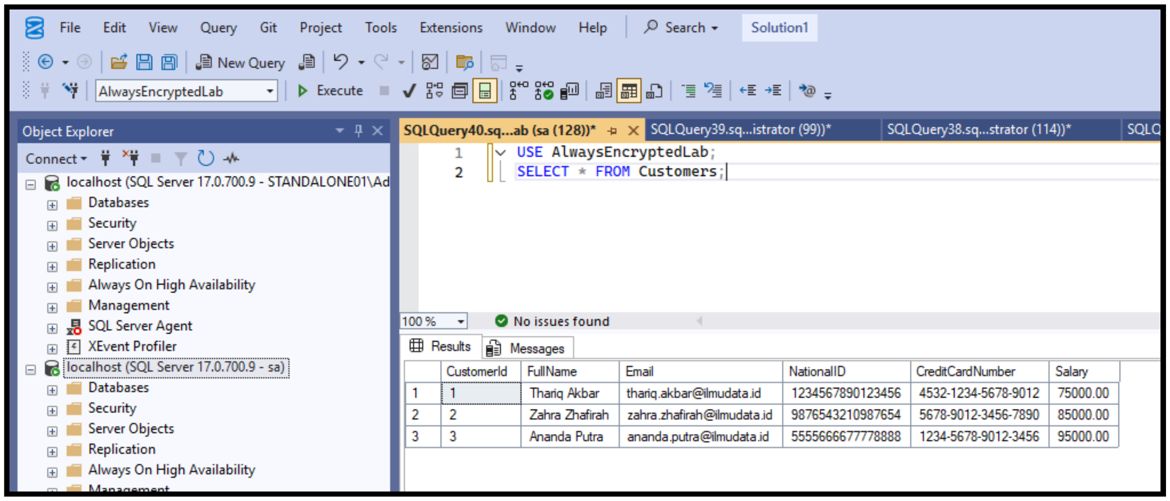
📌 *You should see encrypted binary data in the NationalID and CreditCardNumber columns (showing as hexadecimal values like 0x01A2…).*

12.6.4.10 Step 9: Test Querying with Column Encryption Enabled

Now test with Always Encrypted enabled to see decrypted values:

1. In **Object Explorer**, click **Connect** → **Database Engine**
2. Enter your server details
3. Click **Advanced** button
4. Select **Enabled** on column encryption settings
5. Click **Connect**
6. In this new connection, run:

```
USE AlwaysEncryptedLab;
SELECT * FROM Customers;
```



```
USE AlwaysEncryptedLab;
SELECT * FROM Customers;
```

	CustomerId	FullName	Email	NationalID	CreditCardNumber	Salary
1	1	Thariq Akbar	thariq.akbar@ilmudata.id	1234567890123456	4532-1234-5678-9012	75000.00
2	2	Zahra Zhafirah	zahra.zhafirah@ilmudata.id	9876543210987654	5678-9012-3456-7890	85000.00
3	3	Ananda Putra	ananda.putra@ilmudata.id	5555666677778888	1234-5678-9012-3456	95000.00

Figure 12.11: Showing descrypted data on query.

📌 *You should now see the actual (decrypted) values in all columns, exactly as you inserted them.*

12.6.4.11 Step 10: Test Query Limitations

Try different query types to understand encryption limitations:

✔️ Equality search on deterministic encrypted column (works):

```sql
SELECT * FROM Customers
WHERE NationalID = '1234567890123456';
```

✖️ Pattern search on encrypted column (fails):

```sql
SELECT * FROM Customers
WHERE CreditCardNumber LIKE '4532%'; -- This will give an error
```

✖️ Comparison operators on randomized encrypted column (fails):

```sql
SELECT * FROM Customers
WHERE CreditCardNumber > '4000-0000-0000-0000'; -- This will give an error
```

📌 *Only equality comparisons work with deterministic encryption. No operations work with randomized encryption except retrieval.*

12.6.5 Summary

In this lab, you:

- Created a table with sensitive data using standard SQL
- Used the Always Encrypted Wizard in SSMS 21.x to encrypt existing columns
- Automatically configured Column Master Key (CMK) and Column Encryption Key (CEK)
- Tested querying encrypted data with different connection settings
- Understood the limitations and proper use cases for deterministic vs. randomized encryption

✅ *This wizard-based approach is much more practical for real-world implementations than manual key management, making Always Encrypted accessible for protecting sensitive data in production environments.*

12.7 Exercise 32: Enable and Configure an Audit Policy

12.7.1 Description

This exercise focuses on using **SQL Server Audit**, a built-in feature for tracking and logging server and database-level activity. You'll create an audit specification that captures **SELECT** operations on a sensitive table, configure it to write logs to a file, and then test it by querying the data. This is critical for **compliance** and **data access monitoring** in regulated environments (e.g., GDPR, HIPAA).

12.7.2 Objectives

By the end of this lab, you will:

- Create a server audit and database audit specification
- Configure SQL Server to store audit logs in a file
- Monitor SELECT statements executed on sensitive data
- Review audit log content via T-SQL

12.7.3 Prerequisites

- SQL Server 2025 instance with **FILESTREAM** or file system access
- SSMS 21.x with administrative privileges
- Use `AlwaysEncryptedLab` database from the previous exercise or any test database with a sensitive table

12.7.4 Steps

Here's a step-by-step guide to complete this exercise:

12.7.4.1 Step 1: Create an Audit Object at the Server Level

Start by creating a file-based audit log on your local system.

```
CREATE SERVER AUDIT Audit_Read_Access
TO FILE (
    FILEPATH = 'C:\SQLAuditLogs\',
    MAXSIZE = 10 MB,
    MAX_FILES = 10,
    RESERVE_DISK_SPACE = OFF
)
WITH (
    QUEUE_DELAY = 1000,
    ON_FAILURE = CONTINUE
);
```

📌 *Make sure* `C:\SQLAuditLogs\` *exists and SQL Server has permission to write to it.*

Then enable the audit:

```
ALTER SERVER AUDIT Audit_Read_Access
WITH (STATE = ON);
```

12.7.4.2 Step 2: Create a Database Audit Specification

Switch to the target database and create a specification to track **SELECT** operations on a sensitive table.

```
USE AlwaysEncryptedLab;
GO

CREATE DATABASE AUDIT SPECIFICATION Audit_Select_Customers
FOR SERVER AUDIT Audit_Read_Access
ADD (SELECT ON dbo.Customers BY PUBLIC)
WITH (STATE = ON);
```

📌 *This specification audits all SELECTs on the* `Customers` *table by any user.*

12.7.4.3 Step 3: Generate Audit Events

Run a SELECT query to trigger the audit:

```
SELECT * FROM dbo.Customers;
```

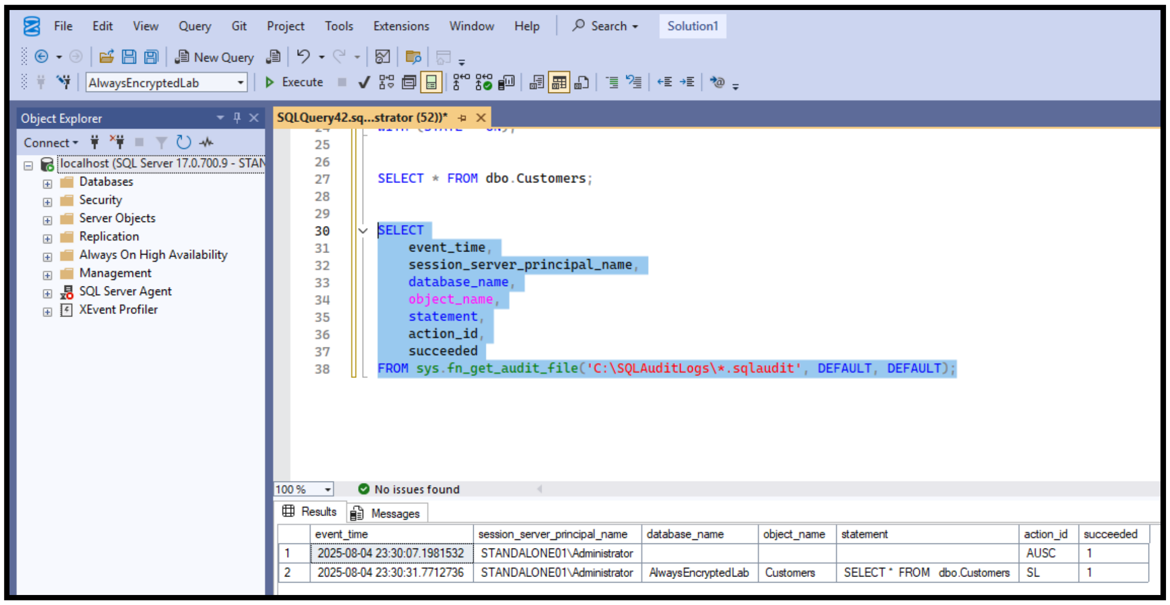
Run this as a different user (if available) to simulate real access.

12.7.4.4 Step 4: Read the Audit Logs

To read and inspect the audit logs stored in the file:

```
SELECT
    event_time,
    session_server_principal_name,
    database_name,
    object_name,
    statement,
    action_id,
    succeeded
FROM sys.fn_get_audit_file('C:\SQLAuditLogs\*.sqlaudit', DEFAULT, DEFAULT);
```

You should see the SELECT statements you ran earlier, with information about the user and the exact query text.



```
SELECT * FROM dbo.Customers;


SELECT
    event_time,
    session_server_principal_name,
    database_name,
    object_name,
    statement,
    action_id,
    succeeded
FROM sys.fn_get_audit_file('C:\SQLAuditLogs\*.sqlaudit', DEFAULT, DEFAULT);
```

	event_time	session_server_principal_name	database_name	object_name	statement	action_id	succeeded
1	2025-08-04 23:30:07.1981532	STANDALONE01\Administrator				AUSC	1
2	2025-08-04 23:30:31.7712736	STANDALONE01\Administrator	AlwaysEncryptedLab	Customers	SELECT * FROM dbo.Customers	SL	1

Figure 12.12: Showing the audit logs.

12.7.4.5 Step 5: (Optional) Audit Other Actions

You can expand the scope to other actions like `INSERT`, `UPDATE`, or `SCHEMA_OBJECT_CHANGE_GROUP`:

```
ALTER DATABASE AUDIT SPECIFICATION Audit_Select_Customers
WITH (STATE = OFF);
GO

ALTER DATABASE AUDIT SPECIFICATION Audit_Select_Customers
ADD (UPDATE ON dbo.Customers BY PUBLIC);
GO

ALTER DATABASE AUDIT SPECIFICATION Audit_Select_Customers
WITH (STATE = ON);
GO
```

This will allow you to track updates on the `Customers` table as well.

12.7.5 Summary

In this exercise, you:

- Created a server audit to capture audit logs in a file
- Monitored SELECT access on a sensitive table (`Customers`)
- Reviewed audit logs via `sys.fn_get_audit_file`
- Learned how SQL Server Audit supports visibility and compliance

✅ *SQL Server Audit is a powerful tool for tracking access to protected data and generating audit trails required by privacy regulations and internal policies.*

12.8 Conclusion

In this chapter, we explored essential features for protecting sensitive data in SQL Server:

- **Dynamic Data Masking (DDM)** to obscure sensitive information in query results.
- **Always Encrypted** to secure data at rest, in transit, and in use.
- **SQL Server Audit** to track and log access to sensitive data.

These features help ensure compliance with data privacy regulations and protect sensitive information from unauthorized access, while still allowing necessary operations on the data.

13 Complying with GDPR and Privacy Regulations

With global regulations like the **General Data Protection Regulation (GDPR)** and others (e.g., CCPA, HIPAA), organizations must ensure that **personal data is handled lawfully, transparently, and securely**. This chapter outlines how to implement privacy-centric practices using SQL Server 2025—supporting **data access, erasure, portability**, and principles like **data minimization** and **pseudonymization**.

13.1 Key GDPR Data Subject Rights

GDPR grants individuals several rights over their personal data. SQL Server solutions must support the ability to fulfill these rights effectively and securely.

13.1.1 Right of Access (Article 15)

The right of access ensures that individuals can obtain confirmation as to whether their personal data is being processed, and, if so, access to that data along with information about its use. Organizations must be able to identify and retrieve all relevant personal data for a specific user upon request. This typically involves searching across multiple tables and systems to gather a comprehensive data set.

In SQL Server, supporting this right means designing your schema and queries to efficiently locate and return all personal data linked to a user. This may require maintaining clear relationships between user identities and their associated records, and ensuring that sensitive data is not inadvertently exposed to unauthorized users. Implementing robust authentication and authorization controls is essential to prevent data leaks during access requests.

> *Users have the right to view their personal data.*

We can create views or stored procedures that return personal records based on the user’s identity. Following the principle of least privilege, we can restrict access to only the data that the user is authorized to see.

- Create **views or stored procedures** to return personal records based on identity.
- Use **Row-Level Security (RLS)** to restrict access to the requesting user only.

Here's an example of a view that allows users to access their own personal information:

```sql
-- Example: View for user to access their own data
CREATE VIEW HR.vw_MyPersonalInfo AS
SELECT FirstName, LastName, Email, HireDate
FROM HR.Employees
WHERE LoginName = SYSTEM_USER;
```

13.1.2 Right to Erasure / Right to Be Forgotten (Article 17)

The right to erasure allows individuals to request the deletion of their personal data when it is no longer necessary for the purposes for which it was collected, or if they withdraw consent on which the processing is based. Organizations must implement processes to handle such requests efficiently while ensuring compliance with legal obligations.

> *Users can request that their personal data be deleted.*

To implement the right to erasure, you can create a stored procedure that deletes a user's personal data from all relevant tables. This procedure should ensure that all associated records are also removed to prevent orphaned data.

- Design `DELETE` logic or masking strategies.
- Implement a **soft-delete** mechanism for traceability.
- Audit every erase request.

We can implement soft-delete logic by adding a flag to indicate that the record is deleted, rather than physically removing it from the database. This allows for traceability while complying with the right to erasure.

```sql
-- Soft-delete example
UPDATE Customers
SET IsDeleted = 1, DeletedAt = GETDATE()
WHERE CustomerID = @UserID;
```

13.1.3 Right to Data Portability (Article 20)

The right to data portability allows individuals to receive their personal data in a structured, commonly used, and machine-readable format, and to transmit that data to another controller without hindrance. This right is particularly relevant when users wish to switch service providers or access their data for personal use.

> *Users can request a structured, machine-readable export of their data.*

We can create a stored procedure that exports user data into a format like CSV or JSON. This procedure should ensure that the exported data is structured and includes all relevant personal information.

- Provide data in formats like JSON, CSV, or XML.
- Use `FOR JSON` or `FOR XML` to generate export formats.

```sql
-- Export customer profile as JSON
SELECT CustomerID, FirstName, LastName, Email
FROM Customers
WHERE CustomerID = @UserID
FOR JSON PATH;
```

> *Combine with application-layer file generation or BCP/export tools.*

13.2 Data Minimization

Data minimization is a key principle of GDPR that requires organizations to collect and process only the personal data that is necessary for the intended purpose. This means avoiding the collection of excessive or irrelevant data, which can reduce the risk of data breaches and enhance user privacy.

> *Collect and retain only data that is necessary for processing.*

Follow these techniques to implement data minimization in SQL Server:

- Avoid `SELECT *` queries; use only needed columns.
- Archive or purge unused fields periodically.
- Use **views** or **column-level security** to restrict visibility.

```
-- Example: Limit exposed data
CREATE VIEW Public.vw_EmployeeDirectory AS
SELECT FirstName, LastName, Department
FROM HR.Employees;
```

13.3 Pseudonymization

Pseudonymization is a technique that replaces personal identifiers with pseudonyms or tokens, allowing data to be processed without directly identifying individuals. This approach helps reduce the risk of re-identification while still enabling data analysis and processing.

> *Transform personal data so it can't be attributed to a person without additional info.*

Here are some common pseudonymization techniques:

- Replace names or emails with pseudonyms or tokens.
- Use hashing, encryption, or masking.

```
-- Hash email for pseudonymization
SELECT CustomerID, HASHBYTES('SHA2_256', Email) AS PseudoEmail
FROM Customers;
```

> *Store original values separately with strict access control.*

13.4 Data Classification in SQL Server 2025

SQL Server 2025 supports data classification and sensitivity labels to help organizations identify and manage sensitive data. This feature allows you to classify data based on its sensitivity level, making it easier to apply security measures and compliance controls.

To classify sensitive data, you can use the `ADD SENSITIVITY CLASSIFICATION` command:

```
-- Mark column as sensitive
ADD SENSITIVITY CLASSIFICATION TO Customers.Email
WITH (LABEL = 'Confidential', INFORMATION_TYPE = 'Contact Info');
```

*Use **SQL Server Management Studio (SSMS)** to audit and review classifications.*

13.5 Auditing for Compliance

Auditing is essential for compliance with GDPR and other privacy regulations. SQL Server provides built-in auditing features that allow you to track data access, modifications, and security events. This helps organizations demonstrate compliance and investigate potential breaches.

To comply with GDPR, organizations must log:

- Data access events
- Erasure and export requests
- Permission changes

Following SQL Server Audit features can help you track compliance:

```
CREATE DATABASE AUDIT SPECIFICATION GDPR_AccessLog
FOR SERVER AUDIT GDPRAudit
ADD (SELECT ON Customers BY public),
ADD (DELETE ON Customers BY public);
```

13.6 Best Practices for GDPR Compliance

Following best practices ensures that your SQL Server environment is compliant with GDPR and other privacy regulations. Here are some key practices:

Practice	Why It Matters
Apply RLS and DDM	Minimize unnecessary data exposure
Track data exports and erasures	Mandatory for legal compliance
Use pseudonymization where possible	Helps reduce risk in analytics
Purge expired or unused data regularly	Aligns with data minimization principle

Practice	Why It Matters
Create procedures/views for data access requests	Supports timely, accurate responses

13.7 Exercise 33: Apply DDM to PII Columns

13.7.1 Description

In this exercise, we'll explore how to apply **Dynamic Data Masking (DDM)** in SQL Server 2025 to **protect Personally Identifiable Information (PII)** such as emails, phone numbers, and national ID numbers. DDM helps minimize exposure of sensitive data by automatically masking it in the result set based on the user's access privileges.

You will define masking rules on selected columns and test how users with different privileges see the data differently—crucial for meeting **GDPR** data minimization and privacy-by-default principles.

13.7.2 Objectives

By the end of this hands-on lab, you will:

- Create a table containing PII fields
- Apply dynamic masking functions to relevant columns
- Create test users to validate how masking works
- Understand how DDM supports GDPR-aligned data protection strategies

13.7.3 Prerequisites

- SQL Server 2025 installed
- SQL Server Management Studio (SSMS) 21.x
- sysadmin or db_owner privileges to create database and users
- Windows or SQL logins (can be test/demo users)

13.7.4 Steps

Here's a step-by-step guide to complete this exercise:

13.7.4.1 Step 1: Create the Working Database and Table

Let’s start by creating a fresh database to isolate this lab and define a simple table with typical PII fields.

```
CREATE DATABASE GDPRLab;
GO
USE GDPRLab;
GO

CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY,
    FullName NVARCHAR(100),
    Email NVARCHAR(100) MASKED WITH (FUNCTION = 'email()'),
    Phone VARCHAR(20) MASKED WITH (FUNCTION = 'partial(0,"XXX-XXX-",4)'),
    SSN CHAR(11) MASKED WITH (FUNCTION = 'default()')
);
```

📌 The `MASKED WITH` clause applies the masking function.

- `email()` masks the email except for the first character and domain.
- `partial()` masks part of the phone number.
- `default()` fully obfuscates the value.

13.7.4.2 Step 2: Insert Sample Data

Add some rows so we can later observe how masking works.

```
INSERT INTO Customers (CustomerID, FullName, Email, Phone, SSN)
VALUES
(1, 'Devi Johnson', 'devi.johnson@ilmudata.id', '555-123-4567', '123-45-6789'),
(2, 'Smith Lee', 'smith.lee@ilmudata.id', '555-987-6543', '987-65-4321'),
(3, 'Hans Müller', 'hans.mueller@ilmudata.id', '555-111-2222', '321-54-6789'),
(4, 'Giulia Rossi', 'giulia.rossi@ilmudata.id', '555-333-4444', '654-32-1987'),
(5, 'Pierre Dubois', 'pierre.dubois@ilmudata.id', '555-555-6666', '789-12-3456'),
(6, 'Sven de Vries', 'sven.devries@ilmudata.id', '555-777-8888', '876-54-3210');
```

This creates a diverse set of customers with different names, emails, phones, and SSNs.

13.7.4.3 Step 3: Create a Low-Privilege User

Now let’s create a login and user with *read-only* access (but no UNMASK permission):

```
CREATE LOGIN AnalystUser WITH PASSWORD = 'StrongPassword!123';
CREATE USER AnalystUser FOR LOGIN AnalystUser;
ALTER ROLE db_datareader ADD MEMBER AnalystUser;
```

🧠 *The AnalystUser can read data but should see it masked.*

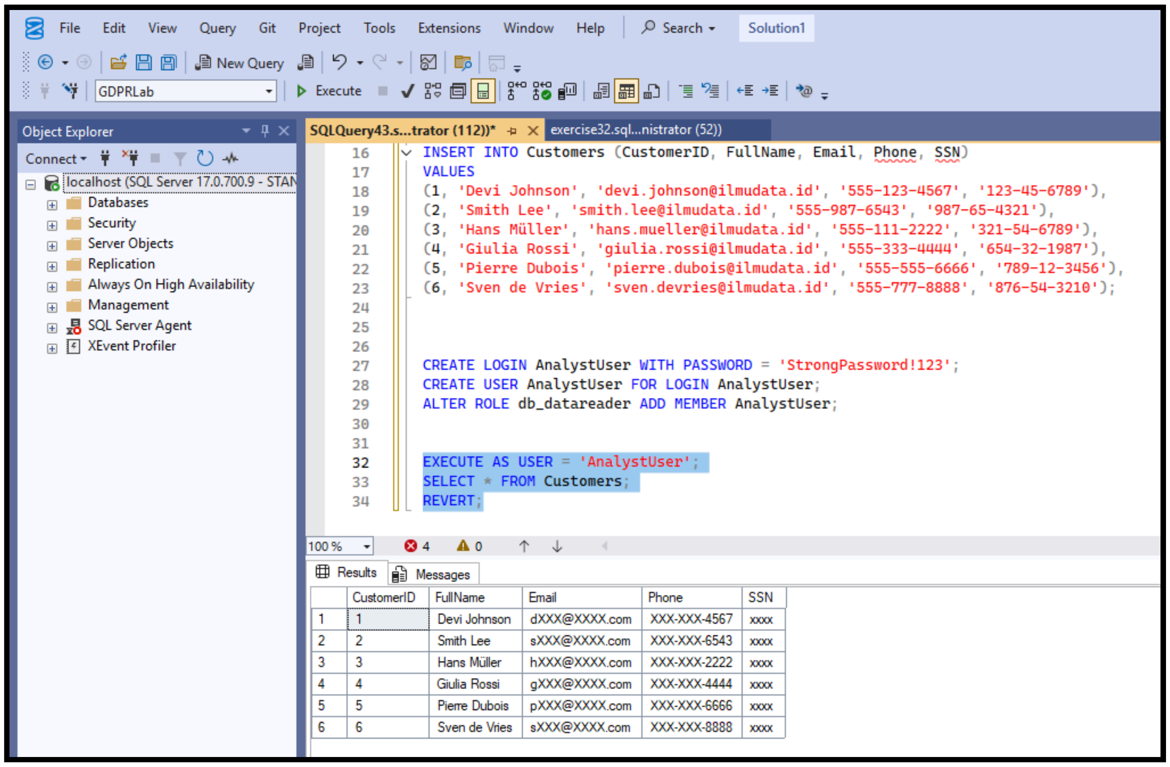
13.7.4.4 Step 4: Test the Masking Behavior

Now log in as `AnalystUser` (or simulate using `EXECUTE AS`) and run:

```
EXECUTE AS USER = 'AnalystUser';
SELECT * FROM Customers;
REVERT;
```

✅ You will see:

- Masked email like `aXXXX@XXXX.com`
- Partially masked phone like `XXX-XXX-4567`
- SSN shown as `XXXXXXX`



```
INSERT INTO Customers (CustomerID, FullName, Email, Phone, SSN)
VALUES
(1, 'Devi Johnson', 'devi.johnson@ilmudata.id', '555-123-4567', '123-45-6789'),
(2, 'Smith Lee', 'smith.lee@ilmudata.id', '555-987-6543', '987-65-4321'),
(3, 'Hans Müller', 'hans.mueller@ilmudata.id', '555-111-2222', '321-54-6789'),
(4, 'Giulia Rossi', 'giulia.rossi@ilmudata.id', '555-333-4444', '654-32-1987'),
(5, 'Pierre Dubois', 'pierre.dubois@ilmudata.id', '555-555-6666', '789-12-3456'),
(6, 'Sven de Vries', 'sven.devries@ilmudata.id', '555-777-8888', '876-54-3210');



CREATE LOGIN AnalystUser WITH PASSWORD = 'StrongPassword!123';
CREATE USER AnalystUser FOR LOGIN AnalystUser;
ALTER ROLE db_datareader ADD MEMBER AnalystUser;


EXECUTE AS USER = 'AnalystUser';
SELECT * FROM Customers;
REVERT;
```

	CustomerID	FullName	Email	Phone	SSN
1	1	Devi Johnson	dXXX@XXXX.com	XXX-XXX-4567	xxxx
2	2	Smith Lee	sXXX@XXXX.com	XXX-XXX-6543	xxxx
3	3	Hans Müller	hXXX@XXXX.com	XXX-XXX-2222	xxxx
4	4	Giulia Rossi	gXXX@XXXX.com	XXX-XXX-4444	xxxx
5	5	Pierre Dubois	pXXX@XXXX.com	XXX-XXX-6666	xxxx
6	6	Sven de Vries	sXXX@XXXX.com	XXX-XXX-8888	xxxx

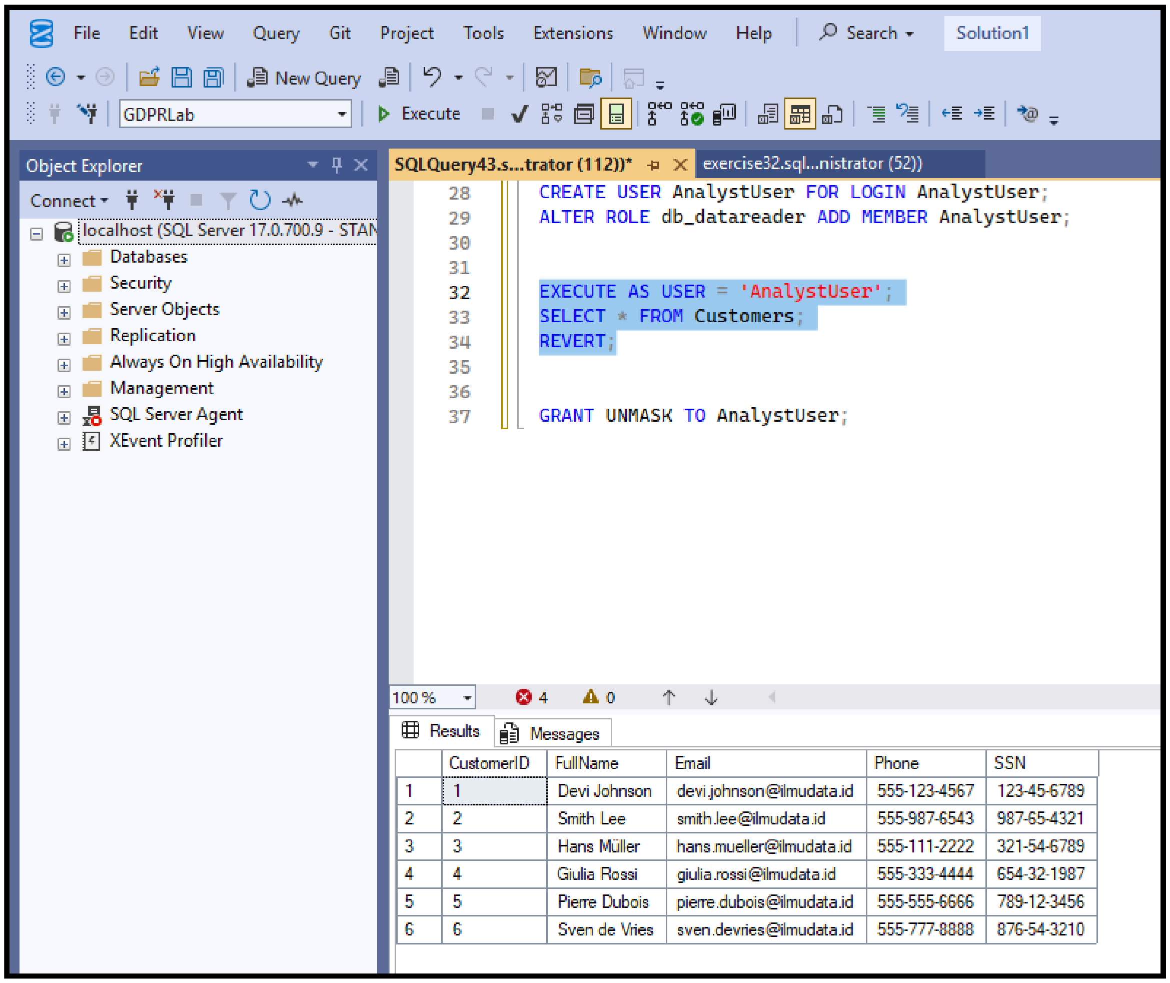
Figure 13.1: Showing masked data.

13.7.4.5 Step 5: Unmask Privilege for Admin Role

If someone with full privileges needs to see the original data, you can grant `UNMASK`.

```
GRANT UNMASK TO AnalystUser;
```

Then re-run the `SELECT * FROM Customers` as `AnalystUser` and observe the difference.



```sql
CREATE USER AnalystUser FOR LOGIN AnalystUser;
ALTER ROLE db_datareader ADD MEMBER AnalystUser;


EXECUTE AS USER = 'AnalystUser';
SELECT * FROM Customers;
REVERT;


GRANT UNMASK TO AnalystUser;
```

	CustomerID	FullName	Email	Phone	SSN
1	1	Devi Johnson	devi.johnson@ilmudata.id	555-123-4567	123-45-6789
2	2	Smith Lee	smith.lee@ilmudata.id	555-987-6543	987-65-4321
3	3	Hans Müller	hans.mueller@ilmudata.id	555-111-2222	321-54-6789
4	4	Giulia Rossi	giulia.rossi@ilmudata.id	555-333-4444	654-32-1987
5	5	Pierre Dubois	pierre.dubois@ilmudata.id	555-555-6666	789-12-3456
6	6	Sven de Vries	sven.devries@ilmudata.id	555-777-8888	876-54-3210

Figure 13.2: Showing unmasked data.

13.7.5 Summary

In this lab, you:

- Created a table with PII data (email, phone, SSN)
- Applied various masking functions using DDM
- Verified the effect of masking on a limited user account
- Used `UNMASK` permission to control visibility

✅ *Dynamic Data Masking is a lightweight, built-in mechanism to help protect sensitive data from unauthorized access and aligns with GDPR's privacy-by-design principle.*

13.8 Exercise 34: Apply Pseudonymization with Computed Columns or Hashes

13.8.1 Description

In this exercise, you'll learn how to implement **pseudonymization** techniques using computed columns and hashing functions in SQL Server 2025. While masking hides data on output, pseudonymization transforms the data at rest—an essential GDPR strategy for reducing re-identification risks while allowing analytical usage.

We'll use SHA2 hashing to pseudonymize sensitive fields like email and phone. This approach supports privacy-by-design practices without sacrificing data integrity in analytics.

13.8.2 Objectives

By the end of this lab, you will:

- Understand the concept of pseudonymization vs. masking
- Use SQL Server's built-in hashing function `HASHBYTES`
- Create computed columns that store hashed PII
- Explore usage in GDPR-compliant analytics

13.8.3 Prerequisites

- SQL Server 2025 installed
- SQL Server Management Studio (SSMS) 21.x
- sysadmin or db_owner privileges
- Enable `CONCAT_NULL_YIELDS_NULL` (default is ON)

13.8.4 Steps

Here's a step-by-step guide to complete this exercise:

13.8.4.1 Step 1: Create a New Database and Table

Start by creating a new working environment to isolate pseudonymized data.

```
CREATE DATABASE GDPRPseudonymLab;
GO
USE GDPRPseudonymLab;
GO

CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY,
    FullName NVARCHAR(100),
    Email NVARCHAR(100),
    Phone VARCHAR(20),
    EmailHash AS CONVERT(VARCHAR(100), HASHBYTES('SHA2_256', Email), 2) PERSISTED,
    PhoneHash AS CONVERT(VARCHAR(100), HASHBYTES('SHA2_256', Phone), 2) PERSISTED
);
```

📌 Here:

- `HASHBYTES('SHA2_256', column)` produces a 256-bit hash
- `CONVERT(..., 2)` renders the binary hash as a hex string
- `PERSISTED` ensures the computed hash is stored physically and indexable

13.8.4.2 Step 2: Insert Sample Data

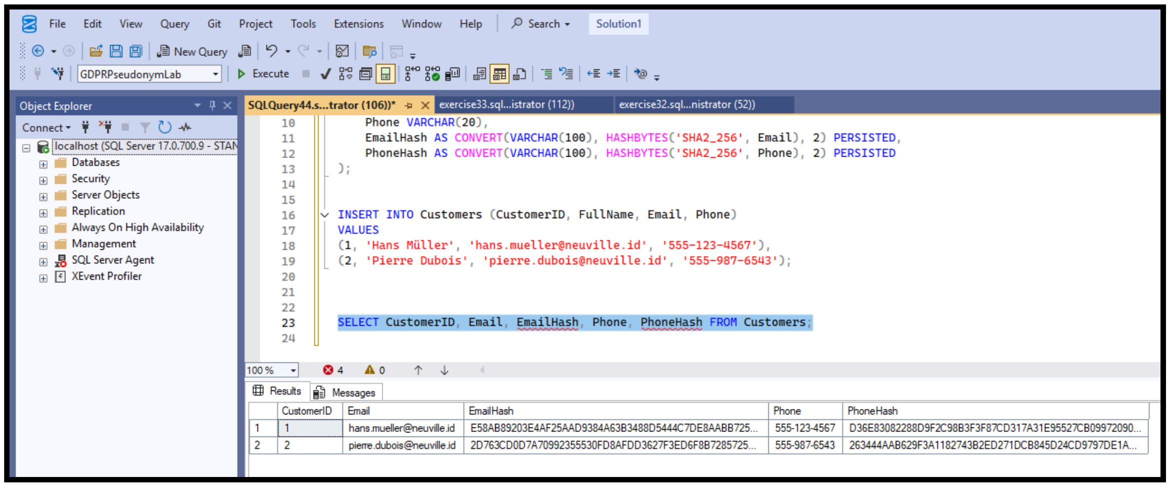
Add records with real email and phone numbers.

```
INSERT INTO Customers (CustomerID, FullName, Email, Phone)
VALUES
(1, 'Hans Müller', 'hans.mueller@neuville.id', '555-123-4567'),
(2, 'Pierre Dubois', 'pierre.dubois@neuville.id', '555-987-6543');
```

Now query to view the hashed fields:

```
SELECT CustomerID, Email, EmailHash, Phone, PhoneHash FROM Customers;
```

You'll see `EmailHash` and `PhoneHash` show long SHA-256 hash values.



```
    Phone VARCHAR(20),
    EmailHash AS CONVERT(VARCHAR(100), HASHBYTES('SHA2_256', Email), 2) PERSISTED,
    PhoneHash AS CONVERT(VARCHAR(100), HASHBYTES('SHA2_256', Phone), 2) PERSISTED
);


INSERT INTO Customers (CustomerID, FullName, Email, Phone)
VALUES
(1, 'Hans Müller', 'hans.mueller@neuville.id', '555-123-4567'),
(2, 'Pierre Dubois', 'pierre.dubois@neuville.id', '555-987-6543');



SELECT CustomerID, Email, EmailHash, Phone, PhoneHash FROM Customers;
```

	CustomerID	Email	EmailHash	Phone	PhoneHash
1	1	hans.mueller@neuville.id	E58AB89203E4AF25AAD9384A63B3488D5444C7DE8AABB725...	555-123-4567	D36E83082288D9F2C98B3F3F87CD317A31E95527CB09972090...
2	2	pierre.dubois@neuville.id	2D763CD0D7A70992355530FD8AFDD3627F3ED6F8B7285725...	555-987-6543	263444AAB629F3A1182743B2ED271DCB845D24CD9797DE1A...

Figure 13.3: Showing hashed data.

13.8.4.3 Step 3: Use Hashed Columns for Analytics Joins

Let’s say you need to pseudonymously join customers with another table using hashed emails:

```
CREATE TABLE EmailCampaigns (
    CampaignID INT,
    TargetEmailHash VARCHAR(100)
);

INSERT INTO EmailCampaigns (CampaignID, TargetEmailHash)
VALUES
(1001, (SELECT EmailHash FROM Customers WHERE CustomerID = 1));
```

Now pseudonymously match records:

```
SELECT c.CustomerID, c.FullName, e.CampaignID
FROM Customers c
JOIN EmailCampaigns e ON c.EmailHash = e.TargetEmailHash;
```

🧠 You achieved a GDPR-aligned join *without* using raw PII.

13.8.4.4 Step 4: Add Index on Hashed Columns (Optional)

You can index the hashed column for fast filtering:

```
CREATE INDEX IX_Customers_EmailHash ON Customers (EmailHash);
```

This supports performance in large analytical datasets.

13.8.5 Summary

In this lab, you:

- Created computed columns with `HASHBYTES` for pseudonymization
- Stored SHA-256 hashes for email and phone
- Used hashed fields for secure joins
- Understood the distinction between masking and pseudonymization

✅ *Pseudonymization enhances privacy compliance and enables GDPR-compatible analytics without compromising data utility.*

13.9 Exercise 35: Implement the Right to Erasure and Portability

13.9.1 Description

The General Data Protection Regulation (GDPR) grants individuals the right to **access**, **port**, and **erase** their personal data. In this lab, you’ll learn how to model and implement these rights in SQL Server 2025 using T-SQL. You will create mechanisms to:

- Export user data in a machine-readable format (CSV-compatible)
- Erase personal data while preserving referential integrity

These operations are foundational for compliance in modern business systems.

13.9.2 Objectives

By completing this exercise, you will:

- Export PII and transactional data in a readable and portable format
- Pseudonymize or erase data fields to support “right to be forgotten”
- Use safe deletion via UPDATE and NULLing strategies instead of physical DELETE

13.9.3 Prerequisites

- SQL Server 2025 installed
- SQL Server Management Studio (SSMS) 21.x
- `GDPRPseudonymLab` database from Exercise 34 (or create a new one)
- Sufficient privileges to modify and select from the database

13.9.4 Steps

Here’s a step-by-step guide to complete this exercise:

13.9.4.1 Step 1: Add Export-Ready View for User Data

You’ll create a view that exposes only the relevant personal data for portability (e.g., to share with the user upon request).

```
USE GDPRPseudonymLab;
GO

CREATE VIEW vCustomerExport AS
```

```
SELECT
    CustomerID,
    FullName,
    Email,
    Phone
FROM Customers;
```

Now, run:

```
SELECT * FROM vCustomerExport WHERE CustomerID = 1;
```

📤 This supports *portability* by exporting a clean snapshot of the user's PII.

13.9.4.2 Step 2: Simulate CSV Output for Portability

You can simulate CSV export using concatenation. Tools like SSMS or apps can copy/paste this output to Excel or flat files.

```
SELECT
    '"' + CAST(CustomerID AS VARCHAR) + '","' +
    FullName + '","' +
    Email + '","' +
    Phone + '"' AS CSVRow
FROM vCustomerExport
WHERE CustomerID = 1;
```

This produces a single CSV row for the customer, which can be copied to a file.

13.9.4.3 Step 3: Implement Right to Erasure with UPDATE

Instead of `DELETE`, which breaks foreign key relationships, use `UPDATE` to nullify or pseudonymize PII:

```
UPDATE Customers
SET
    FullName = NULL,
    Email = NULL,
    Phone = NULL
WHERE CustomerID = 1;
```

Now recheck:

```
SELECT * FROM Customers WHERE CustomerID = 1;
```

✅ The personal data is erased, but surrogate key (`CustomerID`) and related records can be retained for audit or aggregation purposes.

13.9.4.4 Step 4: Use a Reversible Erasure Flag (Optional)

Add a column to track erasure status for audit or recovery in non-production environments:

```sql
ALTER TABLE Customers ADD IsErased BIT DEFAULT 0;

-- Erase customer PII and flag it
UPDATE Customers
SET
    FullName = NULL,
    Email = NULL,
    Phone = NULL,
    IsErased = 1
WHERE CustomerID = 2;
```

Now you can query:

```sql
SELECT * FROM Customers WHERE IsErased = 1;
```

You'll see records with `NULL` PII but still retain the `CustomerID` for traceability.

13.9.5 Summary

In this GDPR-focused hands-on lab, you:

- Exported user data for portability in a flat format
- Simulated a compliant CSV row
- Erased personal information using `UPDATE` rather than `DELETE`
- Optionally tracked erasure with a status flag

🔐 These techniques help businesses comply with GDPR while maintaining data model integrity and traceability.

13.10 Exercise 36: Simulate GDPR "Right to Be Forgotten"

13.10.1 Description

The GDPR mandates that users can request to be forgotten. In database terms, this often translates to **soft deletion** (marking a record as deleted rather than physically removing it) and **anonymization** (removing personal identifiers). This exercise simulates this process on a customer table using SQL Server 2025.

You will create anonymization logic that replaces personally identifiable information (PII) with generic placeholders and mark the record as erased, preserving referential integrity and auditability.

13.10.2 Objectives

By the end of this exercise, you will:

- Soft-delete records using an `IsDeleted` flag
- Anonymize sensitive data like name, email, and phone
- Build and run reusable `Stored Procedure` for GDPR erasure requests

13.10.3 Prerequisites

- SQL Server 2025
- SQL Server Management Studio (SSMS) 21.x
- A database named `GDPRLab` (create it below if needed)
- Table `Customers` with sample PII

13.10.4 Steps

Here’s a step-by-step guide to complete this exercise:

13.10.4.1 Step 1: Create and Seed the `Customers` Table

Let’s build the setup to simulate real user data and PII.

```
CREATE DATABASE GDPRLab;
GO
USE GDPRLab;
GO

CREATE TABLE Customers (
    CustomerID INT IDENTITY PRIMARY KEY,
    FullName NVARCHAR(100),
    Email NVARCHAR(100),
    Phone NVARCHAR(50),
    RegisteredDate DATE,
    IsDeleted BIT DEFAULT 0
);
GO

INSERT INTO Customers (FullName, Email, Phone, RegisteredDate)
VALUES
('Ujang Johnson', 'ujang.johnson@ilmudata.id', '555-1234', '2024-05-10'),
('James Turner', 'james.turner@neuville.id', '555-1122', '2024-04-18'),
('Sophie Evans', 'sophie.evans@ilmudata.id', '555-3344', '2024-03-22'),
('Lucía García', 'lucia.garcia@ilmudata.id', '555-5566', '2024-02-14'),
('Carlos Martínez', 'carlos.martinez@neuville.id', '555-7788', '2024-01-30'),
('Giulia Rossi', 'giulia.rossi@ilmudata.id', '555-9900', '2024-05-25'),
('Marco Bianchi', 'marco.bianchi@neuville.id', '555-2233', '2024-06-05');
```

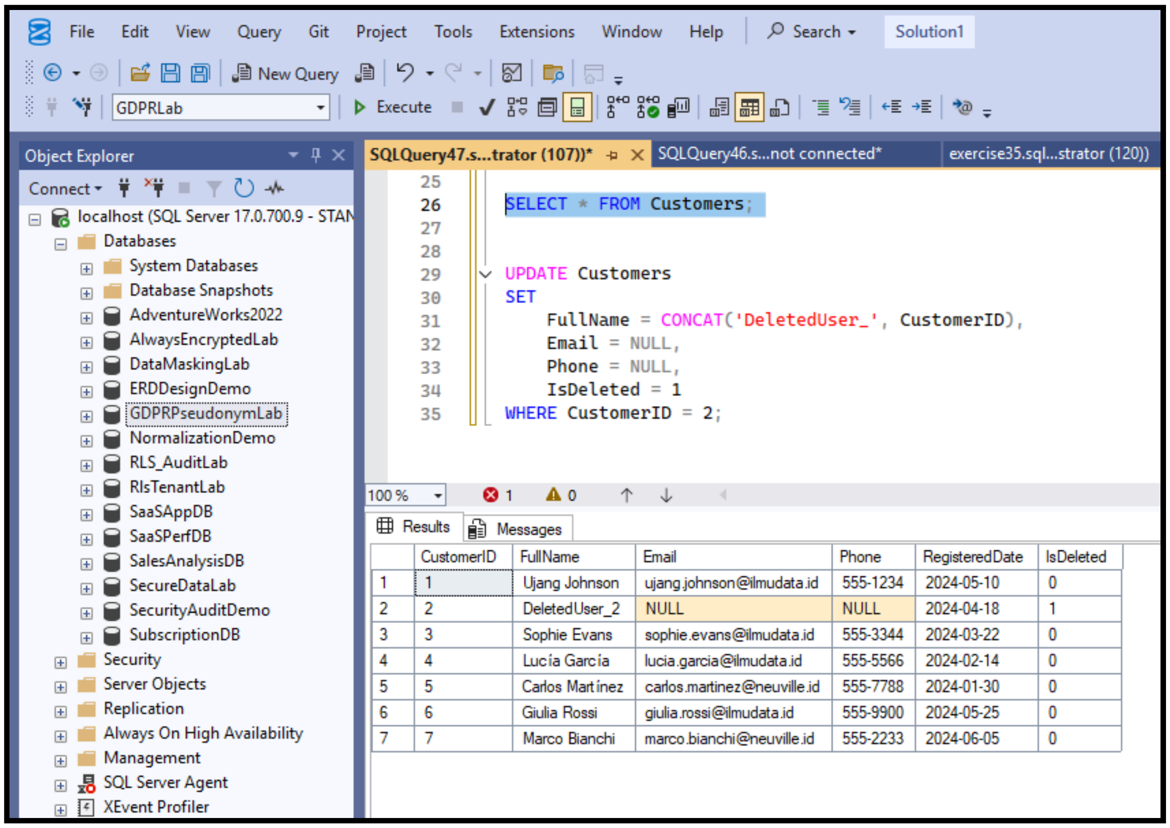
This creates three sample customers in the `Customers` table.

13.10.4.2 Step 2: Define the Anonymization and Soft Delete Logic

We'll simulate the "Right to Be Forgotten" by replacing PII with anonymized values and setting `IsDeleted = 1`.

```
UPDATE Customers
SET
    FullName = CONCAT('DeletedUser_', CustomerID),
    Email = NULL,
    Phone = NULL,
    IsDeleted = 1
WHERE CustomerID = 2;
```

🔐 You retain the `CustomerID` and `RegisteredDate` but anonymize the user.



```
SELECT * FROM Customers;


UPDATE Customers
SET
    FullName = CONCAT('DeletedUser_', CustomerID),
    Email = NULL,
    Phone = NULL,
    IsDeleted = 1
WHERE CustomerID = 2;
```

	CustomerID	FullName	Email	Phone	RegisteredDate	IsDeleted
1	1	Ujang Johnson	ujang.johnson@ilmudata.id	555-1234	2024-05-10	0
2	2	DeletedUser_2	NULL	NULL	2024-04-18	1
3	3	Sophie Evans	sophie.evans@ilmudata.id	555-3344	2024-03-22	0
4	4	Lucía García	lucia.garcia@ilmudata.id	555-5566	2024-02-14	0
5	5	Carlos Martínez	carlos.martinez@neuville.id	555-7788	2024-01-30	0
6	6	Giulia Rossi	giulia.rossi@ilmudata.id	555-9900	2024-05-25	0
7	7	Marco Bianchi	marco.bianchi@neuville.id	555-2233	2024-06-05	0

Figure 13.4: Showing anonymized data.

13.10.4.3 Step 3: Create a Stored Procedure to Generalize the Action

Make the process reusable via a stored procedure:

```
CREATE PROCEDURE AnonymizeAndSoftDeleteCustomer
    @CustomerID INT
AS
BEGIN
    UPDATE Customers
    SET
        FullName = CONCAT('DeletedUser_', @CustomerID),
        Email = NULL,
        Phone = NULL,
        IsDeleted = 1
    WHERE CustomerID = @CustomerID;
END;
```

Now run it:

```
EXEC AnonymizeAndSoftDeleteCustomer @CustomerID = 3;
```

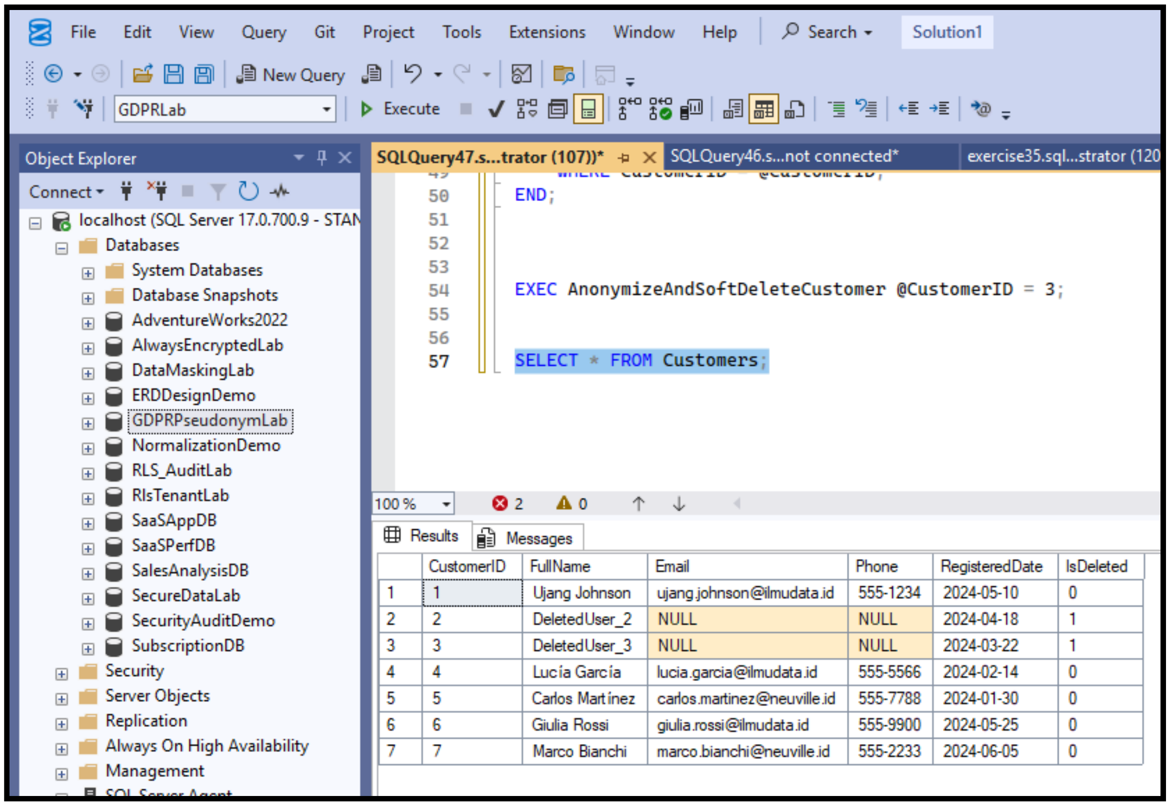
🎯 This ensures consistent anonymization logic across requests.

13.10.4.4 Step 4: Query to Verify Forgotten Records

Check results:

```
SELECT * FROM Customers;
```

You’ll see that James and Sophie’s records are anonymized and flagged as deleted:



```sql
END;


EXEC AnonymizeAndSoftDeleteCustomer @CustomerID = 3;

SELECT * FROM Customers;
```

	CustomerID	FullName	Email	Phone	RegisteredDate	IsDeleted
1	1	Ujang Johnson	ujang.johnson@ilmudata.id	555-1234	2024-05-10	0
2	2	DeletedUser_2	NULL	NULL	2024-04-18	1
3	3	DeletedUser_3	NULL	NULL	2024-03-22	1
4	4	Lucía García	lucia.garcia@ilmudata.id	555-5566	2024-02-14	0
5	5	Carlos Martínez	carlos.martinez@neuville.id	555-7788	2024-01-30	0
6	6	Giulia Rossi	giulia.rossi@ilmudata.id	555-9900	2024-05-25	0
7	7	Marco Bianchi	marco.bianchi@neuville.id	555-2233	2024-06-05	0

Figure 13.5: Showing customer data.

13.10.5 Summary

In this hands-on lab, you:

- Created a sample customer table with PII
- Simulated GDPR erasure using anonymization and soft deletion
- Built a stored procedure to automate the “Right to Be Forgotten” process

This pattern enables regulatory compliance while maintaining integrity in transactional systems.

13.11 Exercise 37: Enable Auditing and Access Log for GDPR

13.11.1 Description

To comply with GDPR, it’s essential to track **who accessed personal data, when, and why**. This exercise demonstrates how to set up **SQL Server Audit** to capture access events to personal data stored in the database. You will create a server-level audit and a database audit specification that tracks `SELECT` operations on a PII-related table.

13.11.2 Objectives

By the end of this exercise, you will:

- Configure a SQL Server Audit object
- Create a database audit specification for `SELECT` access
- Review logs to identify data access events

13.11.3 Prerequisites

- SQL Server 2025
- SQL Server Management Studio (SSMS) 21.x
- Database `GDPRLab` with the `Customers` table (from Exercise 36)
- Sufficient permissions (SQL Server sysadmin or audit admin)

13.11.4 Steps

Here's a step-by-step guide to complete this exercise:

13.11.4.1 Step 1: Create a Server Audit Object

We begin by creating a server audit that writes events to a file.

```
USE master;
GO

CREATE SERVER AUDIT GDPR_Data_Access_Audit
TO FILE (
    FILEPATH = 'C:\SQLAuditLogs\',  -- Ensure this folder exists
    MAXSIZE = 10 MB,
    MAX_ROLLOVER_FILES = 5,
    RESERVE_DISK_SPACE = OFF
)
WITH (ON_FAILURE = CONTINUE);
GO

ALTER SERVER AUDIT GDPR_Data_Access_Audit
WITH (STATE = ON);
```

📝 This sets up a file-based audit log. Replace the `FILEPATH` with a valid path on your SQL Server machine.

13.11.4.2 Step 2: Create a Database Audit Specification

Now link specific audit actions to the `GDPRLab` database:

```
USE GDPRLab;
GO

CREATE DATABASE AUDIT SPECIFICATION GDPR_Customers_Access_Spec
FOR SERVER AUDIT GDPR_Data_Access_Audit
ADD (SELECT ON OBJECT::dbo.Customers BY PUBLIC)
WITH (STATE = ON);
```

📌 This captures any `SELECT` access to the `Customers` table, regardless of the user.

13.11.4.3 Step 3: Simulate a Data Access Event

Let’s mimic a user querying the `Customers` table:

```
USE GDPRLab;
GO

SELECT * FROM dbo.Customers;
```

This action is now captured in the audit logs.

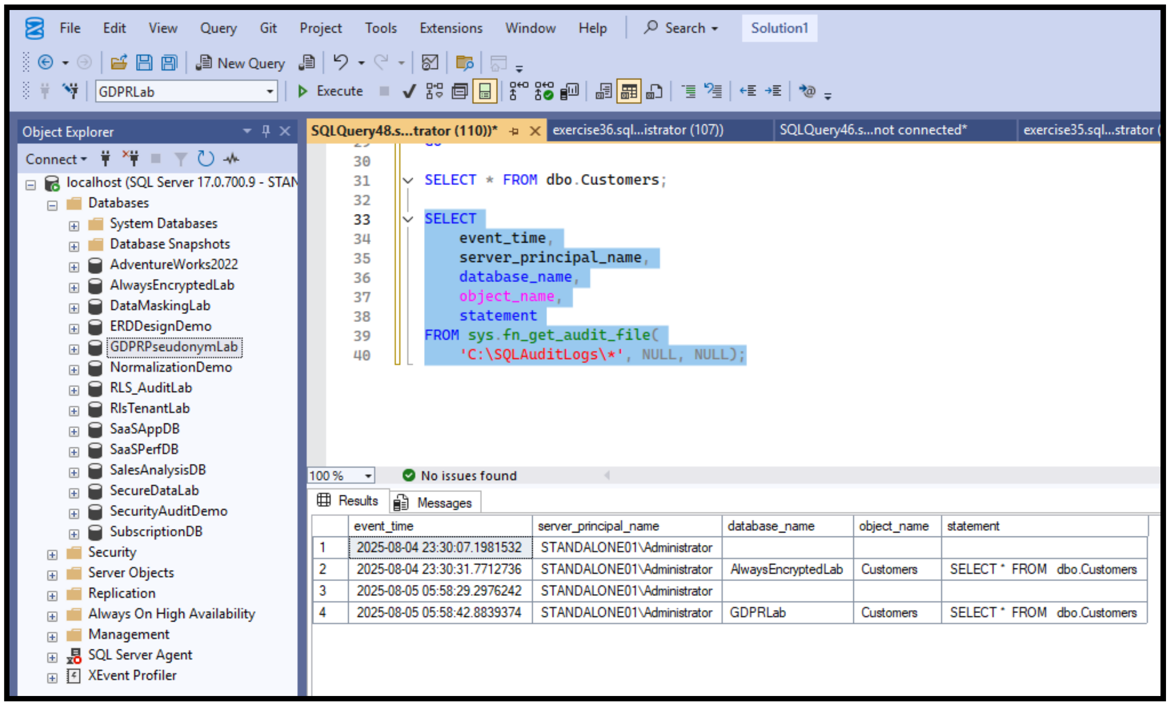
13.11.4.4 Step 4: View the Audit Logs

Use the following script to query the audit records:

```
SELECT
    event_time,
    server_principal_name,
    database_name,
    object_name,
    statement
FROM sys.fn_get_audit_file(
    'C:\SQLAuditLogs\*', NULL, NULL);
```

This shows you:

- When the access occurred
- Who accessed the table
- What object was accessed
- The query issued (if applicable)



```sql
SELECT * FROM dbo.Customers;

SELECT
    event_time,
    server_principal_name,
    database_name,
    object_name,
    statement
FROM sys.fn_get_audit_file(
    'C:\SQLAuditLogs\*', NULL, NULL);
```

	event_time	server_principal_name	database_name	object_name	statement
1	2025-08-04 23:30:07.1981532	STANDALONE01\Administrator			
2	2025-08-04 23:30:31.7712736	STANDALONE01\Administrator	AlwaysEncryptedLab	Customers	SELECT * FROM dbo.Customers
3	2025-08-05 05:58:29.2976242	STANDALONE01\Administrator			
4	2025-08-05 05:58:42.8839374	STANDALONE01\Administrator	GDPRLab	Customers	SELECT * FROM dbo.Customers

Figure 13.6: Showing audit data.

13.11.5 Summary

In this hands-on lab, you:

- Created a server audit and database audit specification
- Monitored `SELECT` access to sensitive PII data
- Verified audit log entries using T-SQL

✅ This setup provides traceability and accountability—an essential part of GDPR compliance when dealing with user personal data.

13.12 Exercise 38: Log Consent and Data Processing Activities for GDPR Audits

13.12.1 Description

GDPR requires that data processing be lawful, transparent, and accountable. Organizations must track whether users have given **explicit consent** to process their personal data, along with when and how that consent was granted. In this lab, you'll implement a **consent logging mechanism** and simulate logging of **data processing events** tied to that consent.

13.12.2 Objectives

By the end of this exercise, you will:

- Create tables to log consent and data processing activities
- Insert and manage consent records
- Simulate GDPR-compliant activity logging for audit purposes

13.12.3 Prerequisites

- SQL Server 2025
- SQL Server Management Studio (SSMS) 21.x
- Database `GDPRLab` with existing user data (from Exercise 36)
- Appropriate permissions to create tables and insert data

13.12.4 Steps

Here’s a step-by-step guide to complete this exercise:

13.12.4.1 Step 1: Create Tables for Consent and Processing Logs

We start by designing two core tables: one for storing user consent, and another for tracking any processing done based on that consent.

```sql
USE GDPRLab;
GO

-- Table to store user consent information
CREATE TABLE ConsentLog (
    ConsentID INT IDENTITY(1,1) PRIMARY KEY,
    UserID INT NOT NULL,
    ConsentGiven BIT NOT NULL,
    ConsentDate DATETIME2 NOT NULL DEFAULT GETDATE(),
    ConsentMethod NVARCHAR(100) NOT NULL, -- e.g., "Checkbox on registration form"
    Notes NVARCHAR(255)
);

-- Table to log data processing activity
CREATE TABLE DataProcessingLog (
    LogID INT IDENTITY(1,1) PRIMARY KEY,
    UserID INT NOT NULL,
    Activity NVARCHAR(100) NOT NULL, -- e.g., "Email marketing", "Exported data"
    ProcessedAt DATETIME2 NOT NULL DEFAULT GETDATE(),
    PerformedBy NVARCHAR(100), -- e.g., app/service/user
    ConsentID INT NULL,
    FOREIGN KEY (ConsentID) REFERENCES ConsentLog(ConsentID)
);
```

✅ These tables support GDPR accountability for **“lawful basis of processing”**.

13.12.4.2 Step 2: Insert a Sample Consent Record

Let’s simulate a user giving consent during signup.

```
INSERT INTO ConsentLog (UserID, ConsentGiven, ConsentMethod, Notes)
VALUES (1001, 1, 'Checkbox on signup form', 'User agreed to receive promotional emails');
```

This stores a positive consent tied to user `1001`.

13.12.4.3 Step 3: Log a Data Processing Activity

Now simulate that data from this user was processed (e.g., for marketing), referencing the consent record.

```
INSERT INTO DataProcessingLog (UserID, Activity, PerformedBy, ConsentID)
VALUES (
    1001,
    'Email campaign - August 2025',
    'MarketingSystemApp',
    (SELECT TOP 1 ConsentID FROM ConsentLog WHERE UserID = 1001 ORDER BY ConsentDate DESC)
);
```

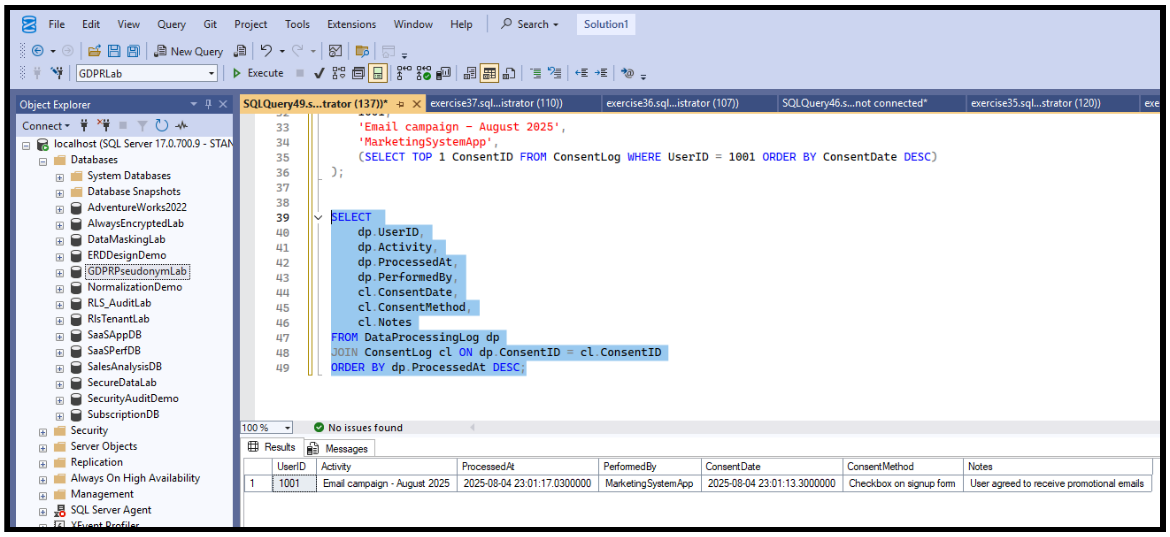
💡 This trace links **who processed what** and **which consent justified it**.

13.12.4.4 Step 4: Query Logs for Compliance Reporting

To retrieve full accountability logs:

```
SELECT
    dp.UserID,
    dp.Activity,
    dp.ProcessedAt,
    dp.PerformedBy,
    cl.ConsentDate,
    cl.ConsentMethod,
    cl.Notes
FROM DataProcessingLog dp
JOIN ConsentLog cl ON dp.ConsentID = cl.ConsentID
ORDER BY dp.ProcessedAt DESC;
```

This gives a **clear audit trail** of activity justified by explicit consent.



```sql
    'Email campaign - August 2025',
    'MarketingSystemApp',
    (SELECT TOP 1 ConsentID FROM ConsentLog WHERE UserID = 1001 ORDER BY ConsentDate DESC)
);


SELECT
    dp.UserID,
    dp.Activity,
    dp.ProcessedAt,
    dp.PerformedBy,
    cl.ConsentDate,
    cl.ConsentMethod,
    cl.Notes
FROM DataProcessingLog dp
JOIN ConsentLog cl ON dp.ConsentID = cl.ConsentID
ORDER BY dp.ProcessedAt DESC;
```

	UserID	Activity	ProcessedAt	PerformedBy	ConsentDate	ConsentMethod	Notes
1	1001	Email campaign - August 2025	2025-08-04 23:01:17.0300000	MarketingSystemApp	2025-08-04 23:01:13.3000000	Checkbox on signup form	User agreed to receive promotional emails

Figure 13.7: Showing audit data.

13.12.5 Summary

In this hands-on lab, you:

- Created GDPR-friendly tables to record user consent and processing activities
- Simulated real-world user consent scenarios
- Ensured every processing activity was tied back to an explicit consent

✅ This supports Article 5 and Article 7 of GDPR—accountability and lawful basis.

13.13 Conclusion

In this chapter, we explored how to implement GDPR and privacy regulations in SQL Server 2025. By leveraging features like Dynamic Data Masking, pseudonymization, data classification, and auditing, organizations can ensure compliance with data protection laws while maintaining the integrity and utility of their databases.

We also covered practical exercises to demonstrate how to fulfill data subject rights such as access, erasure, and portability. By following these guidelines and best practices, you can build a robust SQL Server environment that respects user privacy and adheres to legal requirements.

Section 6: Reporting and Exporting

14 Reporting and Data Connectivity

This chapter focuses on how to **extract business data** from SQL Server 2025 for reporting and visualization. You'll learn how to **export query results** to formats like **Excel and CSV**, and how to **connect SQL Server to Power BI** for building interactive dashboards. These skills are essential for data analysts, BI developers, and business users who rely on SQL Server as their reporting backbone.

14.1 Exporting SQL Data to Excel and CSV

SQL Server provides multiple options for exporting tabular data, including **SQL Server Management Studio (SSMS)** and **T-SQL utilities**.

14.1.1 Export Using SQL Server Management Studio (SSMS)

We can export data directly from SSMS using the following methods:

1. Run your query in SSMS.
2. Right-click on the result grid.
3. Select **Save Results As…**
4. Choose format: **CSV**, **Excel (*.xls, *.xlsx via copy/paste)**, or **Text (tab-delimited)**.
5. Save and distribute.

> *Best for one-time exports or manual reporting.*

14.1.2 xport Using SQL Server Import and Export Wizard

The SQL Server Import and Export Wizard provides a user-friendly interface for exporting data to various formats, including Excel and CSV.

Steps:

1. Right-click on database → **Tasks** → **Export Data**.

2. Choose:

    - **Data source**: SQL Server

  - **Destination**: Flat File Destination or Excel

3. Choose tables or query to export.

4. Define file path and schema mapping.

5. Run and save the package (optional).

> *Ideal for recurring exports or large datasets.*

14.1.3 Export via `bcp` (Bulk Copy Program) CLI

The `bcp` utility allows you to export data from SQL Server to a file using command-line operations. This is useful for automation and scripting.

```
bcp "SELECT * FROM Sales.Orders" queryout "orders.csv" -c -t, -S server -U username -P password
```

- `-c`: character format
- `-t,`: comma delimiter
- `-S`: server name
- `-U`, `-P`: login credentials

> *Use for automation and scripting.*

14.1.4 Export to CSV with T-SQL (via SSMS scripting)

You can also use T-SQL to export data to CSV format by leveraging the `bcp` utility or using `xp_cmdshell` to run command-line operations directly from SQL Server.

```
EXEC xp_cmdshell 'bcp "SELECT * FROM Sales.Customers" queryout "C:\Exports\customers.csv" -c -t,
```

> *Requires enabling `xp_cmdshell` and proper file permissions.*

14.2 Connecting Power BI to SQL Server

Power BI is a Microsoft analytics platform for building **interactive reports** and **dashboards** using data from SQL Server and other sources.

Power BI allows users to create visualizations, perform data modeling, and share insights across the organization. It supports both **Import** and **DirectQuery** modes for connecting to SQL Server databases.

The following steps outline how to connect Power BI Desktop to a SQL Server database:

1. Open Power BI Desktop.

2. Click **Home** → **Get Data** → **SQL Server**.

3. Enter:

    - **Server name**
    - **Database name** (optional)

4. Choose:

    - **Import** or **DirectQuery** mode
    - **Windows** or **SQL Server Authentication**

5. Click **OK**, then select tables or write SQL query.

We can choose between two modes when connecting Power BI to SQL Server:

Mode	Description
Import	Data copied into Power BI (.pbix) file
DirectQuery	Live queries run directly on SQL Server

> *Use **Import** for performance, **DirectQuery** for real-time data.*

To use a custom SQL query in Power BI:

1. Choose **Advanced Options** in Get Data.
2. Paste query:

```sql
SELECT Region, SUM(TotalAmount) AS Revenue
FROM Sales.Orders
GROUP BY Region;
```

> *Ideal for filtering or joining data before loading into Power BI.*

Once your Power BI report is published to the Power BI Service, you can set up **scheduled refreshes** to keep your data up-to-date. This ensures that your reports reflect the latest data from SQL Server without manual intervention.

14.3 Building Reports on Exported Data

Once you have exported data to CSV or Excel, you can use various tools to build reports and visualizations. Common options include:

- Open the CSV or Excel file in Power BI, Excel, or other tools.
- Use pivot tables, charts, slicers, or Power BI visuals to explore trends.

> *Use **Power BI datasets** to standardize reporting sources across the business.*

14.4 Practices for Reporting and Data Access

Following best practices for exporting and accessing data ensures that reports are efficient, secure, and maintainable. Here are some key practices:

Practice	Reason
Limit result sets when exporting	Prevent bloated exports and timeouts
Use views or stored procedures for Power BI	Encapsulate logic, improve reuse
Apply row-level filtering before export	Protect sensitive or irrelevant data
Enable auditing for BI access	Track who accessed or exported reports
Use parameters in Power BI queries	Support dynamic filtering and reuse

14.5 Exercise 39: Export Sales Summary to Excel

14.5.1 Description

Exporting SQL Server query results into Excel-compatible formats is a common requirement for reporting and analysis. In this hands-on lab, you'll create a sales summary query from the `AdventureWorks2022` database and export it to a `.csv` file using **SQL Server Management Studio (SSMS)** and **SQLCMD**, making it easy for stakeholders to open the file in Excel.

14.5.2 Objectives

By the end of this exercise, you will be able to:

- Write a sales summary query using `GROUP BY`
- Export query results from SSMS to `.csv`
- Export query results from SQLCMD to `.csv`

14.5.3 Prerequisites

- SQL Server 2025 instance with the `AdventureWorks2022` database restored
- SQL Server Management Studio (SSMS) 21.x installed
- `sqlcmd` available via terminal or Command Prompt
- Folder like `C:\Exports` available to store CSV files

14.5.4 Steps

Here's a step-by-step guide to complete this exercise:

14.5.4.1 Step 1: Write a Sales Summary Query in SSMS**

We'll summarize total sales by year using the `Sales.SalesOrderHeader` table. Use the `YEAR()` function and aggregate with `SUM()`.

```sql
USE AdventureWorks2022;
GO

SELECT
    YEAR(OrderDate) AS SalesYear,
    COUNT(*) AS OrderCount,
    SUM(TotalDue) AS TotalSales
FROM Sales.SalesOrderHeader
GROUP BY YEAR(OrderDate)
ORDER BY SalesYear;
```

✅ This returns annual sales totals, perfect for exporting.

14.5.4.2 Step 2: Export Results from SSMS as CSV

You can export this result directly from SSMS to a `.csv` file:

1. Open SSMS and run the query above.
2. Right-click the result grid and choose **Save Results As…**
3. Select file type **CSV (Comma delimited) (*.csv)**
4. Save it as: `C:\Exports\SalesSummary.csv`
5. Open the file in Excel to verify the format.

✅ This approach is quick and user-friendly for business analysts.

14.5.4.3 Step 3: Export Results Using SQLCMD (Command Line)

You can also automate exports using `sqlcmd`. Here's how:

```
sqlcmd -S localhost -d AdventureWorks2022 -E -Q "SET NOCOUNT ON; SELECT YEAR(OrderDate) AS Sales
```

Explanation:

- `-S localhost`: SQL Server instance
- `-d AdventureWorks2022`: database name
- `-E`: use Windows Authentication
- `-Q`: query
- `-s ","`: delimiter is comma
- `-o`: output file path

📌 This method is **suitable for automation or scheduling** using PowerShell or Task Scheduler.

14.5.4.4 Step 4: Verify and Open in Excel

Open `C:\Exports\SalesSummary_cmd.csv` in Excel. Check for:

- Correct columns and headers
- No extra messages (due to `SET NOCOUNT ON`)
- Proper comma-separated formatting

If required, you can add headers manually or adjust formatting within Excel.

14.5.5 Summary

In this exercise, you:

- Created a yearly sales summary query

- Exported results from SSMS to CSV
- Automated export using `sqlcmd` for command-line reporting
- Verified that outputs work seamlessly with Excel

✅ This enables business users and external systems to **consume SQL data without direct database access**.

14.6 Exercise 40: Connect SQL Server to Power BI Desktop for Dynamic Visualization

14.6.1 Description

Power BI enables users to analyze and visualize data interactively. This exercise demonstrates how to connect Power BI Desktop to your SQL Server 2025 instance and build a dynamic dashboard using data from the `AdventureWorks2022` database. You'll use Power BI's query editor and visual tools to explore and visualize sales data.

14.6.2 Objectives

By the end of this exercise, you will be able to:

- Connect Power BI Desktop to a SQL Server database
- Load and transform data using Power Query
- Create basic sales visualizations and summaries

14.6.3 Prerequisites

- SQL Server 2025 instance with `AdventureWorks2022` database restored (from Exercise 1)
- Power BI Desktop installed (latest version recommended)
- Access to `localhost` or your SQL Server instance
- Basic familiarity with Power BI UI

14.6.4 Steps

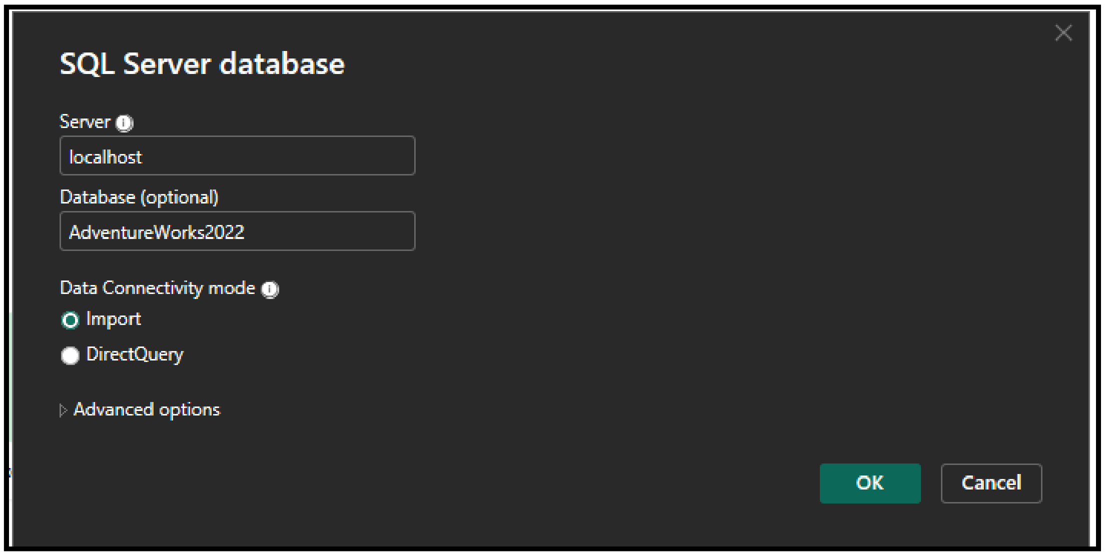
Here's a step-by-step guide to complete this exercise:

14.6.4.1 Step 1: Launch Power BI Desktop and Connect to SQL Server

1. Open **Power BI Desktop**.

2. On the **Home** tab, click **SQL Server** for data source.

3. Enter the server name (e.g., `localhost`) and database name `AdventureWorks2022`.

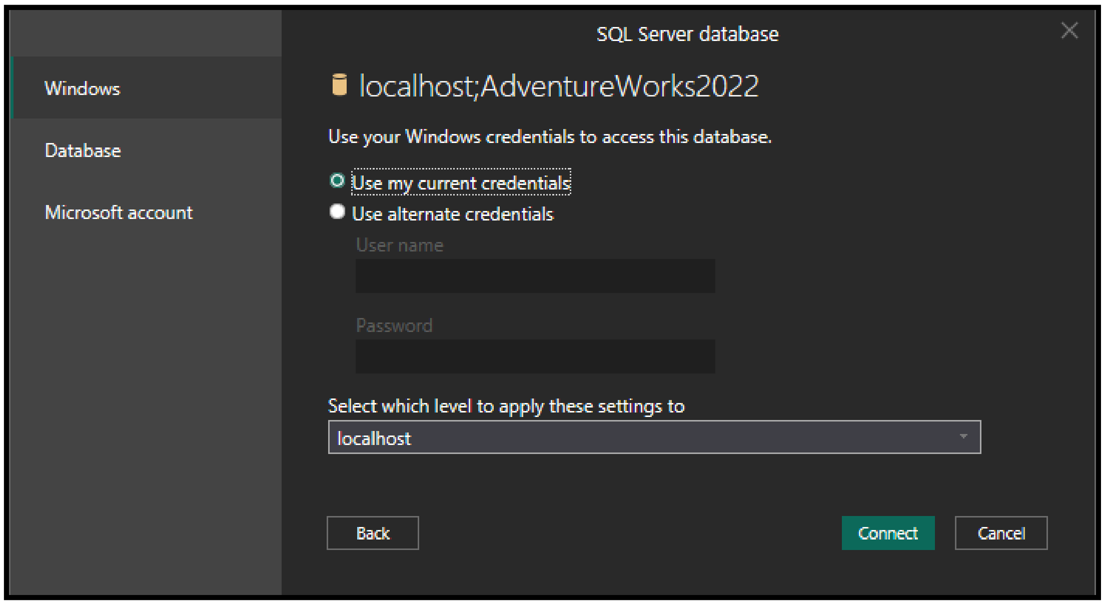


Server: localhost
Database (optional): AdventureWorks2022
Data Connectivity mode: [✓ Import] DirectQuery

Figure 14.1: Connecting to SQL Server.

4. Use either:

   - **Windows authentication**, or
   - **Database authentication** if needed
   - Select my current credentials if using Windows authentication



localhost;AdventureWorks2022
Use your Windows credentials to access this database.
[✓ Use my current credentials] Use alternate credentials
Select which level to apply these settings to: localhost

Figure 14.2: Selecting authentication on SQL Server.

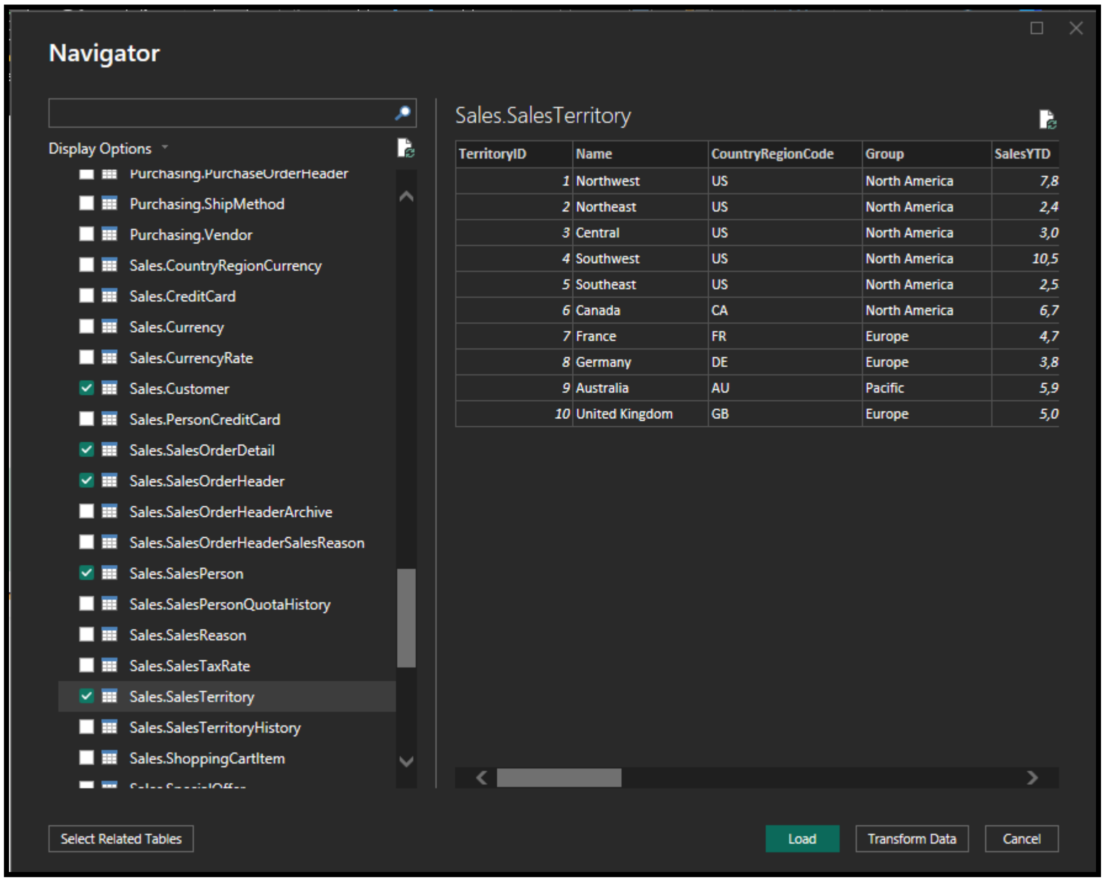
5. Click **OK** and allow Power BI to load available tables.

✅ This step sets up the live connection between Power BI and your SQL Server instance.

14.6.4.2 Step 2: Select and Load Sales Data

In the Navigator window:

1. Expand the `Sales` schema.
2. Select the following tables:
   - `SalesOrderHeader`
   - `SalesOrderDetail`
   - `Customer`
   - `Person`
   - `SalesTerritory`



Navigator

Display Options

- ☐ Purchasing.PurchaseOrderHeader
- ☐ Purchasing.ShipMethod
- ☐ Purchasing.Vendor
- ☐ Sales.CountryRegionCurrency
- ☐ Sales.CreditCard
- ☐ Sales.Currency
- ☐ Sales.CurrencyRate
- ✓ Sales.Customer
- ☐ Sales.PersonCreditCard
- ✓ Sales.SalesOrderDetail
- ✓ Sales.SalesOrderHeader
- ☐ Sales.SalesOrderHeaderArchive
- ☐ Sales.SalesOrderHeaderSalesReason
- ✓ Sales.SalesPerson
- ☐ Sales.SalesPersonQuotaHistory
- ☐ Sales.SalesReason
- ☐ Sales.SalesTaxRate
- ✓ Sales.SalesTerritory
- ☐ Sales.SalesTerritoryHistory
- ☐ Sales.ShoppingCartItem

Sales.SalesTerritory

TerritoryID	Name	CountryRegionCode	Group	SalesYTD
1	Northwest	US	North America	7,8
2	Northeast	US	North America	2,4
3	Central	US	North America	3,0
4	Southwest	US	North America	10,5
5	Southeast	US	North America	2,5
6	Canada	CA	North America	6,7
7	France	FR	Europe	4,7
8	Germany	DE	Europe	3,8
9	Australia	AU	Pacific	5,9
10	United Kingdom	GB	Europe	5,0

Figure 14.3: Selecting tables.

3. Click **Load** (or **Transform Data** if you want to clean the data before loading).

✅ Power BI loads the selected tables into the model for analysis.

14.6.4.3 Step 3: Build Relationships (if needed)

If Power BI doesn't detect relationships automatically:

1. Go to **Model View**.
2. Power BI may suggest relationships based on foreign keys.
3. If not, you can manually connect:
   - `SalesOrderHeader.CustomerID` → `Customer.CustomerID`
   - `Customer.PersonID` → `Person.BusinessEntityID`
   - `SalesOrderHeader.SalesTerritoryID` → `SalesTerritory.TerritoryID`
   - `SalesOrderDetail.SalesOrderID` → `SalesOrderHeader.SalesOrderID`

✅ Establishing relationships allows Power BI to perform cross-table analysis.

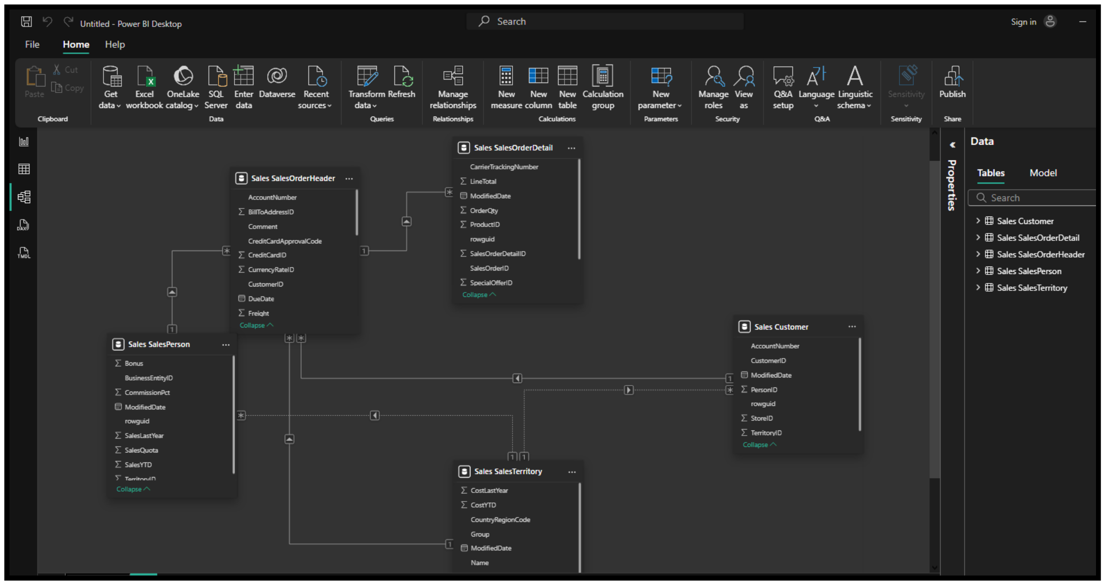


Figure 14.4: Power BI detects model relationship.

14.6.4.4 Step 4: Create a Sales Dashboard

Now let's visualize sales trends:

1. In **Report View**, insert a **Bar Chart**:
   - Axis: `SalesTerritory.Name`
   - Value: `SalesOrderHeader.TotalDue (Sum)`

- Title: “Total Sales by Territory”

2. Insert a **Line Chart**:
   - Axis: `SalesOrderHeader.OrderDate (Month)`
   - Value: `SalesOrderHeader.TotalDue (Sum)`
   - Title: “Sales Trend Over Time”

3. Insert a **Card**:
   - Field: `SalesOrderHeader.TotalDue`
   - Aggregation: `Sum`
   - Title: “Total Revenue”

✅ You’ve created an interactive dashboard from SQL Server data.

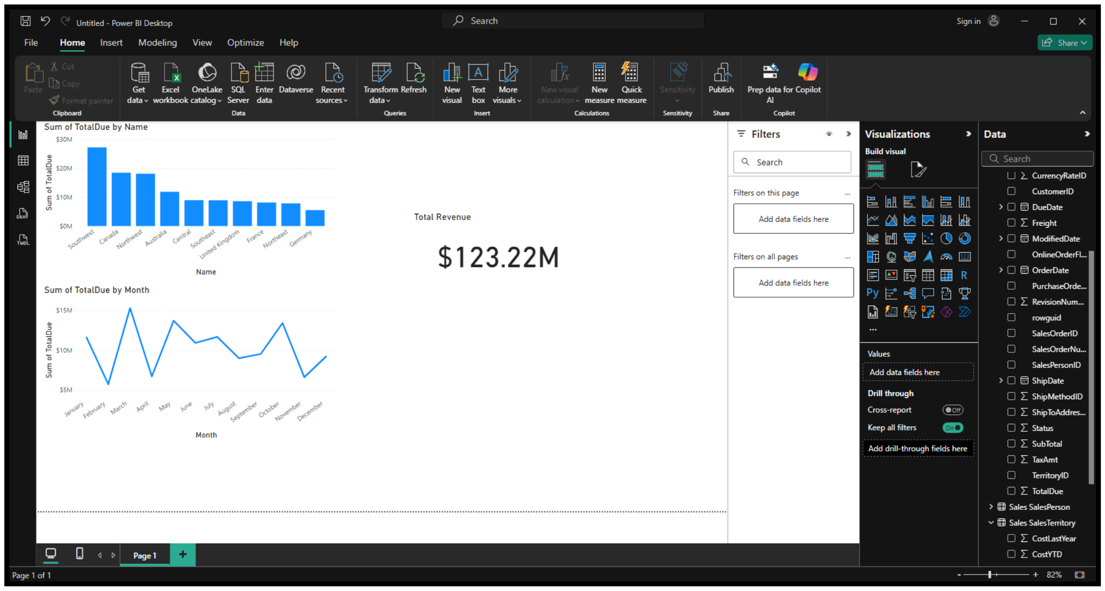


Figure 14.5: Data visualization on Power BI.

14.6.4.5 Step 5: Save and Refresh Data

1. Save your Power BI report as `AdventureWorksSalesReport.pbix`.
2. Click **Refresh** to reload data from the SQL Server source.

Optional:

- Set up **Scheduled Refresh** (requires Power BI Pro or Premium service + gateway).

14.6.5 Summary

In this exercise, you:

- Connected Power BI Desktop to SQL Server 2025
- Loaded and modeled data from the `AdventureWorks2022` database
- Created a dynamic report with charts and KPIs
- Enabled refresh to keep data synchronized

✅ This allows you to empower business users with self-service BI directly from trusted SQL Server data.

14.7 Conclusion

In this chapter, we explored how to export SQL Server data for reporting and visualization, focusing on practical methods using SSMS, `bcp`, and Power BI. By mastering these techniques, you can effectively share insights and build interactive dashboards that drive business decisions.

Appendix A: T-SQL Cheatsheet (SQL Server 2025)

This cheatsheet provides a quick reference to common T-SQL commands and concepts in SQL Server 2025. It covers database operations, data manipulation, filtering, functions, aggregations, joins, subqueries, transactions, stored procedures, window functions, and security.

1. Database and Table Operations

Create Database

```
CREATE DATABASE SalesDB;
```

Use Database

```
USE SalesDB;
```

Create Table

```
CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY,
    FullName NVARCHAR(100),
    Email NVARCHAR(100),
    CreatedAt DATETIME DEFAULT GETDATE()
);
```

Alter Table

```
ALTER TABLE Customers ADD Phone NVARCHAR(20);
```

Drop Table

```
DROP TABLE Customers;
```

2. Data Manipulation (CRUD)

Insert

```sql
INSERT INTO Customers (CustomerID, FullName, Email)
VALUES (1, 'John Doe', 'john@example.com');
```

Update

```sql
UPDATE Customers
SET Email = 'john.doe@example.com'
WHERE CustomerID = 1;
```

Delete

```sql
DELETE FROM Customers
WHERE CustomerID = 1;
```

Select

```sql
SELECT * FROM Customers;
```

3. Filtering and Sorting

WHERE Clause

```sql
SELECT * FROM Customers
WHERE Email LIKE '%@gmail.com';
```

ORDER BY

```sql
SELECT * FROM Customers
ORDER BY CreatedAt DESC;
```

4. Functions

String

```sql
SELECT UPPER('hello'), LEN('hello world');
```

Date/Time

```sql
SELECT GETDATE(), DATEPART(YEAR, GETDATE());
```

Mathematical

```sql
SELECT ROUND(123.4567, 2), ABS(-42);
```

5. Aggregations and Grouping

```sql
SELECT COUNT(*) AS TotalCustomers
FROM Customers;

SELECT YEAR(CreatedAt) AS Year, COUNT(*) AS Count
FROM Customers
GROUP BY YEAR(CreatedAt);
```

6. Joins

INNER JOIN

```sql
SELECT o.OrderID, c.FullName
FROM Orders o
INNER JOIN Customers c ON o.CustomerID = c.CustomerID;
```

LEFT JOIN

```sql
SELECT c.FullName, o.OrderID
FROM Customers c
LEFT JOIN Orders o ON c.CustomerID = o.CustomerID;
```

7. Subqueries and CTE

Subquery

```sql
SELECT * FROM Orders
WHERE CustomerID IN (SELECT CustomerID FROM Customers WHERE FullName LIKE 'J%');
```

CTE

```sql
WITH RecentOrders AS (
    SELECT TOP 10 * FROM Orders ORDER BY OrderDate DESC
)
SELECT * FROM RecentOrders;
```

8. Transactions

```sql
BEGIN TRANSACTION;

UPDATE Accounts
SET Balance = Balance - 100
WHERE AccountID = 1;

UPDATE Accounts
SET Balance = Balance + 100
WHERE AccountID = 2;
```

```
COMMIT;
-- or use ROLLBACK;
```

9. Stored Procedures and Functions

Stored Procedure

```
CREATE PROCEDURE GetCustomerByID @ID INT
AS
BEGIN
    SELECT * FROM Customers WHERE CustomerID = @ID;
END;
```

Scalar Function

```
CREATE FUNCTION dbo.GetYear (@date DATETIME)
RETURNS INT
AS
BEGIN
    RETURN YEAR(@date);
END;
```

10. Window Functions

```
SELECT CustomerID, OrderDate,
       RANK() OVER (PARTITION BY CustomerID ORDER BY OrderDate DESC) AS OrderRank
FROM Orders;
```

11. Security and Users

```
CREATE LOGIN user1 WITH PASSWORD = 'StrongP@ssword!';
CREATE USER user1 FOR LOGIN user1;
GRANT SELECT ON Customers TO user1;
```

Appendix B: Resources

SQL Server 2025 High Availability & Disaster Recovery: Always On Solutions Course

Dive into the world of SQL Server 2025 with our comprehensive Udemy course, **“SQL Server 2025: Build Always On HA & DR Solutions.”** This course is designed for database administrators and IT professionals who want to master high availability (HA) and disaster recovery (DR) solutions using the latest features of SQL Server 2025.

What You’ll Learn

In this course, you will learn to:

- Understand HA and DR concepts in SQL Server 2025
- Build and configure Windows Server Failover Clustering (WSFC)
- Deploy Always On Availability Groups from scratch
- Set up and manage the AG Listener for client connections
- Configure read-only routing for reporting and BI workloads
- Offload backups using Preferred Backup Replica
- Perform failover testing: automatic, manual, and forced
- Monitor and troubleshoot AG health
- Integrate real-world ASP.NET Core apps with AG Listener
- Apply best practices for performance and uptime

100% Hands-On with Real Labs

This course is not just theory. You’ll build your own lab environment using virtual machines and simulate real-world HA/DR use cases.

We guide you through every step — from cluster setup to full availability group testing. Whether you’re creating an AG with two replicas or

Enhance Your Learning with Our Udemy Course

Enroll today: *Red Hat NGINX Web Server: Publishing and Deploying Web Apps* https://www.udemy.com/course/rhel-nginx/?referralCode=C9CFA39AE9E332ADA9FB

Build Secure PHP APIs Like a Pro with Laravel 12, OAuth2, and JWT

Unlock the full potential of **Laravel 12** for REST API development! This hands-on course on Udemy teaches you how to build **robust, secure, and modern APIs** using Laravel, MySQL, OAuth2, JWT, Sanctum, and Role-Based Access Control (RBAC). Perfect for real-world applications and 2025 standards.

🚀 Highlight Topics

- What's New in Laravel 12 for API development
- Build RESTful APIs from scratch (Hello World to full CRUD)
- File upload and user data handling via REST API
- Secure authentication with **Sanctum**, **JWT**, and **OAuth2**
- Role-Based Access Control (RBAC) with middleware
- Legacy support: Laravel 8, 7.x, and 6.x projects included
- Real project codebases and testing tutorials

👨‍💻 Who Should Enroll?

- Laravel developers aiming to modernize their API skills
- Backend engineers securing APIs with token-based auth
- Teams migrating legacy Laravel APIs to newer standards
- Students and professionals building real-world Laravel apps
- Anyone preparing for backend development roles in 2025

> ***Future-proof your Laravel skills.*** *This course gives you* ***everything you need*** *to build secure, scalable, and professional REST APIs in Laravel 12. Learn by doing — with real code, live tests, and full project coverage.*

👉 **Join now and start building APIs that meet today's security demands.** *PHP REST API: Laravel 12, MySQL, OAuth2, JWT, Roles-Based* https://www.udemy.com/course/phprestapi/?referralCode=2C5B2F14100B499E9845

Master Real-World Logging & Visualization with the Full ELK Stack

Take control of your logging, search, and monitoring pipeline with this **hands-on Udemy course** covering Elasticsearch, Logstash, Kibana, and Beats. Learn how to set up, ingest, visualize, and scale log data using practical projects — all designed for developers, sysadmins, and DevOps engineers in **real production environments**.

🚀 Highlight Topics

- Cross-platform installation: Windows, Ubuntu, macOS, Docker
- Elasticsearch REST API: CRUD, mapping, queries, aggregation, SQL, geo fields
- Real-world API integration: PHP, ASP.NET Core, Node.js, Python
- Logstash ingestion: files, folders, and RDBMS (MySQL)
- Kibana Lens visualizations: charts, maps, dashboards, Canvas
- Beats agents: Filebeat, Winlogbeat, Metricbeat, Packetbeat, Heartbeat, Auditbeat
- High Availability (HA) setup for Elasticsearch and Kibana with Nginx

👨‍💻 Who Should Enroll?

- Developers and DevOps engineers building log-driven applications
- System administrators responsible for monitoring and observability
- Backend/API developers seeking integration with Elasticsearch
- Cybersecurity analysts and IT ops engineers using ELK for log auditing
- Teams adopting open-source observability tools for modern infrastructure

> ***Log smarter, visualize better, and scale with confidence.*** *Whether you're just getting started or already managing production systems, this course gives you everything you need to build and operate a* ***powerful ELK Stack pipeline****. With real-world use cases, cross-platform setups, and step-by-step guidance, you'll go beyond the basics and into expert territory.*

👉 **Enroll today** to master the ELK Stack and unlock actionable insights from your data! *Practical Full ELK Stack: Elasticsearch, Kibana and Logstash* https://www.udemy.com/course/elkstack/?referralCode=863C1036F77169C975C5

Appendix C: Source Code

You can download the source code files for this book from GitHub at https://www.github.com/agusk/ilmudata-book-sqlserver.

About

Agus Kurniawan's journey in the field of technology, spanning from 2001, is a remarkable blend of deep technical expertise and a fervent passion for sharing knowledge. As a seasoned professional, Agus has carved a niche in diverse technological domains, including software development, IoT (Internet of Things), Machine Learning, IT infrastructure, and DevOps. His experiences are not just limited to developing cutting-edge solutions but also extend to shaping the future of upcoming technologists through training and workshops.

Agus's career is marked by significant contributions to both technological innovation and community development. His recognition as a Microsoft Most Valuable Professional (MVP) from 2004 to 2022 underlines his proficiency in Microsoft technologies and his dedication to educating others. Agus has been at the forefront of delivering various training sessions and workshops, sharing his insights and helping others grow in the ever-evolving tech industry.

Mastering Business Data with SQL
A Practical Guide to Querying, Modeling, and Compliance Using SQL Server 2025

This book is crafted for professionals and learners aiming to master business data analysis and management using SQL Server 2025. Drawing on Agus Kurniawan's extensive experience in software engineering and data technologies, it delivers practical techniques, real-world scenarios, and best practices for querying, modeling, and ensuring compliance with business data.

Agus invites readers to share feedback, questions, and suggestions. If you have inquiries about SQL, ideas for future editions, or wish to discuss your learning experiences, please get in touch.

For those seeking private or group training on SQL, data management, or related technologies, Agus offers customized programs for individuals and organizations. Contact him for details on available topics, schedules, and formats.

Email: aguskur@hotmail.com, agusk2007@gmail.com

LinkedIn: linkedin.com/in/agusk

Twitter: [@agusk2010]