CopperpotLab · revenue-by-restaurant.sql

```
SELECT r.name, SUM(l.quantity * l.unit_price) AS revenue
FROM dbo.orders AS o
JOIN dbo.order_lines AS l ON l.order_id = o.order_id
JOIN dbo.restaurants AS r ON r.restaurant_id = o.restaurant_id
WHERE o.status = 'completed'
GROUP BY r.name ORDER BY revenue DESC;
```

name	revenue
Copperpot Uptown	100.50
Copperpot Harbor	53.50
Copperpot Riverside	48.00

```
Execution plan
Sort ← Stream Aggregate ← Compute Scalar ← Nested Loops
  ← Clustered Index Scan (restaurants, orders) · Seek (order_lines)
```

FROM FIRST QUERY TO PRODUCTION

Transact-SQL for SQL Server

Queries, stored procedures, indexes, execution plans, and security

Every example run on SQL Server 2022 · SSMS · Azure SQL notes

JOINS · WINDOW FUNCTIONS · PROCEDURES · TUNING

THENEXBOOKS

Transact-SQL for SQL Server

From Your First Query to Stored Procedures, Performance Tuning, and Production

Queries · Window Functions · Procedures · Indexes · Execution Plans · Security

First Edition

THENEXBOOKS

Transact-SQL for SQL Server
From Your First Query to Stored Procedures, Performance Tuning, and Production

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. SQL Server features vary by version and edition; always consult the Microsoft documentation for the version you use. Test every change in a non-production environment first.

Microsoft, SQL Server, Azure, and SQL Server Management Studio are trademarks of the Microsoft group of companies. Other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by Microsoft Corporation.

Copperpot Kitchens, the running example in this book, is fictional, as are all restaurants, customers, staff, and orders in its examples. Any resemblance to real companies or people is coincidental.

Preface

Behind countless business applications sits a SQL Server database, and the language that talks to it is Transact-SQL. T-SQL is how you ask questions of your data, change it safely, enforce its rules, and package logic into procedures and functions that applications can call. It is also how you make that work fast — and how you keep it secure and recoverable when real users and real money depend on it.

Writing a first query takes minutes. Writing T-SQL that stays correct with NULLs and duplicates, performs well when tables grow from thousands to millions of rows, survives concurrent users without deadlocks, and can be deployed and tested like any other code takes understanding. That understanding is what this book aims to give you.

We start from the very first `SELECT` and build steadily: filtering and sorting, data types and functions, joins, aggregation, subqueries and CTEs, window functions, and set operations. Then we design tables and modify data safely with transactions, write stored procedures, functions, and triggers with proper error handling, and finish with indexing, execution plans, query tuning, concurrency, security, backup and restore, and database DevOps. Every example in the book was run against SQL Server 2022, and the results you see are the results it produced.

Who This Book Is For

- **Beginners** who want a complete, practical path from first query to professional T-SQL.
- **Application developers** (C#, Java, Python, JavaScript) who work with SQL Server and want to write better queries and procedures.
- **Data and reporting analysts** who want to master joins, aggregation, and window functions.
- **Developers taking on database responsibilities** — performance, security, deployments, and maintenance.

What You Will Learn

- How to set up SQL Server, SQL Server Management Studio, and the lab database used throughout the book.
- Querying fundamentals, data types, NULL logic, and built-in functions for strings, dates, and conversion.
- Joins, grouping, subqueries, CTEs, `APPLY`, window functions, set operators, pivoting, and recursion.
- Table design with keys and constraints; `INSERT`, `UPDATE`, `DELETE`, and `MERGE` with `OUTPUT`.
- Transactions, isolation levels, locking, views, schemas, temporal tables, JSON, and XML.
- Variables and control flow, stored procedures, user-defined functions, error handling, triggers, and safe dynamic SQL.

- Indexes, execution plans, query tuning, statistics, parameter sniffing, Query Store, blocking, and deadlocks.
- Security with roles and row-level security, backup and restore, and source control, migrations, and testing.

How This Book Is Organized

Part I — Foundations (Chapters 1–5) sets up the tools and teaches querying basics, data types, and functions. **Part II — Querying in Depth** (Chapters 6–10) covers joins, aggregation, subqueries, window functions, and advanced query shapes. **Part III — Modifying Data and Designing Tables** (Chapters 11–15) builds and changes data safely. **Part IV — Programming T-SQL** (Chapters 16–20) writes procedures, functions, error handling, and triggers. **Part V — Performance** (Chapters 21–24) makes queries fast and concurrent workloads healthy. **Part VI — Production and Case Study** (Chapters 25–28) secures, protects, and deploys the database and brings everything together. The appendices provide a cheat sheet, a function and template reference, a production checklist, and a glossary.

Each chapter ends with practice exercises and a **Key Takeaways** box for quick review.

The Running Example: Copperpot Kitchens

Throughout the book we work with the database of **Copperpot Kitchens**, a fictional group of three restaurants. It has a menu, customers with loyalty tiers, staff with a management hierarchy, and orders with line items — including walk-in orders with no customer, a cancelled order, an inactive menu item, and a customer with no email address. The data is small enough to check every result by eye, yet it exercises everything a real order system needs. Chapter 2 gives you the complete script to create it, so you can run every example yourself.

Versions and Conventions

Examples were run on SQL Server 2022 and work in the free Developer and Express editions. Where a feature appeared in a specific version, we say so; where Azure SQL Database differs, we note it. Code appears in `monospace`, query results are shown as tables, and messages SQL Server returns are shown in blocks labeled **Messages**.

> **TIP**
> A recommended practice or a technique that makes your work easier.

> **NOTE**
> Background information, version details, or a useful aside.

WARNING

A common mistake that causes wrong results, data loss, slow queries, or security holes.

Contents

Preface

PART I — FOUNDATIONS

1 What T-SQL Is: SQL Server, Azure SQL, and the Tools

2 Setting Up: SQL Server Developer and Express, SSMS, and the Copperpot Lab Database

3 Querying Basics: SELECT, WHERE, ORDER BY, TOP, and OFFSET-FETCH

4 Data Types, NULLs, and Expressions

5 Built-in Functions: Strings, Dates, Conversion, and Logic

PART II — QUERYING IN DEPTH

6 Joins: INNER, OUTER, CROSS, and Self Joins

7 Grouping and Aggregation: GROUP BY, HAVING, ROLLUP, CUBE, and GROUPING SETS

8 Subqueries, Derived Tables, CTEs, and APPLY

9 Window Functions: Ranking, Aggregates, Offsets, and Frames

10 Set Operators, PIVOT and UNPIVOT, and Recursive CTEs

PART III — MODIFYING DATA AND DESIGNING TABLES

11 Creating Tables: Data Types, Keys, Constraints, IDENTITY, and Sequences

12 INSERT, UPDATE, DELETE, and MERGE with the OUTPUT Clause

13 Transactions, Isolation Levels, and Locking

14 Views, Schemas, and Synonyms

15 Temporal Tables, JSON, and XML

PART IV — PROGRAMMING T-SQL

16 Variables, Batches, and Control Flow

17 Stored Procedures

18 User-Defined Functions

19 Error Handling: TRY...CATCH, THROW, and XACT_ABORT

20 Triggers and Dynamic SQL

PART V — PERFORMANCE

21 Indexes: Clustered, Nonclustered, Covering, Filtered, and Columnstore

22 Reading Execution Plans

23 Query Tuning: Sargability, Statistics, Parameter Sniffing, and Query Store

24 Concurrency in Practice: Blocking, Deadlocks, and Row Versioning

PART VI — PRODUCTION AND CASE STUDY

25 Security: Logins, Users, Roles, Permissions, and Row-Level Security

26 Backup, Restore, and Maintenance for Developers

27 Database DevOps: Source Control, Migrations, and Testing with tSQLt

28 Case Study: Building the Copperpot Kitchens Order Platform

APPENDICES

A T-SQL Cheat Sheet

B Function and Template Reference

C Production Checklist

D Glossary

PART I

Foundations

What T-SQL is and the tools around it, a working SQL Server with the Copperpot lab database, and the fundamentals of querying, data types, NULLs, and built-in functions.

1 · What T-SQL Is
2 · Setting Up
3 · Querying Basics
4 · Data Types and NULLs
5 · Built-in Functions

CHAPTER 1 · PART I — FOUNDATIONS

What T-SQL Is: SQL Server, Azure SQL, and the Tools

Behind almost every order screen, booking form, and monthly report sits a relational database waiting for a question. This chapter explains what Transact-SQL is, where it runs, how SQL Server thinks about your requests, and which tools you will use to talk to it for the rest of this book.

Part I at a Glance

Part I is your foundation. Over five chapters you will learn what T-SQL is and where it runs (this chapter), how to install SQL Server and build the practice database used throughout the book (Chapter 2), how to ask your first real questions with `SELECT`, filtering, sorting, and paging (Chapter 3), how SQL Server stores numbers, text, dates, and missing values (Chapter 4), and finally how to use the large library of built-in functions that turn raw columns into useful answers (Chapter 5). You do not need any previous database experience. If you can use a spreadsheet and you are comfortable typing a few lines of text and pressing a key to run them, you have every skill this part assumes.

We take a steady, deliberate pace in these first chapters. Databases reward a firm foundation: a few minutes spent understanding why a query returns rows in no particular order, or why a missing value is not the same as zero, will save you hours of confusion later when you are working with real data and real deadlines. The ideas in Part I are small, but everything else in the book is built on them.

Databases, Tables, and Relational Thinking

A **database** is an organized collection of data that a program can store, search, and change safely, even when many people use it at the same time. The most widely used kind is the **relational database**, in which data lives in **tables**. A table looks a lot like a spreadsheet: it has named **columns** (also called attributes) and a set of **rows** (also called records). Each column has a fixed data type, so a price column holds only numbers and a date column holds only dates. Each row describes one thing: one restaurant, one customer, one order.

The word "relational" comes from mathematics, where a **relation** is a set of tuples that share the same attributes. You do not need the mathematics to use SQL Server, but two consequences of it matter every day. First, a table is a *set* of rows, and sets have no built-in order. If you want rows in a particular sequence, you must ask for it explicitly; we return to this in Chapter 3. Second, tables are connected by values, not by physical pointers. An order row stores the number of the restaurant where

it was placed, and a query matches that number to the restaurant table whenever it needs the restaurant's name. This simple idea, linking tables through shared values called **keys**, lets a database avoid storing the same fact in many places.

A **relational database management system** (RDBMS) is the software that stores those tables, enforces rules about them, and answers questions about them. Microsoft SQL Server is an RDBMS. So are PostgreSQL, MySQL, Oracle Database, and many others. They all speak a common language for describing and querying data, and that language is SQL.

SQL and Transact-SQL

SQL (Structured Query Language, pronounced either "S-Q-L" or "sequel") is the standard language for working with relational databases. It is standardized by ISO and ANSI, and every major RDBMS implements a large part of that standard. SQL is a **declarative** language: you describe *what* result you want, and the database engine decides *how* to produce it. You say "give me the total revenue for each restaurant", not "open this file, read row one, add it to a running total, read row two". That division of labor is powerful. The engine knows how the data is physically stored, which indexes exist, and how much memory is available, so it can pick a far better strategy than most of us would write by hand.

Transact-SQL, almost always shortened to **T-SQL**, is Microsoft's dialect of SQL. It is the language spoken by SQL Server and by the Azure SQL family of cloud services. T-SQL includes standard SQL (`SELECT`, `INSERT`, `UPDATE`, `DELETE`, `CREATE TABLE`, and so on) and adds a set of extensions that turn it into a complete programming language: variables, `IF` and `WHILE` for control flow, `TRY...CATCH` for error handling, stored procedures, user-defined functions, triggers, and a rich library of built-in functions. Every statement you run against SQL Server, whether you type it yourself or an application generates it, is T-SQL.

Because T-SQL follows the standard closely for everyday queries, most of what you learn here transfers to other databases. The differences show up in the details: the names of functions, the way you limit the number of rows returned, how dates are handled, and the procedural extensions. Where T-SQL behaves differently from the standard or from common practice elsewhere, we point it out.

A first look at T-SQL

Here is a complete T-SQL query. It reads the restaurants table from the practice database you will build in Chapter 2 and returns three columns, oldest restaurant first. Do not worry about running it yet; just read it as English.

Query

```
SELECT name, city, opened_on
FROM dbo.restaurants
ORDER BY opened_on;
```

Results

name	city	opened_on
Copperpot Harbor	Seabrook	2019-04-12
Copperpot Uptown	Seabrook	2021-09-03
Copperpot Riverside	Millbrook	2023-02-17

The query says: select the `name`, `city`, and `opened_on` columns, from the table `dbo.restaurants`, ordered by the opening date. The prefix `dbo` is a **schema**, a named container for tables and other objects inside a database; `dbo` (short for "database owner") is the default schema. Chapter 14 explores schemas in depth. The semicolon ends the statement. T-SQL does not always require it, but some newer statements do, and writing it consistently is a good habit that we follow throughout the book.

Declarative power

The real strength of a declarative language appears when a question involves several tables. The next query computes revenue from completed orders for each restaurant. It combines three tables, filters out cancelled orders, groups the rows by restaurant, adds up quantity times price, and sorts the answer. You will learn every piece of it in Chapters 3, 6, and 7; for now, notice how compact it is compared with the work it describes.

Query

```
SELECT r.name AS restaurant,
       SUM(ol.quantity * ol.unit_price) AS revenue
FROM dbo.orders AS o
JOIN dbo.order_lines AS ol ON ol.order_id = o.order_id
JOIN dbo.restaurants AS r ON r.restaurant_id = o.restaurant_id
WHERE o.status = 'completed'
GROUP BY r.name
ORDER BY revenue DESC;
```

Results

restaurant	revenue
Copperpot Uptown	100.50
Copperpot Harbor	53.50
Copperpot Riverside	48.00

Nowhere did we say which table to read first, whether to use an index, or how to add up the numbers. SQL Server's **query optimizer** worked that out. The optimizer is the part of the engine that considers many possible strategies, called **execution plans**, estimates their cost, and picks a cheap one. Chapter 22 teaches you to read those plans, and Chapter 23 shows how to help the optimizer when it picks badly.

Procedural extensions

T-SQL also lets you write step-by-step logic when you need it. The following **batch** (a group of statements sent to the server together) declares two variables, stores a count in one of them, and prints a message depending on the result.

Query

```
DECLARE @cutoff date = '2021-01-01';
DECLARE @count int;
SELECT @count = COUNT(*) FROM dbo.restaurants WHERE opened_on >= @cutoff;
IF @count > 1
    PRINT CONCAT(@count, ' restaurants opened on or after ', @cutoff, '.');
ELSE
    PRINT 'Only one recent restaurant.';
```

Messages

```
2 restaurants opened on or after 2021-01-01.
```

Variables start with `@`, `DECLARE` creates them, `IF...ELSE` chooses a branch, and `PRINT` sends a text message back to the client rather than a result set. Notice that the output appears under *Messages*, not *Results*. SQL Server sends two kinds of output back to you: **result sets** (rows and columns) and **messages** (informational text, row counts, warnings, and errors). Every tool you use displays both, usually in separate tabs or sections. Chapter 16 covers variables, batches, and control flow properly.

> **NOTE**
> Write set-based queries first and procedural code second. Beginners coming from languages such as Python, Java, or C# often reach for loops that process one row at a time. In T-SQL, a single statement that works on a whole set of rows is almost always simpler and much faster. Use procedural code to orchestrate set-based statements, not to replace them.

The Families of T-SQL Statements

T-SQL statements are traditionally grouped into families according to what they do. Knowing the families helps you read documentation and understand which permissions a statement needs.

Family	Purpose	Examples	Covered in
DQL (data query)	Read data	`SELECT`	Chapters 3–10
DML (data manipulation)	Change rows	`INSERT`, `UPDATE`, `DELETE`, `MERGE`	Chapter 12
DDL (data definition)	Create or change objects	`CREATE`, `ALTER`, `DROP`	Chapters 11, 14

DCL (data control)	Grant or remove permissions	`GRANT`, `REVOKE`, `DENY`	Chapter 25
TCL (transaction control)	Group changes into units	`BEGIN TRAN`, `COMMIT`, `ROLLBACK`	Chapter 13

Many writers fold `SELECT` into DML. The exact labels matter less than the idea behind them.

How SQL Server Processes a Query

When you press Execute, your text travels from the client tool to the server over a network protocol called **TDS** (Tabular Data Stream), even when the server is on the same computer. Inside SQL Server, the request passes through several stages:

1. **Parsing.** The text is checked for correct syntax and turned into an internal tree. A misspelled keyword fails here.
2. **Binding.** Names are resolved to real objects and columns, and data types are worked out. A misspelled table name fails here.
3. **Optimization.** The optimizer builds candidate execution plans and picks one, using statistics about the data. Plans are cached so that repeated queries can skip this step.
4. **Execution.** The storage engine reads and writes data pages, takes locks, and streams results back to the client.

You can see the binding stage in action by misspelling a table name. Our table is called `restaurants`, plural.

Query

```
SELECT name FROM dbo.restaurant;
```

Messages

```
Msg 208, Level 16, State 1, Line 1
Invalid object name 'dbo.restaurant'.
```

Every SQL Server error has the same anatomy. The **message number** (208) identifies the error and is what you search for in documentation. The **severity level** (16) indicates how serious it is: levels 11 to 16 are errors the user can fix, such as a bad name or a broken rule, while levels of 20 and above indicate serious problems and usually end the connection. The **state** gives Microsoft's engineers a hint about where in the code the error was raised, and the **line** number points to the line inside the batch. Read the message text carefully; SQL Server's error messages are usually precise about what went wrong. Chapter 19 shows how to catch and handle errors in code.

> **TIP**
>
> When a batch fails, double-click the error in the SSMS Messages tab. SSMS jumps to the offending line in your editor, which is much faster than counting lines yourself in a long script.

Logical query processing order

One more idea about processing will save you a great deal of confusion. You write the clauses of a query in this order: `SELECT`, `FROM`, `WHERE`, `GROUP BY`, `HAVING`, `ORDER BY`. But SQL Server *logically* evaluates them in a different order:

1. `FROM` (and joins): which rows are available?
2. `WHERE`: which rows survive the filter?
3. `GROUP BY`: how are rows combined into groups?
4. `HAVING`: which groups survive?
5. `SELECT`: which expressions are returned, and with which names?
6. `ORDER BY`: in what order are rows presented?
7. `TOP` or `OFFSET-FETCH`: how many rows are returned?

This is called **logical query processing**. It explains several rules that otherwise look arbitrary. For example, you cannot use a column alias defined in `SELECT` inside `WHERE`, because `WHERE` is evaluated first and the alias does not exist yet. But you *can* use that alias in `ORDER BY`, because `ORDER BY` comes after `SELECT`. That is why the revenue query above could say `ORDER BY revenue DESC`. The optimizer is free to execute the query physically in any order it likes, as long as the result is the same as this logical order would produce. Chapter 3 returns to logical processing with concrete examples.

Where T-SQL Runs: The SQL Server Family

T-SQL is not tied to a single product. The same language, with small differences, runs in several places. Choosing among them is usually a matter of who manages the servers and how much control you need.

SQL Server (the box product)

SQL Server is the product you install on your own Windows or Linux machine, in a virtual machine, or in a container. Microsoft releases a new major version every few years: SQL Server 2016, 2017, 2019, and 2022 are the versions you are most likely to meet in production today, and this book is written and tested against **SQL Server 2022**. Each version is identified internally by a major version number; SQL Server 2022 is version 16. Features introduced after SQL Server 2022 are marked as such whenever we mention them, and you should check the documentation for them.

SQL Server comes in several **editions** that share the same engine but differ in licensing, limits, and features:

- **Enterprise** includes every feature and is licensed for the largest production workloads.
- **Standard** covers most production needs for small and medium businesses, with limits on memory and processor cores.
- **Developer** has the full Enterprise feature set but is licensed only for development and testing, not production. It is free.
- **Express** is free for production use but limited: small maximum database size, limited memory and processor use, and no SQL Server Agent job scheduler. It is ideal for learning, small applications, and this book.

Chapter 2 helps you choose between Developer and Express and walks you through installing one of them. The T-SQL language itself is the same in every edition; only some features and limits differ.

Azure SQL Database

Azure SQL Database is a fully managed cloud database service. Microsoft runs the servers, applies patches, takes backups, and keeps the service highly available; you create a database and connect to it. It runs the same database engine as SQL Server, usually ahead of the boxed product in features, and it speaks T-SQL. Because you get a database rather than a whole server, some instance-level features behave differently or are unavailable: for example, you cannot switch databases with `USE` inside a connection, cross-database queries work differently, and there is no SQL Server Agent. Whenever a topic in this book behaves differently in Azure SQL, we say so, and you should check the documentation for Azure SQL for the details.

Azure SQL Managed Instance and SQL Server on virtual machines

Azure SQL Managed Instance sits between the two: Microsoft manages the platform, but you get something very close to a complete SQL Server instance, including SQL Server Agent and cross-database queries. It is a common target when moving existing applications to the cloud with few changes. Finally, you can simply run **SQL Server on an Azure virtual machine** (or on any cloud provider's virtual machines), which gives you full control and full responsibility.

For learning T-SQL, the differences between these options barely matter. Everything in Parts I to IV of this book works the same way on all of them. The differences appear mostly in administration topics such as backups, security, and scheduling, which we discuss in Part VI.

Instances, databases, and objects

A few words describe how SQL Server organizes things. An **instance** is one installed, running copy of the SQL Server engine. A computer can host several instances side by side: one **default instance**, reached by the computer name alone, and any number of **named instances**, reached as `ComputerName\InstanceName`. SQL Server Express installs a named instance called `SQLEXPRESS` by default, which you can reach on your own machine as `.\SQLEXPRESS` or `localhost\SQLEXPRESS`, where the dot means "this computer".

Each instance hosts many **databases**. Some are **system databases** that SQL Server needs for itself (`master`, `model`, `msdb`, and `tempdb`), and the rest are **user databases** such as the `CopperpotLab` database you will build in Chapter 2. Inside each database, objects are grouped into schemas, and objects are referred to by up to four-part names: `server.database.schema.object`. In daily work you normally use two-part names such as `dbo.orders`, which are clear, efficient, and independent of which database your connection happens to be using.

You can ask the server about itself with the `SERVERPROPERTY` function. This query, run on the instance used to test every example in this book, reports the version and edition.

Query

```
SELECT SERVERPROPERTY('ProductVersion')      AS product_version,
       SERVERPROPERTY('ProductMajorVersion') AS major_version,
       SERVERPROPERTY('Edition')             AS edition,
       SERVERPROPERTY('EngineEdition')       AS engine_edition;
```

Results

product_version	major_version	edition	engine_edition
16.0.1000.6	16	Express Edition (64-bit)	4

Major version 16 confirms SQL Server 2022. Your build number may be higher if you have applied cumulative updates, and your edition may say Developer instead of Express. The `EngineEdition` value is handy in scripts that must behave differently in the cloud: Express reports 4, and Azure SQL Database reports a different value (check the documentation for the full list). You will also often see `SELECT @@VERSION;`, which returns one long descriptive string including the operating system.

> **WARNING**
>
> Always check which instance and which database you are connected to before running a script that changes data. SSMS shows the current database in a drop-down on the toolbar and the server name in the status bar at the bottom of each query window. Running a cleanup script against production when you meant to run it against your laptop is one of the oldest and most painful mistakes in this profession.

The Tools You Will Use

SQL Server has no screen of its own; it is a service that runs in the background and waits for connections. To write and run T-SQL you need a **client tool**. This book uses two of them, and mentions a third.

SQL Server Management Studio (SSMS)

SQL Server Management Studio is Microsoft's free, Windows-only, integrated environment for SQL Server and Azure SQL. It is the tool most SQL Server professionals keep open all day. Its main parts are:

- **Object Explorer**, a tree on the left showing servers, databases, tables, columns, views, procedures, security objects, and more. You can right-click almost anything to script it, view its properties, or open it.
- The **query editor**, where you type T-SQL with syntax coloring and IntelliSense (automatic completion of table and column names). Press **F5** or **Ctrl+E** to execute. If some text is selected, only the selection runs.
- The **results pane**, with a Results tab showing rows in a grid and a Messages tab showing messages, row counts, and errors.
- Graphical **execution plans**, which you enable with Ctrl+M (actual plan) or Ctrl+L (estimated plan), and which become important in Part V.

Chapter 2 walks you through installing SSMS, connecting to your instance, and running the lab script.

sqlcmd

sqlcmd is a command-line client. It runs on Windows, Linux, and macOS, and it is perfect for running scripts automatically, for example in a deployment pipeline (Chapter 27). Every result and message in this book was produced by running the code through sqlcmd against SQL Server 2022 and copying the real output. A typical call looks like this:

Command Prompt

```
sqlcmd -S .\SQLEXPRESS -E -d CopperpotLab ^
       -Q "SELECT COUNT(*) AS menu_items FROM dbo.menu_items;"
```

Output

```
menu_items
-----------
          9

(1 rows affected)
```

The `-S` option names the server, `-E` uses your Windows login (a **trusted connection**), `-d` chooses the database, and `-Q` runs a query and exits. The caret (`^`) at the end of the first line simply continues a long command onto the next line in the Windows Command Prompt. Notice the message text: sqlcmd prints `(1 rows affected)`, while SSMS prints the grammatically correct `(1 row affected)`. The messages you see in this book come from sqlcmd, so small wording differences like this one are expected when you use SSMS.

Visual Studio Code and other clients

The **MSSQL extension for Visual Studio Code** provides a lightweight, cross-platform editor with IntelliSense, result grids, and connection management. It is a good choice on macOS and Linux, where SSMS is not available. Microsoft has retired Azure Data Studio, a cross-platform tool you may still find in older tutorials, and recommends the Visual Studio Code extension as its replacement; check the documentation for the current guidance. Many third-party tools, such as DBeaver and JetBrains DataGrip, also connect to SQL Server. The choice of tool does not change the language: T-SQL that works in one client works in all of them.

Applications connect in exactly the same way, through drivers such as ODBC, OLE DB, the Microsoft JDBC Driver, or ADO.NET (the `Microsoft.Data.SqlClient` library). An order screen in a restaurant's point-of-sale system and a nightly reporting job both end up sending T-SQL text over TDS, just as you do in SSMS.

Batches and the GO Separator

We said earlier that a batch is a group of statements sent to the server together. In SSMS and sqlcmd, you separate batches with the word `GO` on a line by itself. `GO` is *not* a T-SQL statement: the server never sees it. The client tool reads your script, splits it at each `GO`, and sends each piece separately. Two consequences follow immediately.

First, a variable lives only within the batch that declares it. The following script fails in the second batch, because `@city` was declared in the first.

Query

```
DECLARE @city nvarchar(40) = N'Seabrook';
GO
SELECT @city AS city;
GO
```

Messages

```
Msg 137, Level 15, State 2, Line 1
Must declare the scalar variable "@city".
```

Second, some statements must be alone in their batch, or at least first in it. `CREATE VIEW`, `CREATE PROCEDURE`, `CREATE FUNCTION`, and `CREATE TRIGGER` are the common examples. That is why scripts that create several such objects place a `GO` between them, as you will see from Chapter 14 onward. Because `GO` belongs to the tool, applications that send T-SQL through a driver must not include it; they simply send each batch as a separate command.

> **NOTE**
>
> A compile error such as a syntax mistake stops the entire batch before any of it runs. A runtime error, such as a constraint violation, usually stops only the failing statement, and the rest of the batch continues unless you ask it not to. This difference surprises many beginners and is the reason Chapter 19 spends so much time on `XACT_ABORT` and `TRY...CATCH`.

Meet Copperpot Kitchens

Learning a database language with abstract tables named `A` and `B` is dull and hard to remember. Throughout this book we work with **Copperpot Kitchens**, a small, fictional group of three restaurants: Copperpot Harbor and Copperpot Uptown in the town of Seabrook, and Copperpot Riverside in Millbrook. The business needs exactly the kinds of things real businesses need from a database: a menu with prices, a list of loyal customers, staff and their managers, and above all orders.

The practice database is called `CopperpotLab` and has six tables:

Table	One row per	Notable columns
`restaurants`	restaurant	`name`, `city`, `opened_on`
`menu_items`	dish or drink	`category`, `price`, `is_active`
`customers`	registered customer	`email` (may be missing), `loyalty_tier`
`staff`	employee	`role`, `manager_id` (another staff row)
`orders`	order (bill)	`customer_id` (missing for walk-ins), `status`, `tip`
`order_lines`	line on an order	`line_no`, `quantity`, `unit_price`

The data is deliberately small, with eight orders and sixteen order lines, so that you can check every answer by eye. It is also deliberately imperfect in the ways real data is imperfect: one customer has no email address, two orders were placed by walk-in guests with no customer record, one order was cancelled, one dessert is no longer on the active menu, and the staff table contains a management hierarchy in which each employee points to a manager in the same table. Each of these quirks exists to teach something, from missing values in Chapter 4 to outer joins in Chapter 6 and recursive queries in Chapter 10.

As the book progresses we add objects to this database: a view that totals orders, procedures that place and cancel orders, functions, indexes, an audit table, and more. Chapter 28 brings them together into a small but complete order platform. Chapter 2 presents the full script that creates the database and explains how to run and rerun it whenever you want a fresh start.