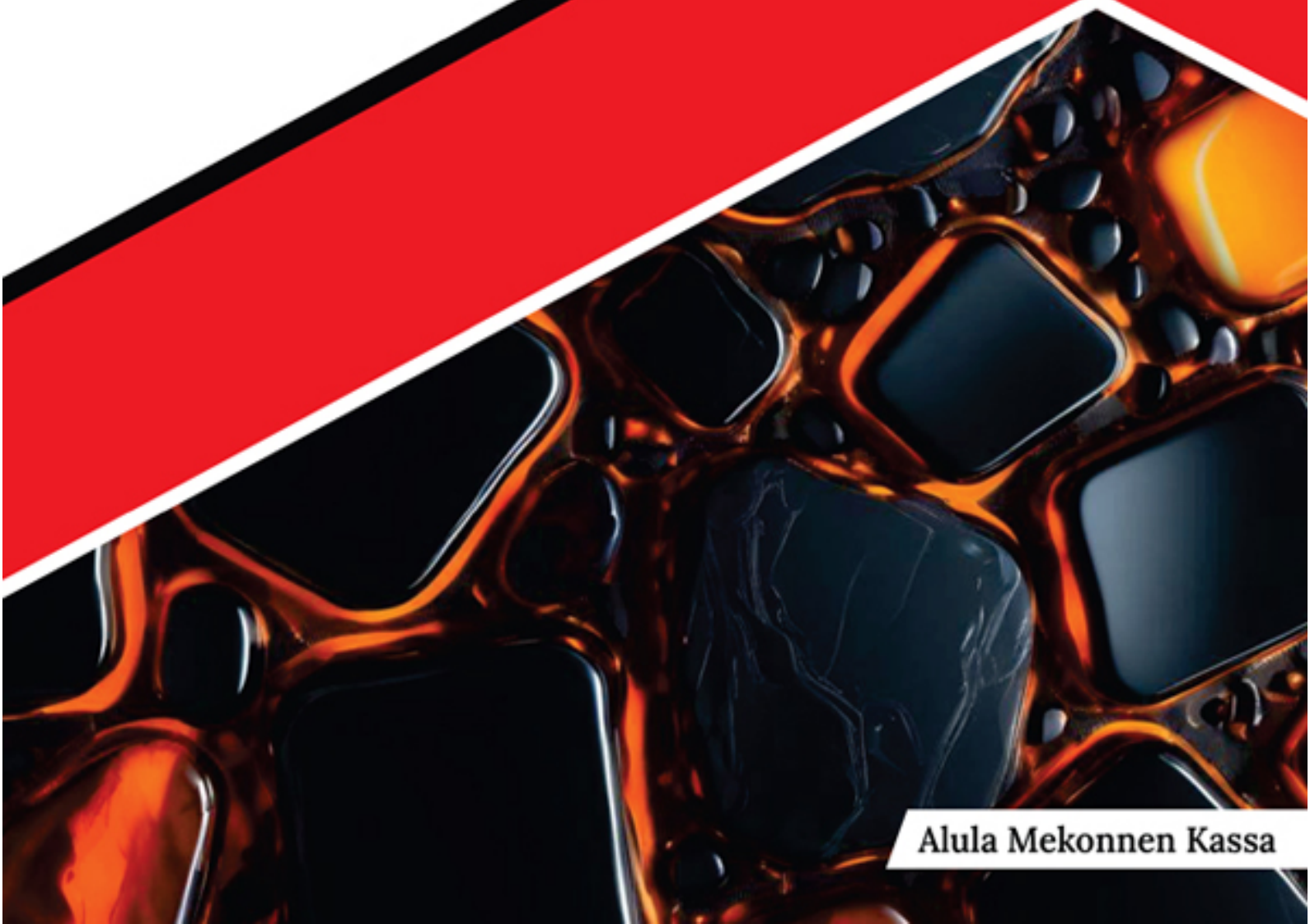
ULTIMATE

T-SQL for SQL Server Professionals

Master Query Optimization, Advanced Programming, and Enterprise Database Solutions in Microsoft SQL Server



Alula Mekonnen Kassa

First Published: April 2026

Published by: Orange Education Pvt Ltd, AVA®

Address: 9, Daryaganj, Delhi, 110002, India

275 New North Road Islington Suite 1314 London,
N1 7AA, United Kingdom

ISBN (PBK): 978-93-49887-89-3

ISBN (E-BOOK): 978-93-49887-05-3

Scan the QR code to explore our entire catalogue

www.orangeava.com

Table of Contents

CHAPTER 1

Getting Started with T-SQL

Introduction

Welcome to the world of T-SQL, which is the powerful extension of SQL designed to unlock the full potential of Microsoft SQL Server. Whether you are new to database programming, or are transitioning from other SQL dialects, this chapter will provide you with a solid foundation to begin writing efficient, clear, and maintainable T-SQL code.

We will start by exploring the fundamental concepts, syntax rules, and execution flow that form the backbone of T-SQL programming. You will learn how to write basic queries, understand batch processing, and discover the best practices that will prepare you for more advanced topics. Hence, by the end of this chapter, you will have the confidence and tools necessary to interact with SQL Server effectively, and build a strong base for mastering data manipulation and retrieval.

Transact-SQL (T-SQL) is Microsoft's proprietary extension of SQL, designed specifically for SQL Server and Azure SQL Database. While ANSI SQL provides the foundational language for interacting with relational databases, T-SQL adds programming capabilities such as variables, loops, conditionals, error handling, and system functions.

Why T-SQL is Critical

T-SQL is not merely a query language in real-world enterprise environments; it serves as the backbone of critical operations across multiple domains. It powers reporting and analytics by enabling efficient data extraction for dashboards, business intelligence tools, and ad-hoc reporting. It also supports transactional systems by managing everyday operations such as inserts, updates, and deletes within business applications. In addition, T-SQL plays a key role in database automation through scheduled jobs, triggers, and stored procedures, and it is essential for data migration and ETL processes by facilitating the movement and transformation of data between systems. By mastering T-SQL, you gain the ability to work closer to the data layer

while ensuring strong performance, maintaining data integrity, and enforcing business rules effectively at scale.

Hence, by the end of this chapter, you will be able to write clean queries, manipulate data confidently, and adopt best practices that can scale into professional database development.

Structure

In this chapter, we will cover the following topics:

- Setting up Your Environment
- T-SQL vs. ANSI SQL
- Batch Processing and the GO Command
- Data Definition Language (DDL)
- CRUD Operations
- Variables, Expressions and Operator Precedence
- Writing Clean, Maintainable, and Well-Documented T-SQL Code
- Debugging and Best Practices

Setting up Your Environment

Before writing T-SQL, you need to install and configure SQL Server and its tools.

Apart from SSMS, users may prefer alternative SQL clients such as Azure Data Studio, DBeaver, or SQLCMD for running scripts. These tools provide similar functionalities like query execution, viewing tables, and debugging, which eventually gives us flexibility to choose a preferred interface.

Installing the SQL Server

Microsoft offers several editions such as:

- **Express Edition (Free)**: Ideal for learning and small apps
- **Developer Edition (Free)**: Full feature set for development only
- **Standard/Enterprise Edition**: Offers production-level licenses

Installing SQL Server Management Studio (SSMS)

SSMS is the GUI for managing SQL Server, and performs the following functions:

- It writes and executes T-SQL scripts.
- It helps us view tables, indexes, and query plans.
- It helps us administer security and backups.

Exercise 1.1: Connecting to SQL Server

1. Open SSMS.
2. Choose "`Database Engine`" as the server type.
3. Enter the server name (for example, localhost for local installations).
4. Click on '`Connect`'.

☑ Once you have connected, expand the Databases node to explore system and user databases.

T-SQL vs. ANSI SQL

T-SQL and ANSI SQL both serve as languages for managing and querying relational databases, but they differ in scope and functionality. ANSI SQL is a standardized language defined by the American National Standards Institute, designed to provide a consistent way to interact with relational databases across different systems. It focuses on core operations such as SELECT, INSERT, UPDATE, DELETE, and basic schema definitions, ensuring portability between database platforms. In contrast, T-SQL (Transact-SQL), used primarily in Microsoft SQL Server, extends ANSI SQL by adding powerful procedural programming features such as variables, control-of-flow statements (IF, WHILE), error handling, and advanced functions. It also includes enhancements for performance tuning, transaction control, and system-level operations. As a result, while ANSI SQL emphasizes portability and standardization, T-SQL provides greater flexibility and control, making it more suitable for complex, enterprise-level database solutions.

Now, as we have understood the differences between ANSI SQL and T-SQL, let us see how batch processing works, and how the GO command affects

execution.

For beginners, understanding the difference between T-SQL and standard SQL can be confusing. In short, ANSI SQL provides the basic language for querying relational databases, while T-SQL adds procedural programming, system functions, batch processing, and advanced error handling to enable more powerful and flexible database operations.

With this foundational understanding of T-SQL and SQL, let us explore batch processing and how the GO command affects execution.

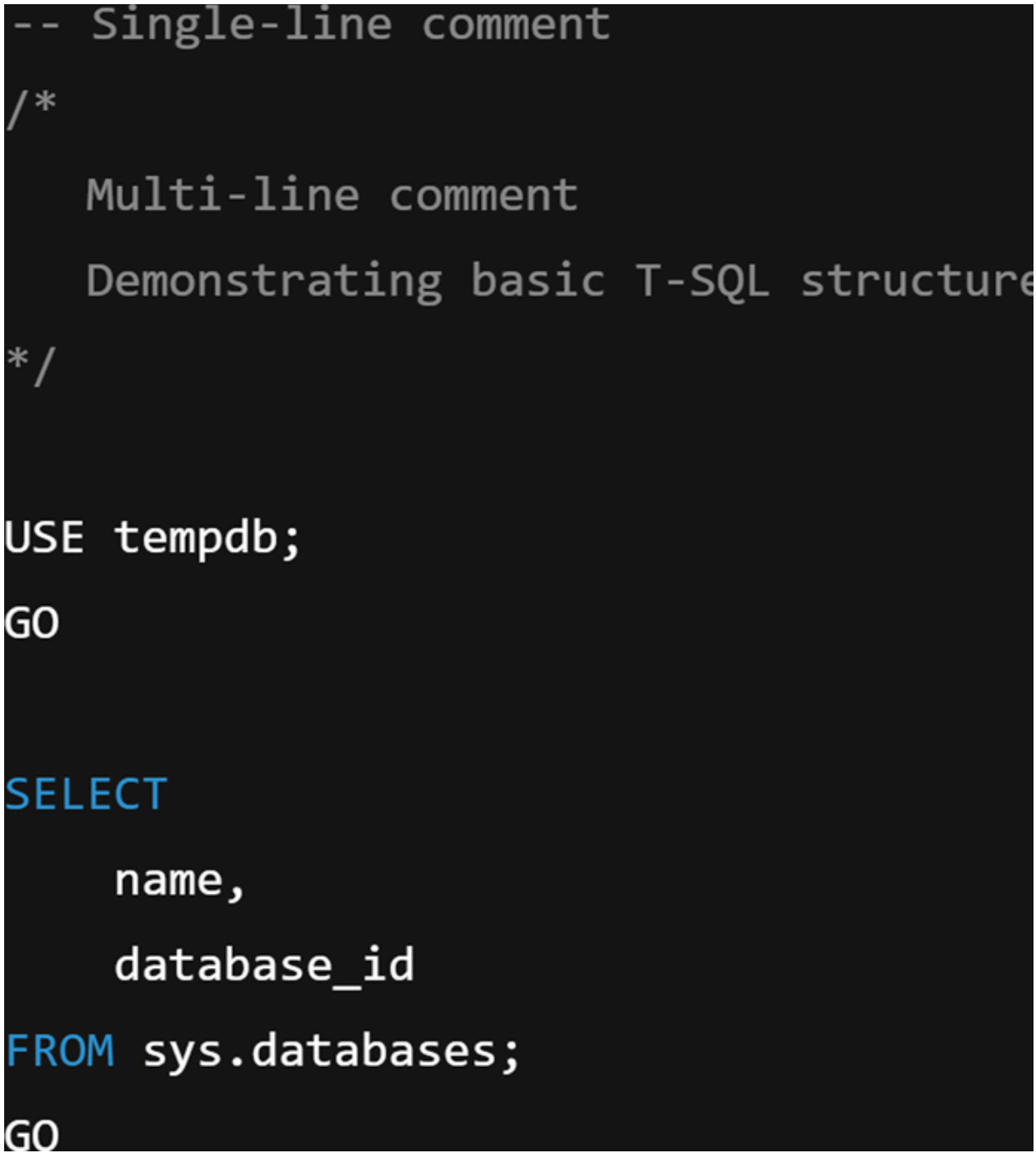


Figure 1.1: *Basic T-SQL Script Structure, Comments, and Batch Separation Using GO*

ANSI SQL: The Foundation

ANSI SQL defines the following:

- Data Query Operations (**`SELECT`**)

- Data Modification (**`INSERT`**, **`UPDATE`**, **`DELETE`**)
- Schema Definition (**`CREATE`**, **`ALTER`**, **`DROP`**)
  Example ANSI SQL query:

```
SELECT customer_name
FROM customers
WHERE customer_id = 101;
```

T-SQL Extensions

T-SQL adds:

1. **Procedural Programming:**

```
DECLARE @Counter INT = 1;
WHILE @Counter <= 5
BEGIN
 PRINT 'Count: ' + CAST(@Counter AS VARCHAR);
 SET @Counter += 1;
END;
```

2. **System Functions**: **`GETDATE()`**, **`ISNULL()`**, **`SCOPE_IDENTITY()`**
3. **Batch Processing**: The **`GO`** keyword to segment scripts
4. **Error Handling**: **`TRY…CATCH`** blocks for runtime errors

Comparison Table

Feature	ANSI SQL	T-SQL (SQL Server)
Portability	High	SQL Server only
Variables	❌	☑
Loops and Conditionals	❌	☑
Functions (Date, and so on)	Limited	Extensive built-ins
Proprietary Keywords	❌	**`TOP`**, **`OUTPUT`**, **`MERGE`**

Table 1.1: *Comparison of ANSI SQL and T-SQL (SQL Server)*

Exercise 1.2: ANSI vs. T-SQL

1. Write an ANSI SQL query to select top 10 customers.

2. Rewrite it in T-SQL using TOP, in the following way:

```
SELECT TOP 10 customer_name FROM customers;
```

Batch Processing and the GO Command

A batch is a group of one or more T-SQL statements that are sent to SQL Server as a single unit.

With batch processing understood, we can now explore how Data Definition Language (DDL) commands help structure your database objects.

With batch processing understood, we can now explore how Data Definition Language (DDL) commands help structure your database objects. The following example demonstrates how batches are used to create a table, insert data, and then retrieve that data step by step. In this example, a table named TestBatch is created using the CREATE TABLE statement, followed by separate execution batches defined by the GO command. A value is then inserted into the table using an INSERT statement, and finally, the data is retrieved with a SELECT query. Each GO keyword signals the end of a batch, ensuring that each set of commands is compiled and executed independently within SQL Server.

```
CREATE TABLE TestBatch (ID INT);
GO
INSERT INTO TestBatch (ID) VALUES (1);
GO
SELECT * FROM TestBatch;
```

Each GO signals SSMS to send a new batch. Without it, SQL Server might show dependency errors.

Variable Scope across Batches

Variables reset between batches looks like:

```
DECLARE @Test INT = 10;
GO
PRINT @Test; -- Error: Must declare the scalar variable "@Test"
```

Exercise 1.3: Batch Experimentation

1. Declare a variable @Name in one batch, and try using it after a GO.

2. Observe the scope error.
3. Fix it by removing GO.

[Diagram Placeholder: SQL Script → Parser → Execution Engine → Batch Execution]

Data Definition Language (DDL)

Data Definition Language (DDL) is a subset of SQL commands used to define, modify, and manage the structure of database objects such as tables, schemas, indexes, and constraints. Unlike DML (Data Manipulation Language), which deals with data itself, DDL focuses on how the data is stored by creating, or modifying database objects.

DDL commands automatically commit changes, meaning once executed, they are permanent, and thus cannot be rolled back (except in some database systems with transactional DDL support).

Now that you have hands-on experience with creating, altering, truncating, and renaming tables, let us move on to CRUD operations to manipulate data within those tables.

Key DDL Commands:

The primary DDL commands are:

- **`CREATE`**: Define new database objects.
- **`ALTER`**: Modify existing structures.
- **`DROP`**: Remove database objects permanently.
- **`TRUNCATE`**: Delete all data from a table while retaining its structure.
- **`RENAME`**: Change the name of existing objects.

CREATE

The **`CREATE`** command is used to create database objects such as tables, views, indexes, or schemas.

Syntax:

```
CREATE TABLE TableName (
  Column1 DataType Constraints,
  Column2 DataType Constraints,
```

```
  …
);
```

Example of Creating a Table:

```
CREATE TABLE Employees (
  EmployeeID INT PRIMARY KEY,
  FirstName NVARCHAR(50),
  LastName NVARCHAR(50),
  HireDate DATE NOT NULL,
  Salary DECIMAL(10,2) DEFAULT 50000.00
);
```

Explanation:

- **`EmployeeID INT PRIMARY KEY`**: It is a unique identifier for each employee.
- **`FirstName`**, **`LastName NVARCHAR(50)`**: It supports variable-length strings up to 50 characters.
- **`HireDate DATE NOT NULL`**: It notifies the mandatory date of hire.
- **`Salary DECIMAL(10,2)`**: It stores monetary values (10 digits, 2 decimal places) with a default of 50,000.00.

Other **`CREATE`** Examples:

- **`CREATE DATABASE`**: Define a new database.
- **`CREATE INDEX`**: Improve query performance by indexing columns.
- **`CREATE VIEW`**: Save complex queries as virtual tables.

ALTER

The **`ALTER`** command modifies the structure of an existing table or object, without needing you to drop it.

Common **`ALTER`** Operations are:

1. **Add a Column**.

   ```
   ALTER TABLE Employees ADD Email NVARCHAR(100);
   ```

   - This adds a new Email column to the Employees table.

2. **Modify a Column**.

```
ALTER TABLE Employees ALTER COLUMN Salary DECIMAL(12,2);
```

- It changes the Salary column from **`DECIMAL(10,2)`** to **`DECIMAL(12,2)`** to store larger salary values.

3. **Drop a Column.**

```
ALTER TABLE Employees DROP COLUMN Email;
```

- It removes a column from the table.

4. **Add Constraints.**

```
ALTER TABLE Employees ADD CONSTRAINT UQ_Email UNIQUE
(Email);
```

- It adds a unique constraint on the Email column.

Thus, the best practice is to verify structural changes in a development environment before altering production tables.

DROP

The **`DROP`** command permanently removes a database object. Once dropped, the object and its data are lost.

Syntax:

```
DROP TABLE TableName;
```

Example:

```
DROP TABLE Employees;
```

This removes the Employees table entirely, including its structure, data, indexes, and constraints.

Other DROP Examples:

·`DROP DATABASE HRDB;` → Deletes the entire database

·`DROP INDEX idx_salary;` → Removes an index

⚠ However, you must be careful and always backup or verify dependencies before using DROP.

TRUNCATE

TRUNCATE quickly removes all rows from a table, and retains its structure for future use.

Syntax:

```
TRUNCATE TABLE TableName;
```

Example:

```
TRUNCATE TABLE Employees;
```

- All data in the Employees table is deleted, but the table structure (columns, constraints) remains.
- It is faster than **DELETE** because it does not log individual row deletions.

Key Differences from **DELETE**:

- **DELETE** logs each row deletion, while **TRUNCATE** does not.
- **TRUNCATE** cannot be used if a table is referenced by a foreign key constraint.

RENAME

The **RENAME** operation changes the name of a database object.

SQL Server Syntax (Using Stored Procedure):

```
EXEC sp_rename 'Employees', 'Staff';
```

This renames the Employees table to Staff.

Other Examples:

- Renaming a column:

  ```
  EXEC sp_rename 'Staff.Salary', 'BaseSalary', 'COLUMN';
  ```

- Renaming indexes or constraints similarly uses sp_rename.

DDL Command Properties

- **Auto-Commit**: Once they are executed, the changes are permanent.
- **Affect Structure, and Not Data**: Unlike DML, DDL commands modify schema-level definitions.
- **Dependency Impact**: Dropping or renaming tables may break dependent views, stored procedures, or foreign keys.

- **Security Control**: DDL operations require higher privileges (for example, **ALTER**, **DROP** permissions).

Exercise 1.4: Build a Schema.

1. Create Departments and Employees tables with foreign keys.

```
CREATE TABLE Departments (
 DeptID INT PRIMARY KEY,
 DeptName NVARCHAR(100) NOT NULL
);
CREATE TABLE Employees (
 EmployeeID INT PRIMARY KEY,
 FirstName NVARCHAR(50),
 LastName NVARCHAR(50),
 HireDate DATE NOT NULL,
 Salary DECIMAL(10,2),
 DeptID INT,
 CONSTRAINT FK_Employees_Departments FOREIGN KEY (DeptID)
  REFERENCES Departments (DeptID)
);
```

2. **Add an Email Column to the Employees table**.

```
ALTER TABLE Employees ADD Email NVARCHAR(100);
```

3. **Truncate all the Employees**.

```
TRUNCATE TABLE Employees;
```

4. Rename Employees to Staff.

```
EXEC sp_rename 'Employees', 'Staff';
```

Key Takeaways:

- DDL commands manage database structure (tables, constraints, schemas).
- **CREATE** defines new objects, while **ALTER** modifies existing ones.
- **DROP** and **TRUNCATE** remove data/objects but differ in scope.
- **RENAME** updates object names without losing data.
- DDL commands are auto-committed; therefore, you should test them in staging before production use.

CRUD Operations

CRUD stands for Create, Read, Update, and Delete, and represents the four basic operations used to manage data in a database. These operations form the foundation of interacting with relational databases using SQL. Create refers to inserting new records into a table, typically using the INSERT statement. Read involves retrieving data from the database using SELECT queries. Update is used to modify existing records with the UPDATE statement, allowing changes to specific columns based on defined conditions. Delete removes records from a table using the DELETE statement, either selectively with conditions or entirely when needed.

Now, as we have learned about each CRUD operation individually, let us see how they come together in a combined workflow to efficiently maintain data.

CREATE (INSERT)

The CREATE operation is performed using the INSERT statement. It adds new records to a table. Each record inserted must match the table's structure, and also respect data types and constraints (for example, primary keys, **`NOT NULL`**, unique constraints).

Syntax:

```
INSERT INTO TableName (Column1, Column2, …, ColumnN)
VALUES (Value1, Value2, …, ValueN);
```

- **`TableName`**: The table where the data will be inserted
- **`Columns`**: List of columns receiving data
- **`Values`**: The values to insert, matching the column order

Example:

```
INSERT INTO Employees (EmployeeID, FirstName, LastName,
HireDate, Salary)
VALUES (101, 'Alice', 'Johnson', '2022-05-01', 70000);
```

Here, we will add a new employee record with details like ID, name, hire date, and salary.

Best Practices for `INSERT`:

- Always specify the column names to avoid dependency on column order.
- Validate data types to prevent insertion errors.
- Use transactions for bulk inserts to ensure atomicity.

READ (SELECT)

The **READ** operation retrieves data using the **SELECT** statement. It allows you to filter, sort, join tables, and project specific columns.

Syntax:

```
SELECT Column1, Column2, …
FROM TableName
[WHERE condition]
[ORDER BY column ASC|DESC];
```

- **WHERE** filters records based on conditions.
- **ORDER BY** sorts results in ascending (ASC) or descending (DESC) order.

Example:

```
SELECT FirstName, LastName
FROM Employees
WHERE HireDate > '2020-01-01'
ORDER BY Salary DESC;
```

This query retrieves employees hired after January 1, 2020, while displaying their first and last names sorted by salary in descending order.

Advanced **SELECT**:

- Aggregations: **COUNT**, **AVG**, **SUM**, **MIN**, **MAX**
- Joins: They combine data from multiple tables.
- Aliases: They are used to rename columns or tables for readability.

UPDATE

The **UPDATE** operation modifies existing records in a table. It uses **SET** to change column values, and **WHERE** to limit the records that are updated.

Syntax:

```
UPDATE TableName
SET Column1 = Value1, Column2 = Value2
WHERE condition;
```

- Always include a **WHERE** clause to avoid updating all rows unintentionally.

Example:

```
UPDATE Employees
SET Salary = Salary * 1.05
WHERE HireDate < '2020-01-01';
```

This increases the salary by 5% for employees hired before January 1, 2020.

Best Practices for UPDATE:

- Test the condition with a **SELECT** first to confirm affected rows.
- Use transactions for critical updates.
- Backup data before mass updates.

DELETE

The **DELETE** operation removes records from a table.

Syntax:

```
DELETE FROM TableName
WHERE condition;
```

- Without **WHERE**, all records are deleted, but the table structure remains intact.

Example:

```
DELETE FROM Employees
WHERE EmployeeID = 101;
```

This removes the employee with ID 101.

Safe **DELETE** Practices:

- Always confirm the condition with **SELECT**.
- Use transactions to allow rollbacks.

- For archiving, consider marking records inactive, instead of deleting them.

CRUD in Action (Combined Workflow)

Now that CRUD operations are clear, we will explore variables, expressions, and operator precedence, which are essential for creating dynamic and flexible T-SQL scripts.

A typical database workflow involves:

1. **INSERT** new data (for example, new hires).
2. **SELECT** data for viewing or reporting.
3. **UPDATE** records when information changes (for example, salary raise).
4. **DELETE** obsolete or incorrect data.

These operations ensure data integrity and consistent access for business processes.

```
DECLARE @BaseSalary INT = 50000;
DECLARE @Bonus INT = 5000;
DECLARE @TaxRate DECIMAL(5,2) = 0.20;

SELECT
    @BaseSalary + @Bonus AS GrossPay,
    (@BaseSalary + @Bonus) * (1 - @TaxRate) AS NetPay
```

Figure 1.2: *Variables, Expressions, and Operator Precedence in T-SQL*

Exercise 1.5: CRUD Hands-On

Practice CRUD with the following exercises:

1. **Insert 5 Employees**.

```
INSERT INTO Employees (EmployeeID, FirstName, LastName,
HireDate, Salary)
VALUES
(102, 'Bob', 'Smith', '2023-02-10', 65000),
```

```
(103, 'Clara', 'Lopez', '2024-04-15', 72000),
(104, 'David', 'Kim', '2021-11-20', 80000),
(105, 'Eva', 'Brown', '2022-09-05', 69000),
(106, 'Frank', 'Wright', '2024-01-18', 71000);
```

2. **Select Employees Hired in the Last Year.**

```
SELECT EmployeeID, FirstName, LastName, HireDate
FROM Employees
WHERE HireDate >= DATEADD(YEAR, -1, GETDATE());
```

3. **Update Salaries for Long-Term Employees**.

```
UPDATE Employees
SET Salary = Salary * 1.10
WHERE HireDate < '2020-01-01';
```

4. **Delete an Employee Record Safely**.

```
-- Verify before deleting.
SELECT * FROM Employees WHERE EmployeeID = 106;
 -- Delete if it is correct.
DELETE FROM Employees WHERE EmployeeID = 106;
```

Key Takeaways

- CRUD is fundamental for database interaction.
- Always use **`WHERE`** in **`UPDATE`** and **`DELETE`** to prevent unintentional changes.
- Combine CRUD with transactions and constraints for data safety.
- Practice CRUD queries regularly to build proficiency.

```sql
CREATE TABLE dbo.EmployeeDemo
(
    EmployeeID INT IDENTITY(1,1),
    EmployeeName VARCHAR(50),
    Salary INT
);

INSERT INTO dbo.EmployeeDemo (EmployeeName, Salary
VALUES ('John Doe', 70000);

SELECT * FROM dbo.EmployeeDemo;

UPDATE dbo.EmployeeDemo
SET Salary = 75000
WHERE EmployeeName = 'John Doe';

DELETE FROM dbo.EmployeeDemo
WHERE EmployeeName = 'John Doe';
```

Figure 1.3: *Example of CRUD Operations Executed in T-SQL*

Variables, Expressions and Operator Precedence

In T-SQL, variables and expressions are fundamental for controlling logic, storing temporary data, and performing calculations within scripts or stored

procedures. Therefore, operator precedence determines how complex expressions are evaluated in SQL.

Hence, with a solid understanding of variables and expressions, we can now focus on writing clean, maintainable T-SQL that your team can understand and work with efficiently.

Declaring Variables

Variables in T-SQL are used to store temporary data during the execution of queries, batches, or stored procedures. They help make SQL scripts dynamic and reusable.

Syntax:

```
DECLARE @VariableName DataType [= InitialValue];
```

- **`DECLARE`** initializes a variable.
- `@` prefix denotes a variable.
- Optionally, you can assign a value during declaration.

Example:

```
DECLARE @Today DATE = GETDATE();
PRINT @Today;
```

- Here, **`@Today`** stores the current date retrieved by **`GETDATE()`**.
- The **`PRINT`** statement outputs its value.

Assigning Values: Variables can be assigned values using **`SET`** or **`SELECT`**.

Using **`SET`**:

```
DECLARE @Name NVARCHAR(50);
SET @Name = 'Alice';
PRINT @Name;
```

Using `SELECT`:

```
DECLARE @Salary DECIMAL(10,2);
SELECT @Salary = Salary FROM Employees WHERE EmployeeID = 101;
PRINT @Salary;
```

- **`SET`** assigns a single value explicitly.

- `SELECT` can retrieve and assign directly from a query (this applies if multiple rows return, and only the last row value is stored).

Scope of Variables:

- **Local Variables**: They exist only within the batch, procedure, or function where defined.
- **Global Variables**: They are prefixed with @@ and predefined by the SQL Server (for example, `@@VERSION`, `@@ROWCOUNT`).

Arithmetic and Logical Expressions

Expressions in SQL involve constants, variables, column values, and operators to compute results. They are of two types, which are discussed as follows:

Arithmetic Expressions

Arithmetic operators perform mathematical operations.

Operator	Description	Example	Result
+	Addition	SELECT 10 + 5;	15
-	Subtraction	SELECT 10 - 3;	7
*	Multiplication	SELECT 6 * 4;	24
/	Division	SELECT 20 / 4;	5
%	Modulus (Remainder)	SELECT 22 % 5;	2

Table 1.2: *Common Arithmetic Operators in T-SQL*

Example:

```
SELECT (100 + 20) * 0.1 AS Bonus;
```

- First (100 + 20) is evaluated, and is then multiplied by 0.1.

Logical Expressions

Logical operators control filtering and conditions.

Operator	Description	Example
AND	It is true, if both conditions are true.	`WHERE Salary > 50000 AND DeptID=2`

OR	It is true, if any condition is true.	`WHERE DeptID=2 OR DeptID=3`
NOT	It reverses the condition itself.	`WHERE NOT DeptID=2`

Table 1.3: *Logical Operators in T-SQL*

Operator Precedence

SQL evaluates expressions based on operator precedence. Operators with higher precedence execute first.

Order of Precedence:

1. Parentheses `()`: Highest priority
2. Arithmetic Operators: Multiplication/division before addition/subtraction.
3. Comparison Operators (for example, =, <, >)
4. Logical Operators: **`NOT`** before **`AND`**, and before **`OR`**.

Example: Operator Precedence

```
SELECT 10 + 5 * 2 AS Result;
Result = 20
```

(Multiplication executes before addition → 10 + (5 * 2))

To override precedence:

```
SELECT (10 + 5) * 2 AS Result;
Result = 30
```

(Parentheses force addition first.)

Logical Precedence (**`AND`** vs. **`OR`**):

`AND` is evaluated before OR.

Example:

```
SELECT * FROM Employees
WHERE DeptID = 2 OR DeptID = 3 AND Salary > 60000;
```

- Here, **`AND`** is evaluated first:

  ```
  DeptID = 2 OR (DeptID = 3 AND Salary > 60000)
  ```

- To change order:

  ```
  SELECT * FROM Employees
  ```

```
WHERE (DeptID = 2 OR DeptID = 3) AND Salary > 60000;
```

Practical Usage in Queries

Variables and expressions are often combined for dynamic filtering:

```
DECLARE @MinSalary DECIMAL(10,2) = 60000;
SELECT FirstName, Salary
FROM Employees
WHERE Salary > @MinSalary AND DeptID = 2;
```

Common Use Cases

- Storing dynamic parameters in stored procedures
- Performing calculations (for example, bonuses, tax deductions)
- Building conditional logic with **IF…ELSE** using variables
- Testing precedence in complex WHERE clauses

Exercise 1.6: Precedence Testing

1. **Test AND vs. OR Precedence**

   ```
   -- Query 1: Without Parentheses
   SELECT * FROM Employees
   WHERE DeptID = 2 OR DeptID = 3 AND Salary > 60000;

   -- Query 2: With Parentheses (Altered Logic)
   SELECT * FROM Employees
   WHERE (DeptID = 2 OR DeptID = 3) AND Salary > 60000;
   ```

   Now, you can compare how the results differ.

2. **Variable Assignment with Arithmetic**

   ```
   DECLARE @BaseSalary DECIMAL(10,2) = 50000;
   DECLARE @BonusRate DECIMAL(5,2) = 0.10;
   DECLARE @FinalSalary DECIMAL(10,2);

   SET @FinalSalary = @BaseSalary + (@BaseSalary *
   @BonusRate);
   PRINT @FinalSalary;
   ```

3. **Logical Testing**

   ```
   DECLARE @Check INT = 1;
   IF (@Check = 1 AND NOT @Check = 0)
    PRINT 'Condition True';
   ```

```
ELSE
  PRINT 'Condition False';
```

Key Takeaways

- Variables store temporary values and improve script flexibility.
- Arithmetic operators follow math rules, while parentheses override defaults.
- Logical **AND** executes before OR, unless parentheses change it.
- Test precedence carefully in complex **WHERE** clauses.

Writing Clean, Maintainable, and Well-Documented T-SQL Code

Clean, well-structured, and maintainable T-SQL code is crucial for collaboration, debugging, and long-term support. It is because readable SQL ensures that developers, DBAs, and analysts can easily understand and modify code. Therefore, following consistent formatting and commenting standards results in fewer errors and better performance tuning.

Now that you know how to document your T-SQL code clearly with comments, it is important to understand that proper documentation not only explains what each part of your code does but also makes it easier for others (and your future self) to read, debug, and maintain. With this foundation, we can now explore broader best practices that help ensure your queries are both maintainable and efficient, including consistent formatting, meaningful naming conventions, and thoughtful structuring of complex operations.

Formatting Conventions

Proper formatting enhances readability and reduces ambiguity in complex queries. Here are the essential guidelines:

1. **Use Uppercase Keywords**.

   SQL keywords such as **SELECT**, **FROM**, **WHERE**, and **JOIN** should always be written in uppercase, while table and column names remain lowercase or PascalCase. This improves visual distinction between SQL syntax and identifiers.

Example:

```
SELECT EmployeeID,
   FirstName,
   LastName
FROM Employees
WHERE HireDate > '2020-01-01';
```

2. **Align Columns Vertically**.

   For multi-column queries, aligning columns vertically makes it easier to scan and modify later.

   Good Formatting:

```
SELECT EmployeeID,
   FirstName,
   LastName,
   HireDate,
   Salary
FROM Employees;
```

   Poor Formatting:

```
SELECT EmployeeID, FirstName, LastName, HireDate, Salary
FROM Employees;
```

   - The compact single-line query is harder to read and maintain.

3. **Break Long Queries into Logical Parts**.

   For better readability:

   - Place each clause (**`SELECT`**, **`FROM`**, **`JOIN`**, **`WHERE`**, and **`ORDER BY`**) on a new line.
   - Indent subsequent lines for nested queries.

   Example:

```
SELECT e.EmployeeID,
   e.FirstName,
   e.LastName,
   d.DeptName
FROM Employees AS e
JOIN Departments AS d
  ON e.DeptID = d.DeptID
```

```
WHERE e.HireDate > '2020-01-01'
ORDER BY e.Salary DESC;
```

4. **Use Aliases Wisely.**
   - Use short and meaningful aliases to simplify code.
   - Always include AS for clarity.

   Example:

```
SELECT e.FirstName,
   e.LastName,
   d.DeptName
FROM Employees AS e
JOIN Departments AS d ON e.DeptID = d.DeptID;
```

5. **Ensure Consistent Indentation.**

   Indent nested queries or CASE expressions, as shown here:

```
SELECT FirstName,
   LastName,
   CASE
    WHEN Salary > 80000 THEN 'High'
   WHEN Salary BETWEEN 50000 AND 80000 THEN 'Medium'
   ELSE 'Low'
   END AS SalaryRange
FROM Employees;
```

Commenting Code

Comments explain logic, which makes T-SQL easier to maintain. Without comments, future developers (or even you) may struggle to understand query intent. You may refer to the following comment categories to know more about them.

Single-Line Comments

Use `--` for quick notes or inline comments.

Example:

```
-- Select employees hired after 2020
SELECT EmployeeID,
```

```
   FirstName,
   HireDate
FROM Employees
WHERE HireDate > '2020-01-01';
```

Multi-Line Comments

Use `/*` … `*/` for detailed explanations, or to temporarily disable blocks of code.

Example:

```
/* Increase salary by 5%
  for employees hired before 2020 */
UPDATE Employees
SET Salary = Salary * 1.05
WHERE HireDate < '2020-01-01';
Commenting for Code Blocks (Disabled Code):
/* The following query is disabled during testing:
SELECT * FROM Employees WHERE Salary > 100000;
*/
```

Explain Complex Logic

Always explain business rules or unusual joins/conditions, such as shown in the example here:

```
-- Use LEFT JOIN to include employees without assigned
departments.
SELECT e.EmployeeID, e.FirstName, d.DeptName
FROM Employees AS e
LEFT JOIN Departments AS d ON e.DeptID = d.DeptID;
```

Best Practices for Maintainable T-SQL

****Avoid `SELECT *`.**

Explicitly specify the required columns:

```
SELECT FirstName, LastName FROM Employees; -- Better than
SELECT *
```

Use Meaningful Names.

Choose clear table, column, and variable names (**`EmpSalary`** vs. **`ESal`**).

Break Down Complex Queries.

- Use Common Table Expressions (CTEs) to simplify nested subqueries.

```
WITH RecentHires AS (
  SELECT EmployeeID, FirstName, HireDate
  FROM Employees
  WHERE HireDate > '2020-01-01'
)
SELECT * FROM RecentHires;
```

Ensure Consistent Case Usage

Pick a case style for identifiers (**`PascalCase`**, **`snake_case`**), and make sure that you stick with it.

Avoid Deep Nesting

Excessive nested queries or **`CASE`** statements reduce readability. Therefore, you can break it into smaller queries.

Document Stored Procedures and Functions

Include header comments, such as:

```
/* Procedure: usp_GetRecentHires
  Purpose: Fetch employees hired last year.
  Author: John Doe
  Date: 2025-08-01
*/
```

Separate Logic from Presentation.

Use views or stored procedures to encapsulate logic, instead of embedding complex queries in reports or applications.

Example: Poor vs. Clean SQL

Here is how a poorly written query looks like:

```
select * from employees e join departments d on
e.deptid=d.deptid where salary>50000 or hiredate>'2020-01-01'
order by salary desc;
```

Now, look at a well-written query:

```
-- Fetch employees earning above $50K or hired after 2020
```

```
SELECT e.EmployeeID,
   e.FirstName,
   e.LastName,
   d.DeptName,
   e.Salary
FROM Employees AS e
JOIN Departments AS d
   ON e.DeptID = d.DeptID
WHERE e.Salary > 50000
  OR e.HireDate > '2020-01-01'
ORDER BY e.Salary DESC;
```

Key Takeaways

- Readable code saves time for debugging and teamwork.
- Use uppercase keywords, aligned columns, and proper indentation.
- Always comment on complex logic to explain reasoning.
- Break down complex queries into manageable parts by using CTEs or views.
- Maintain consistency in formatting and naming, as it is essential for maintainability.

Debugging and Best Practices

Efficient debugging and adherence to best practices are critical to ensure that T-SQL code is reliable, performant, and safe for production environments. Debugging helps identify errors early, while best practices prevent common mistakes and performance issues.

Finally, after exploring debugging techniques and best practices, let us summarize the key takeaways from this chapter, and test your knowledge with a quiz.

Debugging T-SQL Code

Debugging involves testing queries step-by-step to validate their correctness before applying them in a live database. Hence, you must consider the following tips while debugging:

Always Test in Development

- Never execute untested queries directly on production databases.
- Use a development or staging environment with a copy of production data.
- Validate:
    - Query Correctness
    - Data Accuracy
    - The Impact of Updates or Deletions

Example:

Test a delete query safely, as shown in the following way:

```
-- Test using SELECT first
SELECT * FROM Employees WHERE EmployeeID = 999;
 -- Then run DELETE only if correct
DELETE FROM Employees WHERE EmployeeID = 999;
```

Use Transactions for Safe Changes

Transactions allow you to group operations, and either commit them permanently, or roll them back if issues occur.

Syntax:

```
BEGIN TRAN;    -- Start transaction.
DELETE FROM Employees WHERE EmployeeID = 999;
ROLLBACK TRAN; -- Undo changes (Testing Phase).
-- COMMIT TRAN;   -- Finalize only when it is verified.
```

- Use **`ROLLBACK TRAN`** to undo mistakes during debugging.
- In production, confirm the results first, and then use **`COMMIT TRAN`**.

Use `PRINT` or `SELECT` for Debugging

- Use **`PRINT`** to display variable values or debugging messages.
- Use **`SELECT`** statements to preview affected rows before running updates/deletes.

Example:

```
DECLARE @Count INT;
```

```
SELECT @Count = COUNT(*) FROM Employees WHERE Salary > 80000;
PRINT 'Employees above 80K: ' + CAST(@Count AS NVARCHAR(10));
```

Use TRY…CATCH for Error Handling

Handle runtime errors gracefully, as shown here:

```
BEGIN TRY
  UPDATE Employees SET Salary = Salary / 0; -- Error
END TRY
BEGIN CATCH
  PRINT 'Error: ' + ERROR_MESSAGE();
END CATCH;
```

Best Practices for T-SQL

Adopting best practices improves code quality, prevents mistakes, and enhances performance. Therefore, here are a few of them:

Avoid SELECT * in Production

Fetch only the necessary columns to reduce overhead, as shown here:

```
-- Avoid
SELECT * FROM Employees;

-- Use
SELECT EmployeeID, FirstName, LastName FROM Employees;
```

Use Execution Plans to Optimize Performance

Execution plans visualize how SQL Server executes your queries, while showing index usage, joins, and scans. To view them in SSMS, press *Ctrl+M* before running a query to display the estimated execution plan. This should be done because reading execution plans helps identify bottlenecks, besides allowing us to optimize query performance.

Example:

```
SELECT e.EmployeeID, e.FirstName, d.DeptName
FROM Employees e
JOIN Departments d ON e.DeptID = d.DeptID
WHERE e.Salary > 50000;
```

Execution plans in SQL Server Management Studio (SSMS) help identify:

- Index Usage
- Table Scans vs. Index Seeks
- Bottlenecks in Queries

Shortcut: Press *Ctrl + M* in SSMS before executing to view the estimated plan.

Use Proper Indexing

Create indexes on frequently queried columns (for example, **`WHERE`**, **`JOIN`**, **`ORDER`** BY).

Example: `CREATE INDEX idx_Employees_HireDate ON Employees(HireDate);`

Parameterize Queries

Avoid hardcoding values in production queries, as shown here:

```
DECLARE @MinSalary DECIMAL(10,2) = 60000;
SELECT FirstName, Salary FROM Employees WHERE Salary >
@MinSalary;
```

Comment Critical Logic

Always explain complex conditions, joins, or business rules:

```
-- Filtering employees in the Sales department who earn above
$70,000.
SELECT FirstName, Salary FROM Employees WHERE DeptID = 3 AND
Salary > 70000;
```

Validate Row Counts before DML

Use **`SELECT COUNT(*)`** to confirm the affected rows before **`UPDATE`** or **`DELETE`**.

Key Takeaways

- Debug in development, and not during production.
- Use transactions and rollbacks for safe testing.

- Avoid `SELECT *`, and retrieve only needed columns.
- Use execution plans and indexing to improve performance.
- Comment and parameterize code for maintainability.

Conclusion

In this chapter, we have built a solid foundation for working with T-SQL by covering the essential concepts and practices that every database professional should know. You learned how to set up your environment, understood the differences between T-SQL and ANSI SQL, and explored batch processing with the GO command. We examined how Data Definition Language (DDL) commands help structure your database objects, and delved into the core CRUD operations for managing data. Additionally, you became familiar with variables, expressions, and operator precedence, and learned the importance of writing clean, maintainable, and well-documented T-SQL code. Finally, we discussed debugging techniques and best practices to ensure your queries are efficient and reliable. With these tools and knowledge, you are now equipped to write T-SQL code that is both powerful and professional, setting the stage for more advanced topics in database development and management.

Key Takeaways

- **T-SQL Extends ANSI SQL with Procedural Logic**: T-SQL includes variables, loops, conditions, and error handling, which makes it more powerful than standard SQL for complex database operations.
- **Batches and GO Define Execution Boundaries**: A batch is a group of SQL statements sent to SQL Server for execution together. The GO command separates batches, and then resets variable scope.
- **CRUD and DDL Form the Foundation of T-SQL**: CRUD (Create, Read, Update, Delete) handles data manipulation, while DDL (Data Definition Language) manages database structures.
- **Clean Formatting and Comments Facilitate Maintainability**: Consistent indentation, clear naming conventions, and proper commenting improve readability and teamwork, which eventually reduces errors in large projects.

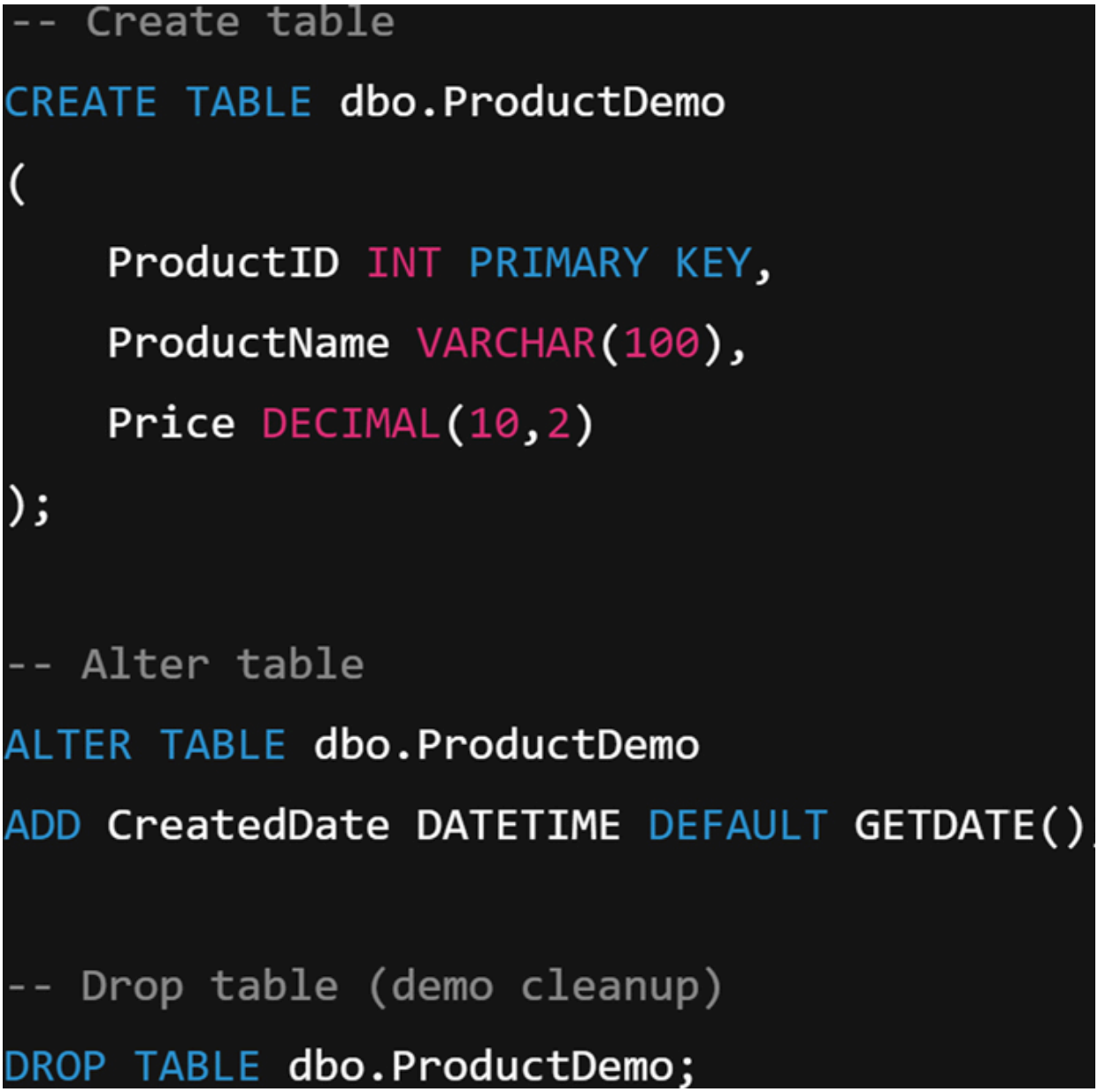


Figure 1.4: *DL Operations with Clean Formatting and Comments*

Quiz

1. What is the difference between **`DELETE`** and **`TRUNCATE`**?
    - **`DELETE`** removes rows one by one, logs each deletion, and can include a **`WHERE`** clause.
    - **`TRUNCATE`** quickly removes all rows, resets identity columns, and cannot include **`WHERE`**.

2. How does GO affect variable scope?

    - Variables declared in one batch are not accessible after a GO statement because GO ends the batch, and also resets the scope.

3. Which has higher precedence: **AND** or **OR**?

    - **AND** has higher precedence than OR. Moreover, parentheses should be used to enforce the intended logical grouping.

4. Write a query to list the top 3 highest-paid employees.

```
SELECT TOP 3 EmployeeID,
 FirstName,
  LastName,
 Salary
FROM Employees
ORDER BY Salary DESC;
```