



Learn T-SQL

from

SCRATCH

An Easy-to-Follow Guide for Designing, Developing, and Deploying Databases
in the SQL Server and Writing T-SQL Queries Efficiently



BRAHMANAND SHUKLA



Learn T-SQL From Scratch

*An Easy-to-Follow Guide for Designing,
Developing, and Deploying Databases in the
SQL Server and Writing T-SQL Queries Efficiently*

Brahmanand Shukla



www.bpbonline.com

FIRST EDITION 2022

Copyright © BPB Publications, India

ISBN: 978-93-91392-413

All Rights Reserved. No part of this publication may be reproduced, distributed or transmitted in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication, photocopy, recording, or by any electronic and mechanical means.

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.

To View Complete
BPB Publications Catalogue
Scan the QR Code:



www.bpbonline.com

Forewords

SQL Server has evolved over the last 2 decades into one of the most prominent databases for enterprise application development out there. It is never too late to start learning and I wouldn't miss an opportunity to learn SQL Server Coding from scratch from a book as useful as "Learn T-SQL from Scratch".

I have known Brahmanand Shukla (aka Brahma) when I worked with Ebixcash Financial Technologies as CTO. Brahma worked on some of the toughest and complex SQL Server projects in that stint with me. He is a SQL Server development authority and runs a SQL Server based blog & consultancy, SQL Server Carpenters working with marquee Customers in SQL Server area.

The book is by practitioner for a to-be practitioner, by an expert developer for a novice developer. It explains the concepts, and quickly jumps into the code and implementation of it. It does so with simply explained concepts, pictures, code snippets, SSMS snapshots, with a flair for hands on approach. So, anyone aiming to start off quickly with SQL Server development, with no background, will find this book hard to keep down. From simple concepts like tables, indexes, functions, and procedures, it goes to touch upon advanced topics as handling of XML & JSON data, transactions, and error handling.

Thanks to people like Brahmanand Shukla, author of this book, who share their practical knowledge with easy journey from basic

to advanced concepts, you can discover that learning T-SQL has never been this easy.

— **Pradeep Paliwal**

MTech (Data Science), BITS Pilani

MBA (Systems), XIMB

BE, VJTI

Regardless of the topic, from martial arts, to woodworking, to flying a plane, and certainly to building and maintaining databases, fundamentals matter more than anything else. If you don't have a good foundation for your building, it will collapse. Same thing goes for working with SQL Server. If you don't have the fundamentals well understood and properly applied, you, and the organization you're supporting, will suffer.

All of this is why a book like this one is so important.

While you're reading through this book, keep that concept in mind. You need to understand the fundamentals of SQL Server in order to do basic tasks, yes, but also the advanced ones. The structure and layout of the book builds knowledge and doesn't assume you already understand concepts. You'll be walked up to topics carefully. You'll get the fundamentals that are so important, vital in fact.

So, study it well and be better at your database development, maintenance, administration, analysis, pretty much anything and everything to do with SQL Server.

—Grant Fritchey

If you are looking to get started with T-SQL and SQL Server, this book is for you. This book guides a novice right from understanding basic database concepts, installation, which menu to click on Management studio, T-SQL concepts, and SQL query writing.

Gradually, the book moves towards complex topics, such as transactions, XML handling, error handling, CTE, Indexes, stored procedures, views, triggers, and many more concepts.

Moreover, Conclusion, Points to remember, and Test questions after every chapter just helps readers to revise and test themselves on where they stand.

Kudos to Mr. Brahmanand to squeeze and pack his 15 years expertise in this book.

—Shivprasad Koirala

Dedicated to

Late. Shri Ramkishor Shukla & Late. Smt. Chandrakali Shukla
My grandparents, the root of my existence.

&

Late. Shri Sitaram Pednekar
My beloved uncle, the biggest motivation during my difficult times

About the Author

Brahmanand Shukla is a SQL Server Consultant with over 13 years of experience in Software Design, Development and Support. Brahmanand started his journey with SQL Server 2000 in the year 2007, and since then the SQL Server has been his second wife.

SQL Server is not the limit for Brahmanand. He has also designed and developed the databases in PostgreSQL, Oracle and Apache Cassandra. He has developed applications using VB 6.0, VB.Net, C#, ASP.Net, and Crystal Report. He has also delivered various cutting edge ETL solutions in SSIS.

Brahmanand believes, the database is no different than an ordinary storage such as cupboard. He feels proud to be known as the SQL Server Carpenter. He started blogging on SQL Server way back in 2016 and launched his personal weblog. He started his Consulting and Training firm under the same brand in 2020. He is also a community member of SQL Server Central.

He can be reached out at

Brahmanand.Shukla@SQLServerCarpenter.com

About the Reviewers

Suresh Kumar comes with a rich experience of 20+ years in IT industry with specialization in Database technologies. He is an architect with expertise in redesigning legacy systems and performance tuning. He pursued BE in Electronics & Comm. from CR State College of Engineering, Murthal (known as DCRUST now). He started his professional career with Tata Consultancy Services in 2001 and has worked with fortune 500 companies like Hartford Insurance, AcNielsen. He is currently leading the development vertical at Scalability Engineers, Dehradun.

Sandesh Jadav comes with a rich experience of 12+ years in the IT industry. He is an expert full-stack developer skilled in various technologies ranging from ASP.Net, C#, Angular JS, JavaScript, SQL Server, SSIS, SSRS, and Business Objects.

He is a full-stack developer but has more inclination towards the T-SQL and SQL Server. Besides, he is also a seasoned Technical Project Manager and has led various complex deliveries. In his current assignment, he is leading several business-critical applications at HDFC ERGO General Insurance.

He is equally competent in various businesses of Finance domains such as Asset Management, Wealth Management, Investment Banking, and Insurance. He has worked with esteemed organizations like Financial Technologies, EbixCash.

He is a Computer Science post-graduate from the University of Mumbai. He also holds a Management degree from Wellingkar's Institute, Mumbai.

Acknowledgement

There is always someone behind one's success. I give this credit to Sai Kiran Gangam, my ex-colleague. It was because of his push, I started my blog and finally decided to write this book. I can never forget my other ex-colleagues Vinod Naidu and Saumitra Bhardwaj. They always pushed me and made me explore my capabilities. Vinod is also my best mate and Alexa since 2012. Alexa, because he is the one whom I approach for my questions. Vinod also contributed to this book at various stages.

I would not have been able to write this book if my uncle Shri Baldev Prasad Shukla would not have supported my childhood education. I would not have known the 'S' of SQL Server, if my mentor and teacher Mr. Sikander Manihar would not have trained me and given me the first chance to work in the IT industry. It was Mr. Sikander who made me realize my potential.

I'm indebted to my parents for everything they did for me. They struggled to raise their three kids, including me. I would not have been what I am without their support and blessings.

उत्सवे व्यसने चैव दुर्भिक्षे राष्ट्रविप्लवे ।

राजद्वारे श्मशाने च यतिष्ठति स वाण्धवः

Meaning: He who stands with, in good times, bad times, draught, riot, war, king's court and after death; is a real friend.

Life is a roller-coaster ride. I am blessed to have my family and friends who stand true to the aforesaid mentioned Sanskrit shloka. My brother Krishnanand and his wife Pooja, my sister Shivangi, my friends for life Santosh Gupta and Harkesh Sharma. They are witnesses of all my good and bad times. They stood by me always.

This book has been written during the biggest pandemic I've ever seen. I've spent many nights, weekends and holiday on writing this book. It would not have been possible for me to take up the assignment to write this book, and complete it, without the support of my lovely wife Himani and two sons – Shashank and Shreyansh.

Finally, I would like to thank BPB Publications for giving me this opportunity to write my first book for them.

Preface

Data, isn't it a fascinating word nowadays? Indeed! You would often hear people talking about it. With increasing data, there is a need of effective method to manage, process and query it.

With the increasing digitalization, even the offline processes are being transformed into the digital online processes. It has nearly become impossible to handle such a vast data in the form of excels. The job profile of such data entry operators, and other people purely doing only physical processes, will demand to acquire new skills. Many data analysts who were earlier working on excel for more than 2 decades, are now told to upgrade their skills and learn SQL.

Most of these data is now being maintained in the database, and most of them are based out of SQL. Data has opened many new job opportunities. Earlier only the application developers, database developers and database administrators were expected to have the knowledge of the SQL. But the game has changed now. Even the non-technical job profile demands the knowledge of SQL. Set of people feels, this change is taking lot of jobs. But we must understand the things changes along with the time. The best part is – it gives precious opportunities to people who were not earlier exposed to the IT. They can now become the integral part of IT, and can grow by gaining the SQL skills.

Imagine you've an excel file which has 10 sheets, each having millions of records. What would be the situation if you are told to perform an analytics over the data available in excel file. Opening such excel file itself is a challenge. It is also prone to crashes, and there is high possibility that you may lose the data, if such a crash happens.

But, if you know SQL, the same excel file can be dumped to a relational database. You can write SQL scripts, and can perform various types of analytics. You can also easily combine the multiple datasets with simple SQL commands. You can do slice and dice the way you want, and can complete the data analytics quicker than excels. Such SQL queries can be reused as many times you want. This has truly transformed the way the data used to be dealt.

There are multiple SQL based database in existence today. Most of them are relational database. We are talking about one of such relational database called SQL Server. It is a product offering of Microsoft. SQL is of various types. The base SQL is called ANSI SQL, which is a standard. Almost all the relational database supports this. But, most of them have their own flavor of SQL.

Because standard can't be customized. Hence the relational database have come up with their own version of SQL. In the same line T-SQL is the extension of SQL for the SQL Server and Sybase. If you know any kind of SQL, you'll be able to work on almost all kinds of SQL. There are slight syntax change though which you can learn quickly.

Database opens multiple branches, and obviously opens multiple job opportunities. Here are few job profile which is based on SQL. Since we are talking about T-SQL in this book, so consider SQL Server as the database and T-SQL as SQL language.

Business Analyst: They are basically IT people who understand the business well. They are kind of bridge between the Business and IT teams. Business Analysts are now also expected to know SQL to better analyze the data and requirement, before getting it developed. Their knowledge of SQL can also help them to properly plan the data migration, and define the data API's required to integrate between two different applications.

Database Architects: They are the founders of your database. They understand the business requirement, and effectively model your database. They define the structure and flow of objects in your database. They define the tables, columns, their datatypes etc. A best performing database is the outcome of a good data model. If you have one, you should appreciate your database architect.

Database Developer: They are the people who develop the database based on the model shared by the Database Architect. They also develop the queries, and other programs as desired by the respective business.

Database Administrator: They make sure the database servers and databases are up and running all the times, and obviously in a healthy state. It is also their responsibility to plan the hardware capacity such as Disk, Memory, and CPU etc. based on the data growth and load on the system. They also monitor your databases

and database servers, and tune the performance, if there is any performance hiccups. They also plan the database backups, and help you recover from the unplanned disasters.

Data Analyst: They are the people who serve the business on their data requirements. They develop and run the queries as per the requirement and provide the data to the requestor.

Data Scientists: They leverage the historical data to come up with the prediction or the various forecasting, based on the business demand.

It doesn't matter which profile you chose amongst mentioned aforesaid. You need to learn T-SQL. You must have now got an idea of why SQL is important and why you should learn it. We've tried in this book to keep the contents simple as much possible, so that even a reader who is not from IT and Programming background can learn T-SQL with ease.

The overall book has 15 chapters as follows. Each of these chapters are the ladder to reach the goal of learning T-SQL:

[Chapter 1 \(Getting started\):](#) You'll understand the basics of Relation Database, SQL Server and SQL. You'll also learn about the SSMS which is an IDE for SQL Server.

[Chapter 2 \(Table\):](#) You'll learn about the tables, columns, datatypes and constraints. You'll be able to create, modify and drop the tables on your own.

[Chapter 3_\(Index\):](#) You'll understand the indexes. You'll learn to create and drop them.

[Chapter 4_\(DML\):](#) You'll learn about insert, update, delete and select commands. You'll be able to create a new records, modify and delete existing records, and read the existing records. After this chapter you'll be able to read the data from single table. You'll also be able to filter and sort the result. You'll also be able to fetch the top N records and perform the paging on the result-set.

[Chapter 5_\(Built-In Functions - Part 1\):](#) You'll learn about various built-in functions. After the end of this chapter, you'll be able to perform aggregation, and also make use of various string, numeric and date built-in functions. These functions will be useful if you want to do more than just reading the raw data from the table.

[Chapter 6_\(Join, Apply and Subquery\):](#) You'll learn to combine the multiple tables to get a unified result-set. After the end of this chapter, you'll be able to join multiple tables, and you'll be able to perform advanced analytics by combining aggregation, sorting, filtering etc.

[Chapter 7_\(Built-In Functions - Part 2\):](#) You'll learn about various advanced built-in function. They are called – Window, Ranking and Analytic functions. After the end of this chapter, you'll be able to perform complex queries, by combining the knowledge gained through [Chapter 2](#) to

Chapter 8 (Dealing with XML and JSON): At the end of this chapter, you'll be able to query the semi-structured data in the form of XML and JSON. You'll be able to convert XML / JSON to Table and vice versa.

Chapter 9 (Variables and Control Flow Statements): You'll learn the basics of programming in this chapter. At the end of this chapter, you'll be able to declare a variable, assign the value to it, and read the variable value. You'll also be able to define control flow statements such as IF statement, CASE statement and loop.

Chapter 10 (Temporary Tables, CTE and MERGE Statement): In this chapter you'll learn to break a complex query into multiple smaller queries. You'll also learn to store and use the intermediate result in temporary tables and table variables.

Chapter 11 (Error Handling & Transaction Management): At the end of this chapter, you'll be able to implement the Error Handling in your T-SQL program. You'll be able to handle the errors, and implement the action upon an error, including the rollback mechanism. You would not want a half-baked solution. Learnings of this chapter will help you control the transactions in your program, and the way you commit or rollback it, especially in case of errors.

Chapter 12 (Data Conversion, Cross-Database and Cross-Server Data Access): After completing this chapter, you'll be able to

convert data between various types. You'll also be able to access the data from various databases on the same or different servers.

Chapter 13_(Programmability): At the end of this chapter, you'll be able to package your T-SQL program in the form of programmability objects such as Stored Procedures, Functions, Triggers and Views etc. Each of these programmability objects have specific purpose. You'll be able to decide when to use what and how.

Chapter 14_(Deployment): You'll learn the various deployment methods, and its pros and cons. At the end of this chapter, you'll be able to choose the right deployment method for you.

Downloading the coloured images:

Please follow the link to download the
Coloured Images of the book:

<https://rebrand.ly/437226>

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to

the eBook version at www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at business@bpbonline.com for more details.

At you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

BPB is searching for authors like you

If you're interested in becoming an author for BPB, please visit www.bpbonline.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

The code bundle for the book is also hosted on GitHub at In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at Check them out!

PIRACY

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at business@bpbonline.com with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit

REVIEWS

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit

Table of Contents

1. Getting Started

Structure

Objectives

Database

Table, row, and column

DBMS

RDBMS

De-normalization

Normalization

Entity Relationship (ER).

Constraints

Primary Key.

Foreign key.

Trigger

ACID

Transaction

Atomicity (A).

Consistency (C).

Isolation (I).

Durability (D).

T-SQL

SQL

T-SQL commands

Data Definition Language (DDL).

Data Manipulation Language (DML).

Transaction Control Language (TCL).

Data Control Language (DCL).

[SQL Server physical architecture](#)

[Quick summary of SQL Server physical architecture](#)

[SQL Server version and editions](#)

[SQL Server download and installation](#)

[SQL Server Management Studio \(SSMS\).](#)

[Getting started with SSMS](#)

[Object Explorer](#)

[Query editor](#)

[Parse \(Ctrl + F5\).](#)

[Execute \(F5\).](#)

[Results to Text \(Ctrl + T\).](#)

[Results to Grid \(Ctrl + D\).](#)

[Results to File \(Ctrl + F\).](#)

[Comment out the selected lines \(Ctrl + K + C\).](#)

[Uncomment out the selected lines \(Ctrl + K + U\).](#)

[Increase indent](#)

[Decrease indent](#)

[Conclusion](#)

[Points to remember](#)

[Multiple choice questions](#)

[Answers](#)

[Questions](#)

[Key terms](#)

[2. Tables](#)

[Structure](#)

[CREATE DATABASE](#)

[USE DATABASE](#)

[CREATE SCHEMA](#)

[Data types](#)

[String](#)

[Binary.](#)

[Numeric \(non-decimal values\).](#)

[Numeric \(decimal values\).](#)

[Approximate floating point numeric data types](#)

[Date and time](#)

[Boolean](#)

[More on data types](#)

[CREATE TABLE](#)

[Three-part naming](#)

[ALTER TABLE](#)

[DROP TABLE](#)

[What is NULL?](#)

[Primary Key constraint](#)

[NOT NULL constraint](#)

[UNIQUE constraint](#)

[Primary Key versus UNIQUE constraint](#)

[DEFAULT constraint](#)

[CHECK constraint](#)

[Foreign Key constraint](#)

[CASCADING in Foreign Key.](#)

[Conclusion](#)

[Points to remember](#)

[Multiple choice questions](#)

[Answers](#)

[Questions](#)

[Key terms](#)

3. Index

[Structure](#)

[Objective](#)

[Index](#)

[Rowstore indexes](#)

[Clustered index](#)

[Non-clustered index](#)

[Covering index](#)

[Filtered index](#)

[Unique index](#)

[Columnstore indexes](#)

[Clustered columnstore index](#)

[Non-clustered columnstore index](#)

[Filtered columnstore index](#)

[DROP INDEX](#)

[Indexes in practical](#)

[Conclusion](#)

[Points to remember](#)

[Multiple choice questions](#)

[Answers](#)

[Questions](#)

[Key terms](#)

4. DML

[Structure](#)

[Objective](#)

[INSERT statement](#)

[SELECT statement](#)

[SELECT *](#)

[SELECT-specific columns](#)

[Filtering](#)

[Wildcard search](#)

[Range search](#)

[Sorting](#)

[Paging](#)

[TOP](#)

[UPDATE statement](#)

[DELETE statement](#)

[TRUNCATE statement](#)

[Magic tables \(inserted and deleted tables\)](#)

[OUTPUT clause with INSERT statement](#)

[OUTPUT clause with DELETE statement](#)

[OUTPUT clause with UPDATE statement](#)

[Conclusion](#)

[Points to remember](#)

[Multiple choice questions](#)

[Answers](#)

[Questions](#)

[Key terms](#)

[5. Built-In Functions - Part 1](#)

[Structure](#)

[Objective](#)

[Aggregate functions](#)

[GROUP BY](#)

[HAVING](#)

[SUM \(\)](#)

[COUNT \(\)](#)

[COUNT \(name>\)](#)

[COUNT \(*\)](#)

[COUNT \(DISTINCT name>\)](#)

[AVG \(\)](#)

[MIN \(\)](#)

[MAX \(\)](#)

[Aggregate functions with INSERT](#)

[String functions](#)

LEFT ().

RIGHT ().

SUBSTRING ().

REPLACE ().

LEN ().

LTRIM ().

RTRIM ().

LOWER ().

UPPER ().

SPACE ().

REPLICATE ().

REVERSE ().

CHARINDEX ().

QUOTENAME ().

STRING_SPLIT ().

Numeric functions

ABS ().

ISNUMERIC ().

ROUND ().

FLOOR () and CEILING ().

POWER ().

RAND ().

Date functions

GETDATE ().

ISDATE ().

DATEPART ().

DATEADD ().

DATEDIFF ().

Conclusion

Points to remember

Multiple choice questions

Answers

Questions

Key terms

6. Join, Apply, and Subquery.

Structure

Objective

JOIN

VLOOKUP in Excel vs JOIN

INNER JOIN

One-to-one relationship

One-to-many relationship

Many-to-many relationship

OUTER JOIN

LEFT OUTER JOIN

RIGHT OUTER JOIN

FULL OUTER JOIN

APPLY

CROSS APPLY

OUTER APPLY

Subquery.

Types of subqueries

Conclusion

Points to remember

Multiple choice questions

Answers

Questions

Key terms

7. Built-In Functions – Part 2

[Structure](#)

[Objective](#)

[Window functions](#)

[Aggregate functions](#)

[T-SQL example of aggregate windowing functions](#)

[Ranking functions](#)

[ROW_NUMBER \(\)](#)

[RANK \(\)](#)

[DENSE_RANK \(\)](#)

[NTILE \(\)](#)

[T-SQL example of ranking functions](#)

[Analytic functions](#)

[T-SQL example of analytic functions](#)

[Conclusion](#)

[Points to remember](#)

[Multiple choice questions](#)

[Answers](#)

[Questions](#)

[Key terms](#)

8. Dealing with XML and JSON

[Structure](#)

[Objective](#)

[Dealing with XML data](#)

[XQuery](#)

[Filtering data using XQuery](#)

[Other XQuery methods](#)

[OPENXML](#)

[Table to XML](#)

[Dealing with JSON data](#)

[OPENJSON](#)

[Other JSON functions](#)

[Table to JSON](#)

[Conclusion](#)

[Points to remember](#)

[Multiple choice questions](#)

[Answers](#)

[Questions](#)

[Key terms](#)

[9. Variables and Control Flow Statements](#)

[Structure](#)

[Objective](#)

[Variables](#)

[Declaring the variables](#)

[Assigning values to the variables](#)

[Reading values from the variables](#)

[Tiny, yet powerful keywords](#)

[BEGIN ... END](#)

[Semicolon \(;\).](#)

[RETURN](#)

[GOTO](#)

[WAITFOR](#)

[CASE statement](#)

[IF statement](#)

[WHILE loop](#)

[BREAK](#)

[CONTINUE](#)

[Conclusion](#)

[Points to remember](#)

[Multiple choice questions](#)

[Answers](#)

[Questions](#)

[Key terms](#)

[10. Temporary Tables, CTE, and MERGE Statement](#)

[Structure](#)

[Objective](#)

[Temporary tables](#)

[Common Table Expression \(CTE\).](#)

[MERGE command](#)

[Conclusion](#)

[Points to remember](#)

[Multiple choice questions](#)

[Answers](#)

[Questions](#)

[Key terms](#)

[11. Error Handling and Transaction Management](#)

[Structure](#)

[Objective](#)

[Error handling](#)

[TRY ... CATCH](#)

[Retrieving error information](#)

[@@ERROR](#)

[Transaction management](#)

[@@TRANCOUNT](#)

[XACT_STATE\(\).](#)

[Implementing explicit transaction](#)

[Conclusion](#)

[Points to remember](#)

[Multiple choice question](#)

[Answers](#)

[Questions](#)

[Key terms](#)

12. Data Conversion, Cross-Database, and Cross-Server Data Access

[Structure](#)

[Objective](#)

[Data conversion](#)

[CAST and CONVERT](#)

[PARSE](#)

[TRY CAST, TRY CONVERT, and TRY PARSE](#)

[Cross-database data access](#)

[Cross-server data access](#)

[Conclusion](#)

[Points to remember](#)

[Multiple choice questions](#)

[Answers](#)

[Questions](#)

[Key terms](#)

13. Programmability

[Structure](#)

[Objective](#)

[Dealing with NULL values](#)

[ISNULL](#)

[COALESCE](#)

[NULLIF](#)

[Dealing with IDENTITY value](#)

[@@IDENTITY](#)

[IDENT_CURRENT \(\)](#)

[SCOPE_IDENTITY \(\)](#)

[Dynamic SQL](#)

[SET NOCOUNT {ON | OFF}](#)

[IS and IS NOT](#)

[EXISTS and NOT EXISTS](#)

[EXISTS](#)

[NOT EXISTS](#)

[Programmability objects](#)

[Views](#)

[User-defined functions](#)

[Scalar user-defined functions](#)

[Table-valued user-defined functions](#)

[Stored procedures](#)

[Stored procedures with output parameters](#)

[Stored procedures with the default parameters](#)

[Getting hands dirty with stored procedures](#)

[Triggers](#)

[DDL triggers](#)

[DML triggers](#)

[Conclusion](#)

[Points to remember](#)

[Multiple choice questions](#)

[Answers](#)

[Questions](#)

[Key terms](#)

[14. Deployment](#)

[Structure](#)

[Objective](#)

[Deployment](#)

[Different methods of deployment](#)

[Generating and executing script](#)

[Generating Scripts](#)

[Backup and Restore](#)

[Conclusion](#)

[Points to remember](#)

[Multiple choice questions](#)

[Answers](#)

[Questions](#)

[Key terms](#)

[Index](#)

CHAPTER 1

Getting Started

This chapter will provide you with an overview of the fundamentals of relational database and SQL Server. You will understand the overview of the physical architecture of SQL Server. You will also understand the various versions and editions of SQL Server.

This chapter will also talk about what are SQL, T-SQL, and various T-SQL commands. Most importantly, you will learn about the SSMS, which is an **Integrated Development Environment** used for developing and managing the SQL Server database and writing T-SQL queries.

This chapter will act as a founding stone in the process of learning T-SQL.

Structure

In this chapter, we will cover the following topics:

Database

DBMS

RDBMS

ACID

T-SQL

SQL Server physical architecture

SQL Server version and editions

SQL Server download and installation

SQL Server Management Studio

Objectives

After reading this chapter, the readers will be able to:

Understand the fundamentals of the relational database and SQL Server.

Learn about SSMS and its various features.

Apprehend the concepts, terminologies, and SSMS.

Database

We will not talk about a lot of theories on database. However, it is important to know *what does a database mean?*

You would have heard words such as *data* and *database* many a times.

Data in simpler terms is information, which can be in various forms. Data could be information written on a piece of paper. It could be a file (of any type). It could be sound/video recording stored in a file. It could be educational records, medical records, legal records, employment records, and any other records you can think of.

Data is everywhere and everything is data.

As we all know, a cupboard is a storage unit, which we use to store our stuff. In the same way, a database is the storage unit to store data.

There are various types of databases in existence. Even Excel files, text files, and so on are databases. A folder is also a database that stores the files.

I remember my college days. We used to have floppy disks. Floppy disks had limited storage capacity in KBs. The maximum

capacity of the floppy disk was **1.44** Then, CDs came in the picture with a higher storage in 100 MB. CDs were replaced by DVDs, which had a storage capacity in GB. DVDs were later replaced by pen drives. Today, we can store 100 GB of data in a tiny pen drive.

The purpose of sharing this experience is to make you understand how data is growing day by day.

The world today is digital. Digital technologies and the Internet have truly made the world a family. We are connected with each other, irrespective of wherever we are with Internet. Recently, we were attacked by the worst pandemic of the recent times called

We could handle the situation in such a worst pandemic only because of technology. People are working from home. Companies are closing their offices and making their employees work from home permanently.

We can avail almost any service from our mobile phone, without stepping out of our home.

Just imagine the amount of data that is being produced today! You would have also experienced that Google, Facebook, and so on are selling ads to its consumers based on their search history, buying patterns, and so on. Amazon is suggesting the products based on the customers' buying choices and patterns. We are able to predict the monsoon, track the storms, and take precautionary measures to save precious lives.

This would not have been possible without data. Obviously, data just stored is nothing, unless mined to make purposeful information out of it.

With such an increase in the adaptation of digital technologies and a drastic increase in the volume of data day by day, there was a need to have specialized databases to store, retrieve, and make meaningful analytics out of available data.

That is why databases were evolved further in DBMSs. We'll understand about DBMSs in the upcoming topics.

Table, row, and column

Have you seen an Excel file? It contains multiple sheets. Each sheet is a table. A sheet has multiple columns which are named as ..., and so on and rows ..., and so on. The number of rows supported generally depends on the version of Excel.

To summarize, **.xls** or **xlsx** file is a database that contains multiple tables in the form of sheets and each table is a collection of rows and columns. The following screenshot shows the Excel file:

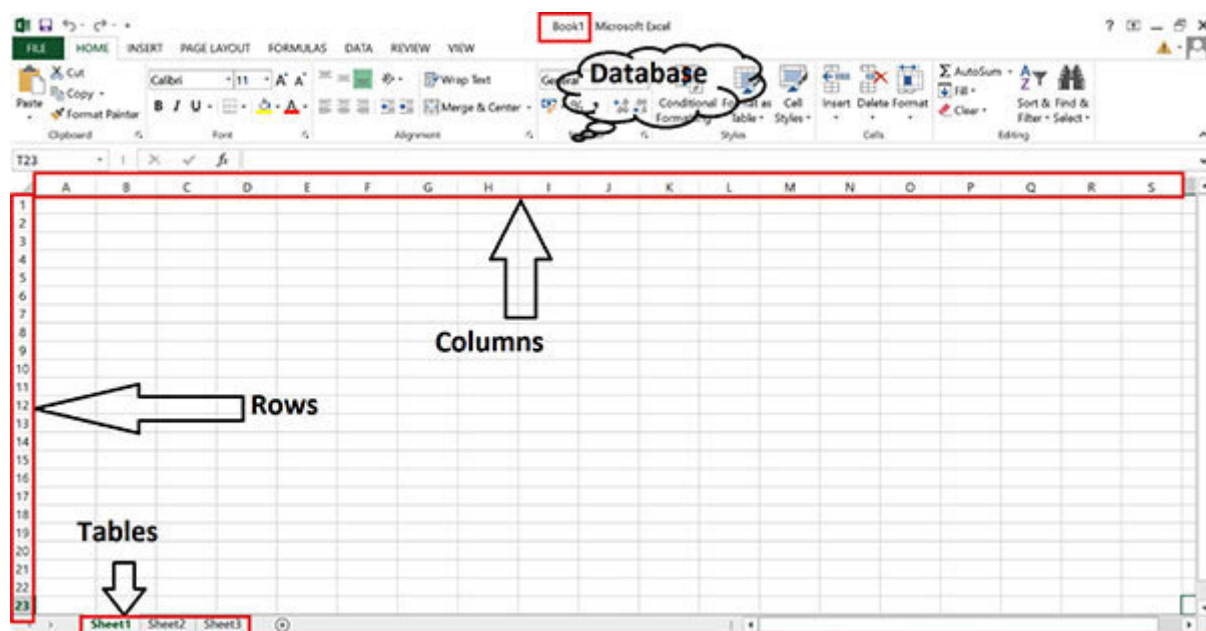


Figure 1.1: Table, row, and column in excel

From [figure](#) we understand that:

Book1 is a database.

and **Sheet3** are tables.

A, and so on are columns and and so on are rows.

Similarly, a database stores the data in the form of a table which is composed of rows and columns. The simple definition of a database and table could be a *database is a collection of A table is a collection of rows and*

Excel is not RDBMS.

DBMS

Databases were further evolved into **Database Management Systems**. DBMS is a packaged database solution with built-in handling to interact with both the database and user. These solutions allow users to connect to the remote database. Most of them have their own language. These languages are used to store the data in the database and to retrieve the stored data and perform analytics over it.

There are various kinds of DBMSs such as RDBMS, DDBMS, and so on. However, in this book, we will focus only on RDBMS.

RDBMS

RDBMS stands for **Relational Database Management**. These are specialized databases built to store data with proper relationships called

We will understand the relations along with few other important concepts using an example of a well-known two-wheeler manufacturer *Royal*

De-normalization

The following table shows the sample data of purchase orders. **PurchaseOrder** is used for recording the information of all the purchases. It has various attributes as can be seen in the following table. We have taken the sample data set. The actual data set can have many more attributes.

You'll also observe that few attributes such as and **ProductName** are repeated for each order. The customer **John** has three orders and each order has exactly the same customer information. This kind of dataset is called as *de-normalized* data.

The process of including all the required attributes in the same table is called As you can see in the following table, we have all the attributes of the purchase order in a single table:

PurchaseOrder								
Order ID	Transaction Date	Customer Name	Customer Address	Customer Mobile	ProductName	Quantity	Rate	Amount
1	01-Apr-18	John	Paris	1111111111	Electra	1	1,80,000.00	1,80,000.00
2	10-Apr-19	Kiva	London	2222222222	Thunderbird	1	2,20,000.00	2,20,000.00
3	20-May-19	Reema	India	3333333333	Classic	1	2,00,000.00	2,00,000.00
4	18-Jun-19	John	Paris	1111111111	Thunderbird	1	2,25,000.00	2,25,000.00
5	20-Dec-19	John	Paris	1111111111	Classic	1	2,10,000.00	2,10,000.00

Table 1.1: *De-normalized PurchaseOrder table*

In the era of physical registers, one needs to fill the physical forms, and then it needs to be registered in the physical registers.

Each time a customer comes to buy a product, we used to get a physical receipt manually filled by someone at the shop. Carbon papers are used to keep the copy of each receipt so that in case the original is lost, details can be gathered from the old receipt book with the help of the carbon copy. However, if the receipt book itself is damaged/lost, there is no record at all.

This does not happen in digital era. Repetition has gone. A customer who is registered once does not need to register again. Whenever we go for shopping, we just tell our customer ID, mobile number, and so on to the shopkeeper. If our original bill is lost, we can get the duplicates. Now, the record once written to the database stays there forever, unless intentionally removed.

Let us come back to our example:

You can see the **ProductName** are repeated for each row. *Does it make sense repeating the same data again and again?*

Let us analyze the dataset more closely:

and **CustomerMobile** are related to the customer. **ProductName** is related to the product.

Can't we separate these attributes to other tables? Yes, we can and we should!

Normalization

The process of dividing the data into multiple tables with the help of a primary key and foreign key is called

Normalization is the opposite of de-normalization. We divide a dataset into multiple smaller datasets and when we need the combined output, we join them. We'll learn about joins in [Chapter 6, Join, Apply, and](#)

We will talk about the primary key and foreign key in detail in the upcoming topics and chapters. However, we will learn about these terminologies a bit using the following example and explanation:

CustomerID of the **Customer** table is the **Primary Key** and **CustomerID** of the **PurchaseOrder** table is the **Foreign**. Similarly, **ProductID** of the **Product** table is the **Primary Key** and **ProductID** of the **PurchaseOrder** is the **Foreign**.

The Primary key table is also called the *referenced*. For example, the **Customer** and **Product** tables have primary keys; hence, it's a referenced table.

The Foreign key table is also called the *referencing*. For example, the **PurchaseOrder** table has a foreign key; hence, it's a referencing table.

Normalization helps to avoid the data redundancy. Redundant data is the data that is repeated again and again. Similarly, if there is any change in any attribute of a customer or product, then without normalization such an attribute needs to be modified across all the purchase orders pertaining to that particular customer or product.

Let us now separate the customer-related attributes to the **Customer** table and product-related attributes to the **Product** table. So now, we'll have three tables each for and

The Customer table stores the details of the customers as shown in [table](#)

Customer			
CustomerID	CustomerName	CustomerAddress	CustomerMobile
1	John	Paris	1111111111
2	Kiya	London	2222222222
3	Reema	India	3333333333

Table 1.2: Customer table

The Product table stores the details of the products as shown in [table](#)

Product	
ProductID	ProductName
1	Electra
2	Thunderbird
3	Classic

Table 1.3: *Product table*

The **PurchaseOrder** table stores the details of purchase orders as shown in [table](#)

PurchaseOrder						
OrderID	TransactionDate	CustomerID	ProductID	Quantity	Rate	Amount
1	01-Apr-18	1	1	1	1,80,000.00	1,80,000.00
2	10-Apr-19	2	2	1	2,20,000.00	2,20,000.00
3	20-May-19	3	3	1	2,00,000.00	2,00,000.00
4	18-Jun-19	1	2	1	2,25,000.00	2,25,000.00
5	20-Dec-19	1	3	1	2,10,000.00	2,10,000.00

Table 1.4: *Normalized PurchaseOrder table*

What is the purpose of having these tables and why separate tables for each?

You can see that the **PurchaseOrder** table has the **CustomerID** and **ProductID** columns. **CustomerID** is from the **Customer** table which is a unique identification number of a customer. The same goes to **ProductID** is a unique identification of a product.

Once a customer/product is added in the database, there is no need to add it again. Simply choose the desired **Customer** and **Product**, and you can place the Similarly, if there is any change in any attribute of **Customer** such as the address or mobile, and so on, it just needs to be updated in the **Customer** table.

Imagine if we did not have a separate table for the customer, and instead, we had all the customer-related attributes in the **PurchaseOrder** table. In that case, the same customer information would have been repeated for each order of a customer. A change of a customer attribute would result in major modification across the multiple rows of the **PurchaseOrder** table.

As shown in [table](#) **CustomerID = 1** which belongs to **John** has **3** orders in the **PurchaseOrder** table. Whereas, we've only one row for **CustomerID = 1** which belongs to **John** in the **Customer** table as shown in [table](#)

In our example, we have taken only four columns in the **Customer** table. However, in reality, there could be many more columns holding various information pertaining to customers and products.

More information on database normalization can be found in the official Microsoft documentation at the following URL:

<https://docs.microsoft.com/en-us/office/troubleshoot/access/database-normalization-description>

Entity Relationship (ER)

If we refer to our example as mentioned in [tables](#) **CustomerID** is the relation between the **Customer** and **PurchaseOrder** tables. Similarly, **ProductID** is the relation between the **Product** and **PurchaseOrder** tables.

In the relational database terminology, a table is called as an *entity* and the relationship between tables is called **Entity Relationship** ER is denoted with the help of a primary key and foreign key.

Almost all the modern financial and ERP data is stored in relational databases.

Constraints

Constraints can be understood as the rules defined on the columns of a table to govern what kind of data is allowed. There are various kinds of constraints in the relational database which we will learn in [Chapter 2,](#) It is important to know that the primary key and foreign key are also constraints.

Primary Key.

It ensures that the column(s) on which the Primary Key is defined contains unique values across all the rows of the table. If you'll try to add duplicate values, then the rule will fail and result in an error. It is important to note that the Primary Key constraints can be created with the combination of multiple columns.

Foreign key.

It ensures that the column(s) on which the Foreign Key is defined always has the same values as available in the Primary Key. If you'll try to add a value in the Foreign Key column(s), which is not available in the Primary Key, then the rule will fail and result in an error. The Foreign Key columns(s) should always reference a table having the Primary Key. Generally, the Primary Key is defined only on one column. However, if it is defined with a combination of multiple columns, then Foreign Key should be also defined on the same number of columns. The number of columns in both referencing and referenced tables should be the same.

The Foreign Key column(s) may contain NULL values, if the columns are marked nullable. Whereas, the Primary Key column(s) can never have the NULL values.

Trigger

A trigger is a kind of programmability object that can be automatically triggered upon an action performed on a table. Such actions are generally used to write a new record, modify an existing record, or delete an existing record. We will talk about triggers in detail in [*Chapter 14, Trigger and Stored*](#)

ACID

You would be wondering why I am talking about Chemistry while discussing a database which is a Computer Science subject. *Just kidding!*

ACID is an acronym of and These are the distinctive properties of a relational database.

We will understand each of these. But before that, let's understand *what Transaction is?*

Transaction

A transaction in the database world can be understood as a set of instruction(s). In the instruction is nothing but the T-SQL statement. Every individual T-SQL statement is an implicit transaction, in the absence of a definition of an explicit transaction. In order to get an overview of ACID properties, let's take a look at this simple example of a transaction. We'll talk about implicit and explicit transactions more in detail in [Chapter 11, Error Handling and Transaction](#)

Suppose we have the following instructions. Each of them is a T-SQL statement. All of these instructions are embedded in an explicit transaction. But in the absence of an explicit transaction embedding these instructions, each of the following instructions would be an individual implicit transaction:

Instruction 1: Read all students from the **Score** table who has scored beyond

Instruction Count the number of records from *Instruction*

Instruction Rank each student as per their score in descending order. The highest score means *Rank*

Instruction Show the count of rank holders from *Instruction 2* and the name and rank of each rank holders from *Instruction*

Let us understand each property of ACID.

Atomicity_(A)

Atomicity means surety that either all the instructions within a transaction will succeed or none of them. If either of the instruction of a transaction fails, then the whole transaction will fail.

If a transaction contains only a single instruction (T-SQL statement) such as or then depending on the respective T-SQL statement either all the rows will be inserted, updated, or deleted, or none of the rows will be inserted, updated, or deleted.

Consistency_(C)

Ensure that all data will be consistent. This means all the data will pass through the defined rules, including constraints and triggers.

Isolation (I).

Isolation offers assurance that no transaction will affect another transaction and each of them will run in isolation.

Durability_(D)

Durability guarantees that once a transaction is committed (successful), then it will remain in the database permanently.

T-SQL

One needs to learn T-SQL in order to become a Microsoft SQL Server database developer or an administrator. Before we understand T-SQL, let us understand *what is SQL?*

SQL

SQL stands for **Structured Query**. It is adapted by major RDBMSs. However, few RDBMSs have their own version of SQL such as PL/SQL is the proprietary language of Oracle and T-SQL is the proprietary language of Microsoft SQL Server.

SQL is an ANSI/ISO standard, but there are multiple versions. ANSI stands for **American National Standards Institute** and ISO stands for **International Organization for**

All such variants of SQL need to support the major commands in order to comply with the standard.

T-SQL commands

There are different kinds of T-SQL commands as shown in [figure](#). There are other commands as well that we'll discuss in the upcoming chapters:

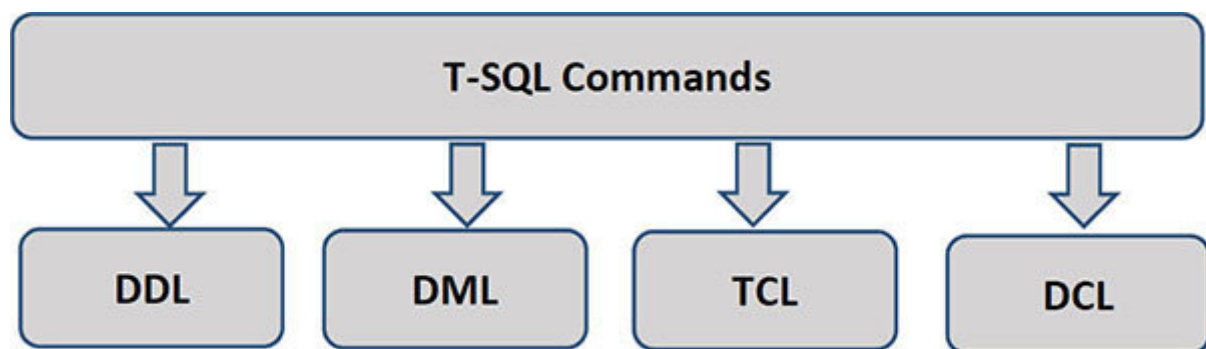


Figure 1.2: T-SQL commands

Let us understand each of these categories of T-SQL commands.

Data Definition Language (DDL)

These commands are and

They are used to create, alter, and drop databases, tables, constraints, indexes, stored procedures, views, triggers, functions, and all other SQL Server objects.

ALTER is nothing but modifying the object in the database. **DROP** is nothing but removing/deleting the object from the database.

Data Manipulation Language (DML)

These commands are and

They are used to insert and update records in the tables, delete records from the tables, and select records from the tables.

In summary, these commands are used to read, add, modify, and delete data/records described as follows:

nothing but writing fresh data/records in the tables.

nothing but modifying the data/records in the nothing but removing the data/records from the tables.

nothing but reading the data/records from the tables.

used only on the tables. used on tables, views as well as functions. We'll learn about view and function in [Chapter 10, View and User-Defined](#)

Transaction Control Language (TCL)

This is used to manage the transaction. We discussed about the transaction in the *ACID* section. These commands are **BEGIN COMMIT ROLLBACK** and **SAVE**

Data Control Language (DCL)

This is used to manage the database security such as granting or revoking access. These commands are **GRANT** and

SQL Server physical architecture

The following diagram shows the high-level physical architecture of Microsoft SQL Server:

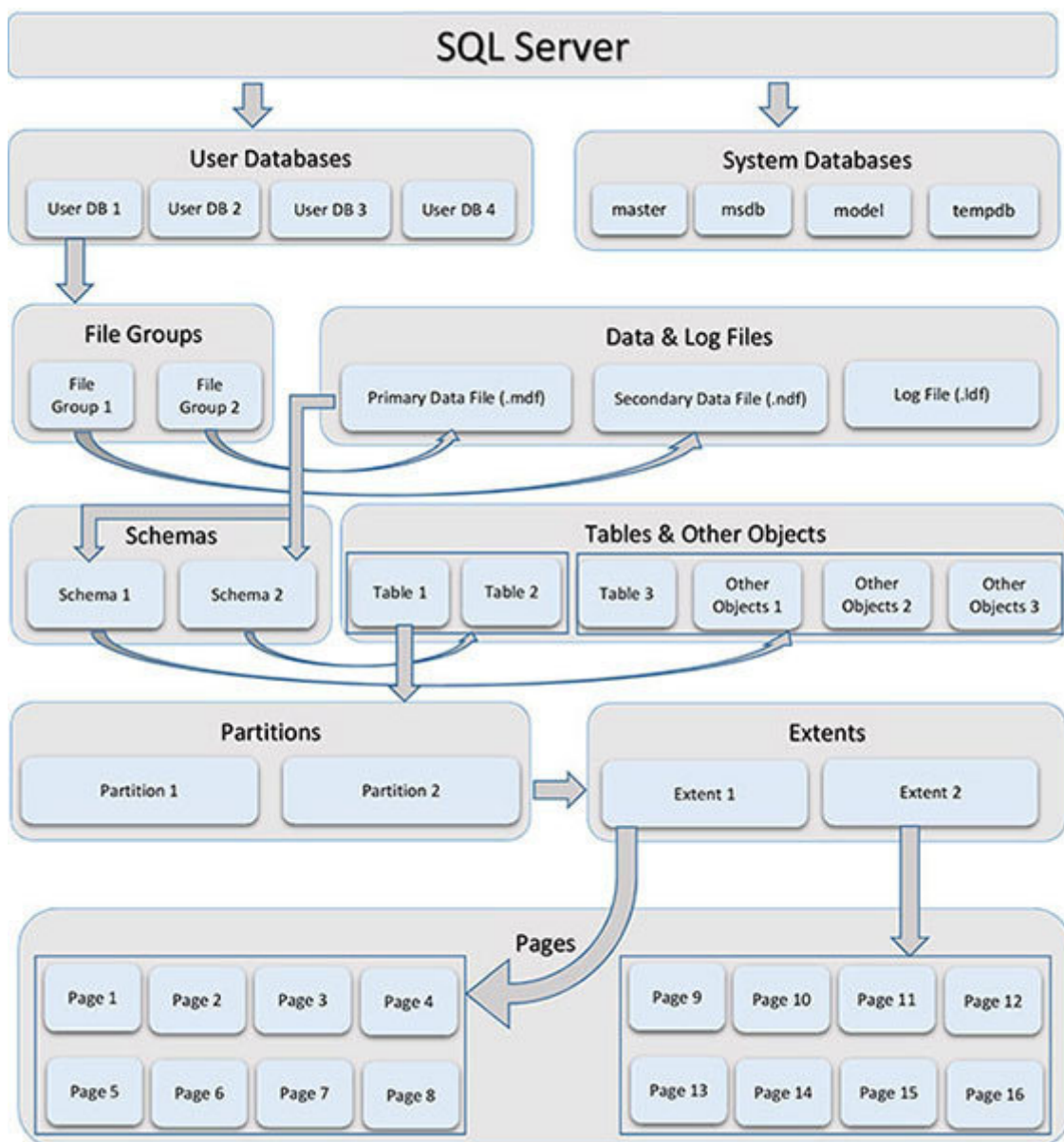


Figure 1.3: SQL Server physical architecture

Let us understand each of the components from the physical architecture:

Pages are the lowest storage unit in SQL Server and each page is of size 8 KB.

Each extent is the collection of 8 pages. Extents are of two types:

Mixed Shared by multiple tables. Holds index as well as data pages.

Uniform Belong to only one table. It can either hold data page or an index page.

1 MB is 128 pages in SQL Server terminology.

Partition is the logical grouping of smaller datasets created out of large dataset by the way of partitioning. Partitioning can be very useful if you are dealing with the high volume of data. Especially, in read loads.

Consider a scenario of a table with 100 billion rows of the past 10 years. Reading from single partition of 100 billion rows can be very costly. Suppose each year has approximately 10 billion rows. We can partition the table by year using **Date** column to have 10 partitions. Now when we will query this table then the read will happen from the respective partitions instead of the whole set of 100 billion rows.

A table is a collection of rows and columns. We already discussed about the table in the preceding topic *Table, Row, and* All the data in SQL Server is logically stored in tables but physically in the data file. A user trying to add, modify, delete, or read a record actually deals with the table. However, the data is physically stored on a disk in the form of data files.

You cannot have a row with a size more than 8 KB. Columns with variable-length data types such as LOB types are given an exception to the 8 KB rule and are not considered in the 8 KB. LOB types can hold up to 2 GB of data in a cell.

A schema is a logical grouping of SQL Server objects. A schema can have multiple objects. For example, if the database is created for an organization having multiple departments, then objects pertaining to each department can be placed in a schema.

Objects are tables, stored procedures, functions, triggers, and views, and so on.

Database is a container that holds all kinds of objects. When we say object, then it also includes the underlying data of the table. There are two types of databases in SQL Server:

System These are special kinds of databases used internally by SQL Server. They come along with the SQL Server installation:

Used to record SQL Server instance level information, information about existing databases, and location of their respective data and log files. It also holds info about all the objects in the database. This database is so important that the SQL Server instance would not start if it is not available.

Act as a model for all databases created on SQL Server. Place all the generic objects in this database and all such objects will be auto-created upon creation of the new database.

Used by SQL Server agent to schedule the jobs, alerts. It also keeps a log of job execution and DB mails.

SQL Server Agent is a built-in application offered by SQL Server to schedule the jobs. A job can execute the T-SQL queries and other SQL Server objects. It can also execute the SSIS package. SSIS is Extract, Transform, and Load (ETL) tool.

Used by SQL Server to store the temporary objects such as tables, stored procedures, functions, views, and so on. Table variables also use the **tempdb** is recreated every time SQL Server is started/re-started. We'll discuss about temporary tables and table variables in the succeeding chapters.

User These are the databases created by us, the users.

Data Stores the data in the form of pages logically organized in extents. There are two types of data files: primary data file and secondary data file

Log Stores log of all modifications done using DDL or DML statements. This is the main organ of SQL Server that ensures data written is ACID compliant. The extension of log file is

File Holds data files (primary and secondary). Each database has one default file group but there could be multiple file groups. File groups can be mapped to different drives.

Quick summary of SQL Server physical architecture

A database can have multiple file groups. A file group can have multiple data files. A data file can have multiple schemas. A schema can have multiple tables/objects. A table can have multiple partitions. A partition can have multiple rows, and each row can have multiple columns. A page can hold one or more rows.

SQL Server version and editions

At the time of publishing this book, SQL Server 2019 was released by Microsoft which is the latest version. Developer Edition (64 bit) of SQL Server 2019 is used all across this book

SQL Server 2019 comes with various editions such as Enterprise, Standard, Web, Developer, and Express editions. All of them follows different licensing model.

More about these editions can be found at Microsoft documentation with the following URL:

<https://docs.microsoft.com/en-us/sql/sql-server/editions-and-components-of-sql-server-version-15?view=sql-server-ver15>

Developer and Express editions are the free editions offered by Microsoft.

The Developer edition offers full features but is licensed only for development and test. Production use is not allowed.

However, the Express edition has some feature limitations and is allowed to use in production. Small production workloads can leverage this edition.

SQL Server download and installation

Microsoft documentation at the following URL can be referred to download and install the SQL Server 2019 (including the free editions such as Developer and Express).

The documentation includes the step by steps explanation:

<https://docs.microsoft.com/en-us/sql/database-engine/install-windows/install-sql-server-from-the-installation-wizard-setup?view=sql-server-ver15>

SQL Server Management Studio (SSMS)

Microsoft SQL Server Management Studio 18 is shipped default with SQL Server 2019. However, it can be also downloaded and installed separately.

Each SQL Server installation installs both client and server. So, both client and server get installed on the same system. However, the client can be installed on different systems too. There could be multiple clients (installed on different machines) connecting to one server.

SSMS is a client. However, client can be various sources including application source code.

SQL Server Management Studio (SSMS) is a SQL Server client.

SSMS is the IDE or developer's tool to deal with SQL Server. With this tool, you can:

Create and manage databases and database objects.

Write and execute T-SQL queries.

Perform database administration activities.

In short, this is a tool to do everything in SQL Server.

Getting started with SSMS

Perform the following steps to start the SSMS and connect to your SQL Server instance:

Go to **Start** and type **SQL Server Management** SQL Server Management Studio will be shown as best match as shown in the following screenshot:

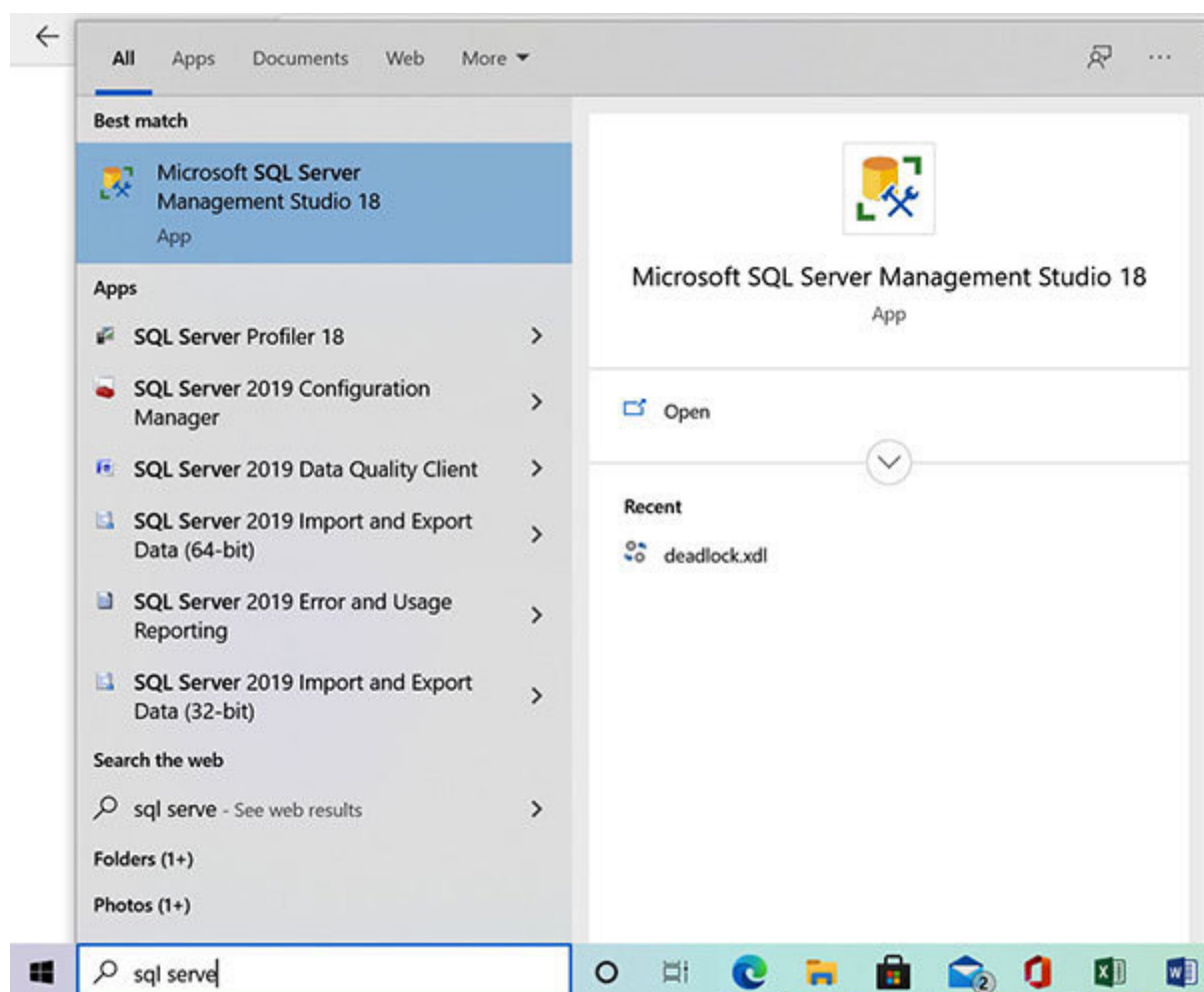


Figure 1.4: Starting SSMS

Click on Microsoft SQL Server Management Studio 18. Login screen will pop-up as shown in the following screenshot. Login screen has various fields as can be seen in the following screenshot:

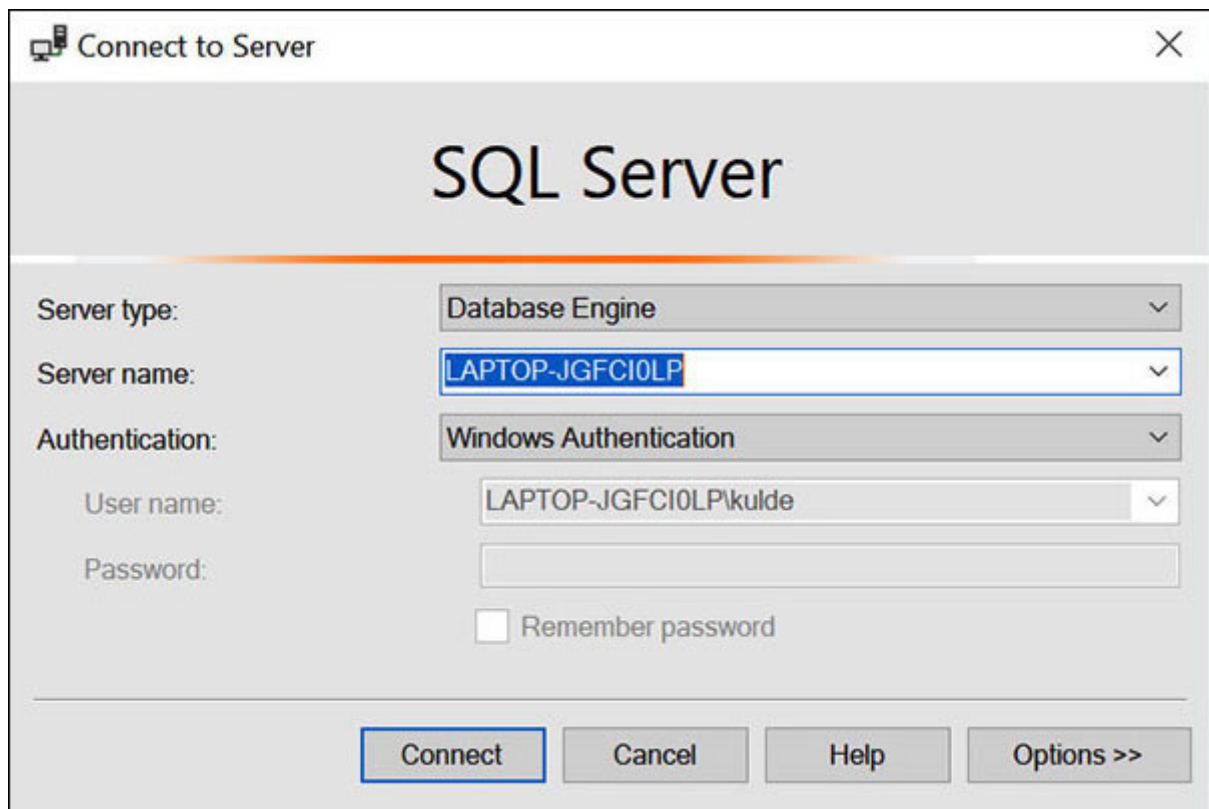


Figure 1.5: SSMS login screen

There are various options as shown in the following screenshot while selecting the server type, but we will focus only on **Database**. This option is used to deal with SQL Server databases:

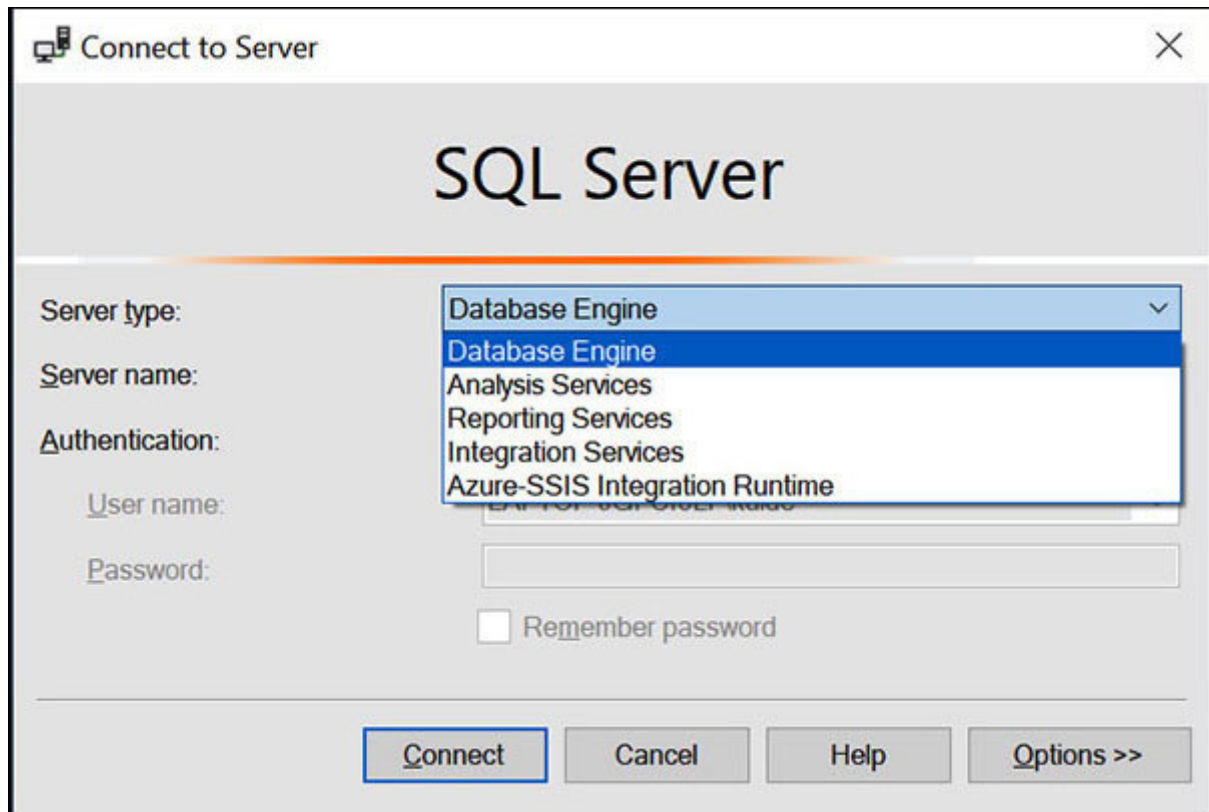


Figure 1.6: SSMS login screen server types

Other fields are:

Server This is name/IP of the server or name of the instance. If you are running it on a local system then you can even mention localhost. The name of instance can be specified during the installation.

There are various authentications as can be seen in the following screenshot. These settings can be configured at the time of installation. However, Windows authentication and SQL Server authentication are the most commonly used authentication types:

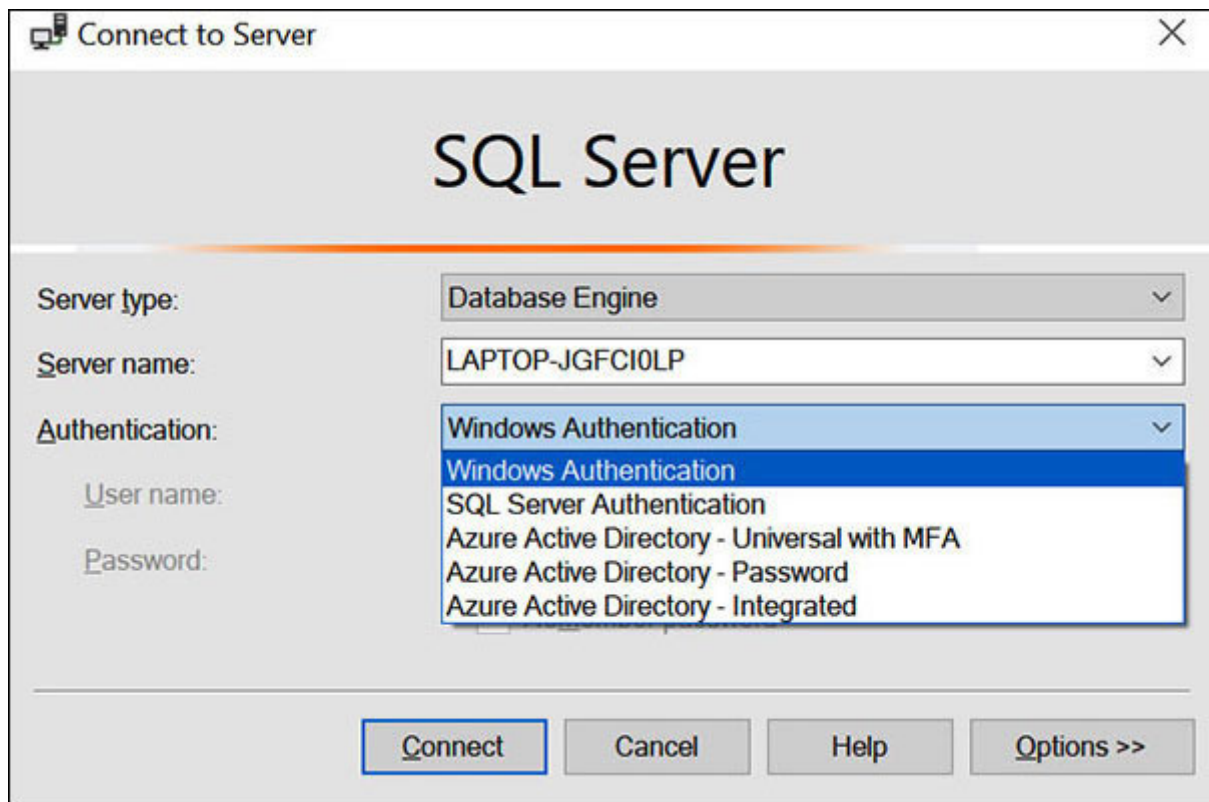


Figure 1.7: SSMS login screen authentication types

If you are running the SSMS with same Windows user as the one used for installation then you would not be asked for credentials. However, if the SQL Server was installed with different user and you have logged in with a different user then you won't be able to access the SQL Server, unless there is a SQL Server login created for the Windows user through which you are trying to connect to SQL Server.

sa is the default SQL Server login which is shipped with SQL Server installation. However, you have the option to specify your own password. It is also often referred to as **super**

Authentication and SQL Server login are advanced topics. These are part of SQL Server security, which by itself is a whole big subject. It is beyond the purview of this book. However, you can use either Windows or SQL Server authentication to log-in to your installed instance of SQL Server for the purpose of learning and practicing T-SQL.

User If you have chosen **Windows** and you are running the SSMS with the same Windows user as the one used for installation, then you won't be asked for credentials. You can directly click on the **Connect** button.

If you have chosen **SQL Server Authentication** then you need to provide the user name (if created) or with default user

You need to supply the password of user/default user

sa is the default user shipped with SQL Server installation.

Now since you have entered the necessary inputs, you can click on the **Connect** button. A screen similar to the following screenshot will get opened:

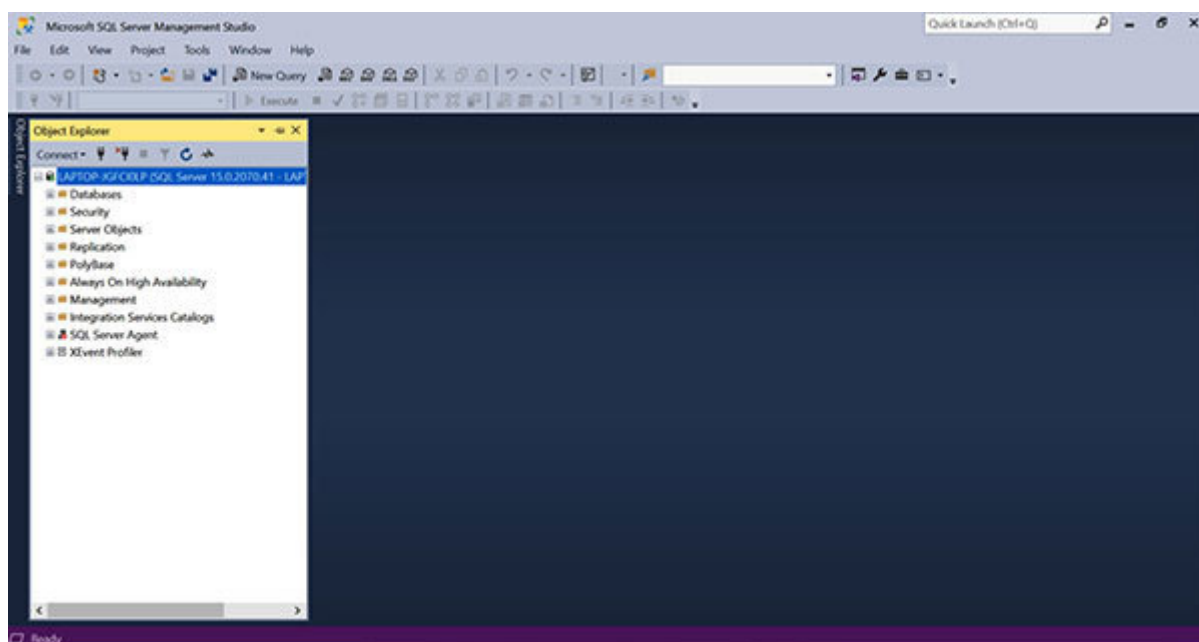


Figure 1.8: *SSMS post login screen*

SSMS is a very comprehensive IDE. But we will try to cover the key areas used in day-to-day work for writing T-SQL queries and developing and dealing with SQL Server databases.

Object Explorer

It is an important section within SSMS. You can see a lot of things under it. However, we will more specifically focus on **Databases** folder only. Refer to the following screenshot:

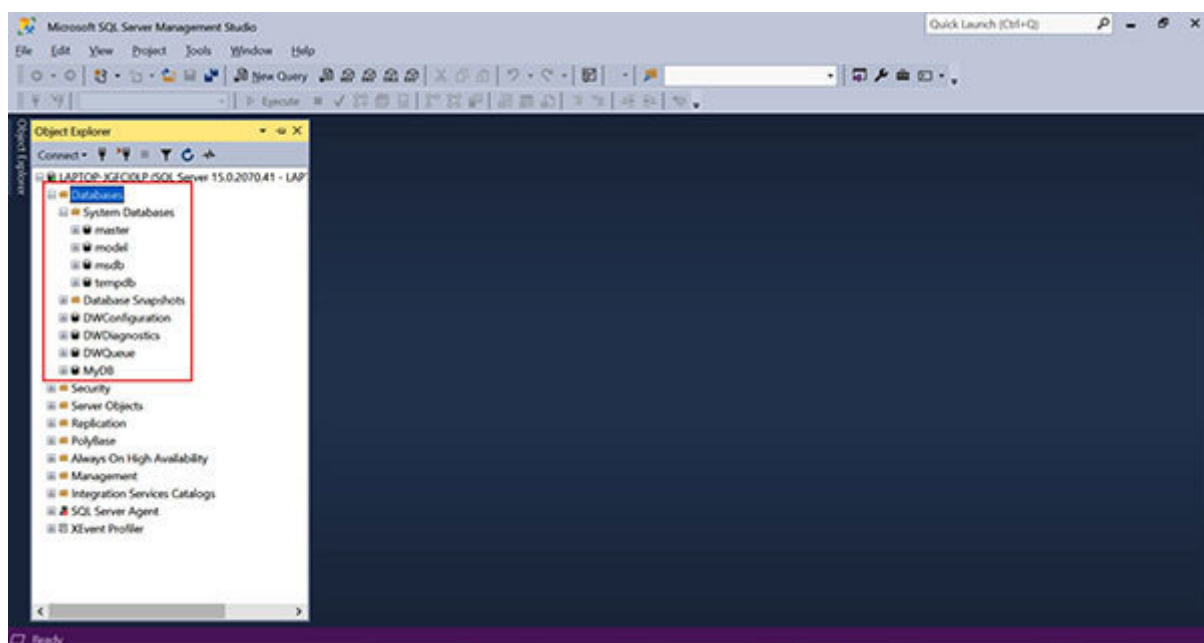


Figure 1.9: Object Explorer in SSMS

Under the **Databases** folder, you can see **System Databases** folder that contains all the system databases as we discussed in the preceding *SQL Server Physical Architecture* topic.

You can also see few other databases and These databases are shipped along with SQL Server 2019 installation and are created if you choose to install **PolyBase** feature. You can even see the

folder for **PolyBase** within Object Explorer. In my installation I chose to install PolyBase, hence these databases were created.

We will focus only on the T-SQL and items dependent to learn T-SQL. Any other feature including PolyBase is beyond the purview of this book.

You can also see a database This is user database that I created to show an example. You can create multiple user databases.

This is what we will learn in this book. *How to create user databases and database objects and how to query databases?*

System databases are shipped with SQL Server setup and internally used by SQL Server as we discussed in the preceding topic. Whereas user databases are created by the user.

Query editor

The next important section of our interest in SSMS is **Query** It can be opened either by clicking the **New Query** option or right click on the desired database and choosing the **New Query** menu option. Refer to the following screenshot:

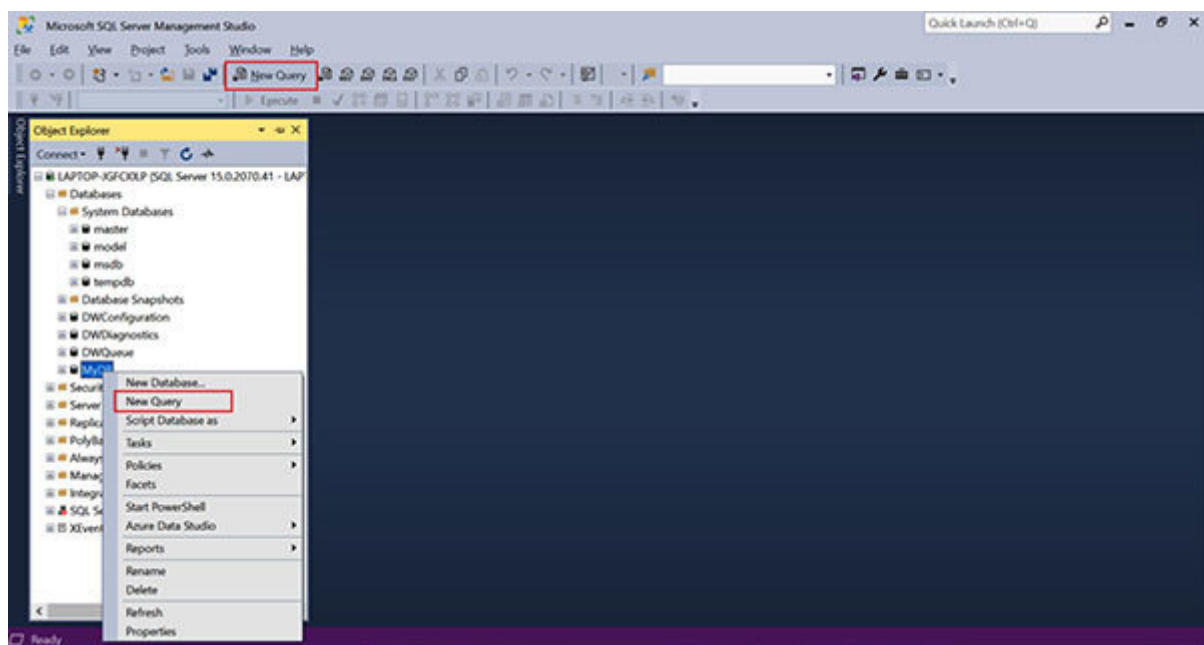


Figure 1.10: Opening a new query editor in SSMS

Window similar to the following screenshot will get open. The white area is called **Query Editor** where the T-SQL queries are written.

If database is not selected while clicking New Query, then Query Editor will get open with default database selection that is master as can be seen in [figure](#) Whereas if database was chosen while

clicking New Query, then Query Editor will be opened with that particular database selected. However, user has the option to change the database of the Query Editor by choosing the desired one as can be seen in [figure](#)

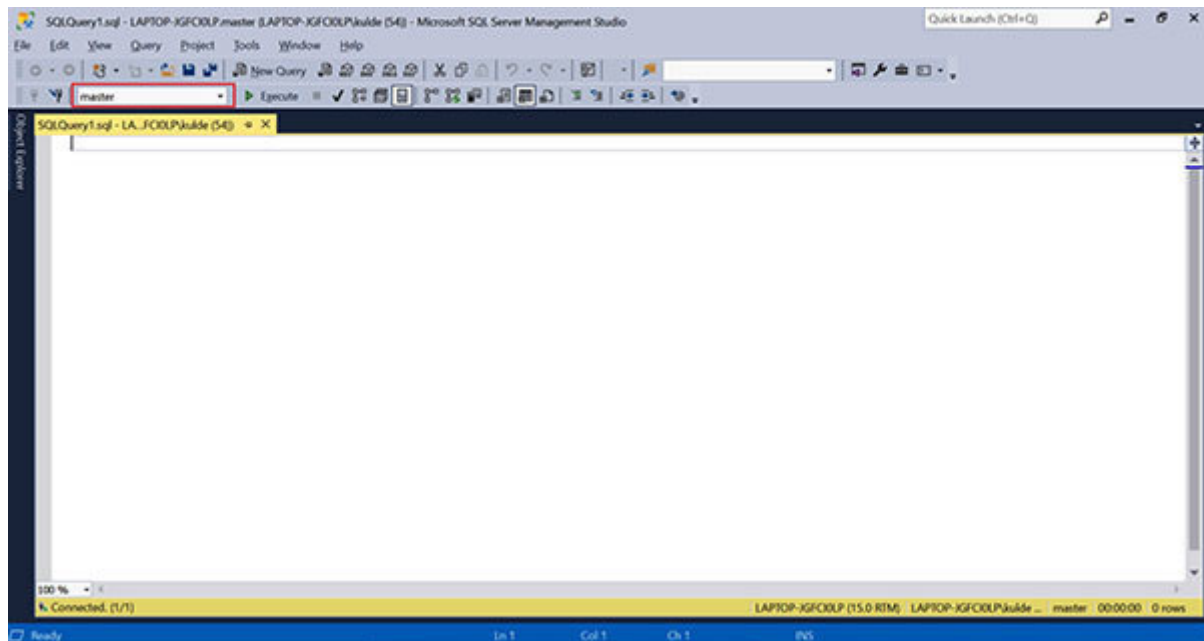


Figure 1.11: Database dropdown in SSMS query editor

Database can be changed in the Query Editor as shown in the following screenshot:

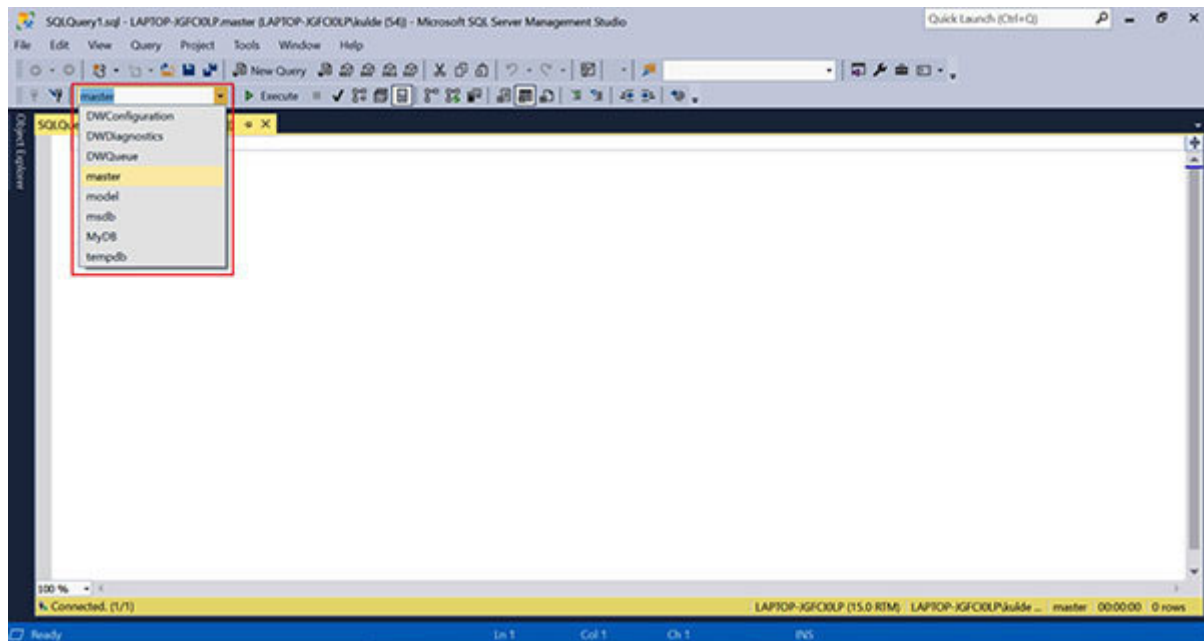


Figure 1.12: *Choosing database in SSMS query editor*

We will now understand the important options within SSMS which are used very frequently when you are working on SQL Server database or T-SQL.

Most of these options come in three variants. You can access them from the menu, toolbar, and using shortcut keys. It's always better to remember the shortcut keys.

Parse (Ctrl + F5)

The very first thing in any program is parsing. Parsing is a process that compiles the program and checks for the syntax. It will check *if your T-SQL code is syntactically correct?*

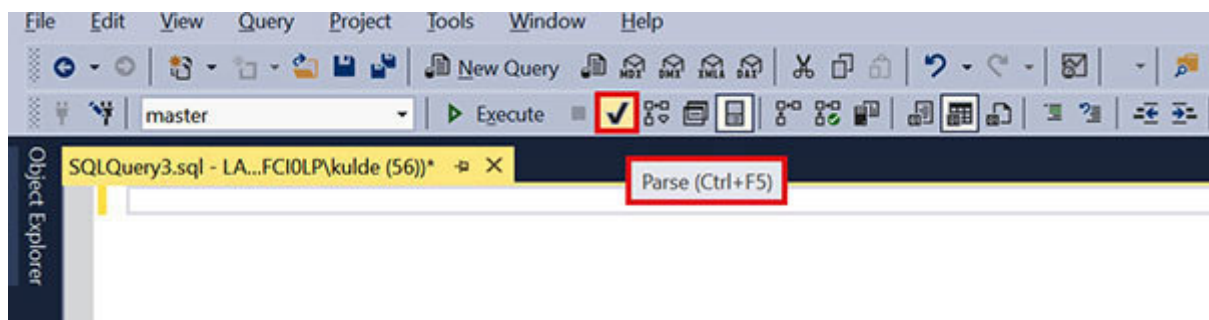


Figure 1.13: Parsing the query in SSMS

Execute (F5)

Execute is a process in which your T-SQL query is submitted to SQL Server Database Engine. SQL Server Database Engine executes the query and returns the result.

The result is seen in the result pane + Upon query execution completion result pane automatically gets populated. However, you can also open it using the menu option under the **Query** menu (as shown in [figure](#) or using the shortcut key *Ctrl* +

There are three ways in which the results can be seen. If you want result in any of these specific formats then you need to choose the desired option before executing the query:

Result to Text (Ctrl + T) – Refer [figure](#)

Result to Grid (Ctrl + D) – Refer [figure](#)

Result to File (Ctrl + Shift + F) – Refer [figure](#)

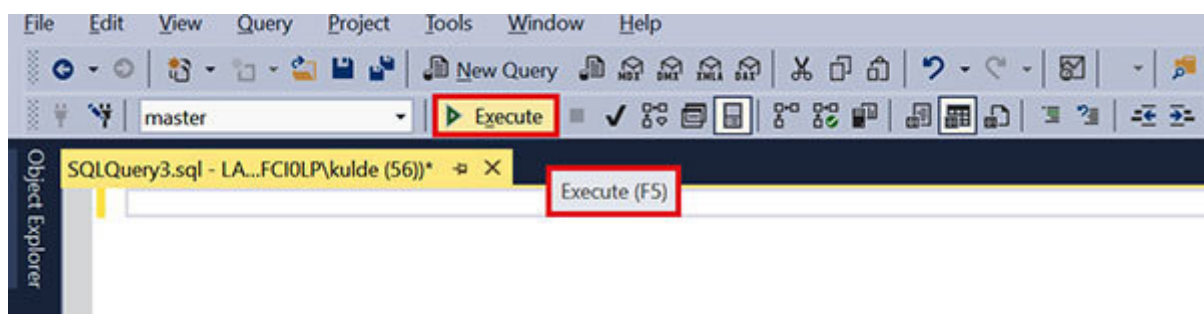


Figure 1.14: *Executing the query in SSMS*

SQL Server is an ocean. The more we will enter into it, the deeper it is. Similarly, for us we simply write query and execute it. But it is not that simple as we think. There are a set of processes involved in the execution of each query.

Database Engine and Query Optimizer is an advanced topic. We will not deviate from our goal of learning T-SQL. However, we'll understand a very high level of what happens behind the scene when the query is executed in SQL Server.

As we already understood in the preceding section of this book that SQL Server is based on client and server. There could be multiple clients connecting to the same server at the same time. We also discussed that SSMS is a client.

So, when the query is submitted through the client, the query is passed to Database Engine. Database Engine has various built-in processes which runs sequentially:

Submitted query is first passed through the parser. Parsing performs syntactical checks to make sure the query is syntactically correct. If it will find any error then it will be populated back to the client. If the query is syntactically correct then parse tree is generated.

Binding is also called This step accepts the parse tree and generates the Algebrizer tree after performing name resolution, type derivation, aggregate binding, and group binding.

Name resolution checks whether the objects are available and the user with whose context the query is being executed has the access to those objects.

Type derivation derives the type of each node in the parse tree. Aggregate and group binding determines and binds the aggregation to the appropriate select list.

Parsing produces parse tree and binding/Algebrizer produces Algebrizer tree. However, the common output of parsing and binding is also known as logical tree. Binding/Algebrizer is technically part of parsing itself.

Query Process accepts the logical tree and generates the possible execution plans. It then evaluates the cost of each plan and chooses the plan with the lowest cost. The plan is also stored in the cache for reuse called **Plan** Next time if the same query will be executed again then optimizer may reuse the existing plan, thus offering better query performance.

SQL Server Query Optimizer is cost-based optimizer. It means it will choose the plan with the lowest cost by maintaining a balance between optimization time and plan quality. For example, it doesn't make sense to spend 5 seconds to find the best plan for a query that can execute within 15 seconds. Similarly, if query

is going to take several minutes to execute then it really makes sense to spend that much time.

Query execution: Executes the query as per the chosen plan and returns the result to the client such as SSMS.

If the query was executed using T-SQL then the result will also be displayed in SSMS under the same **Query** windows in the result pane as per the result option (text/grid/file) selected before executing the query:

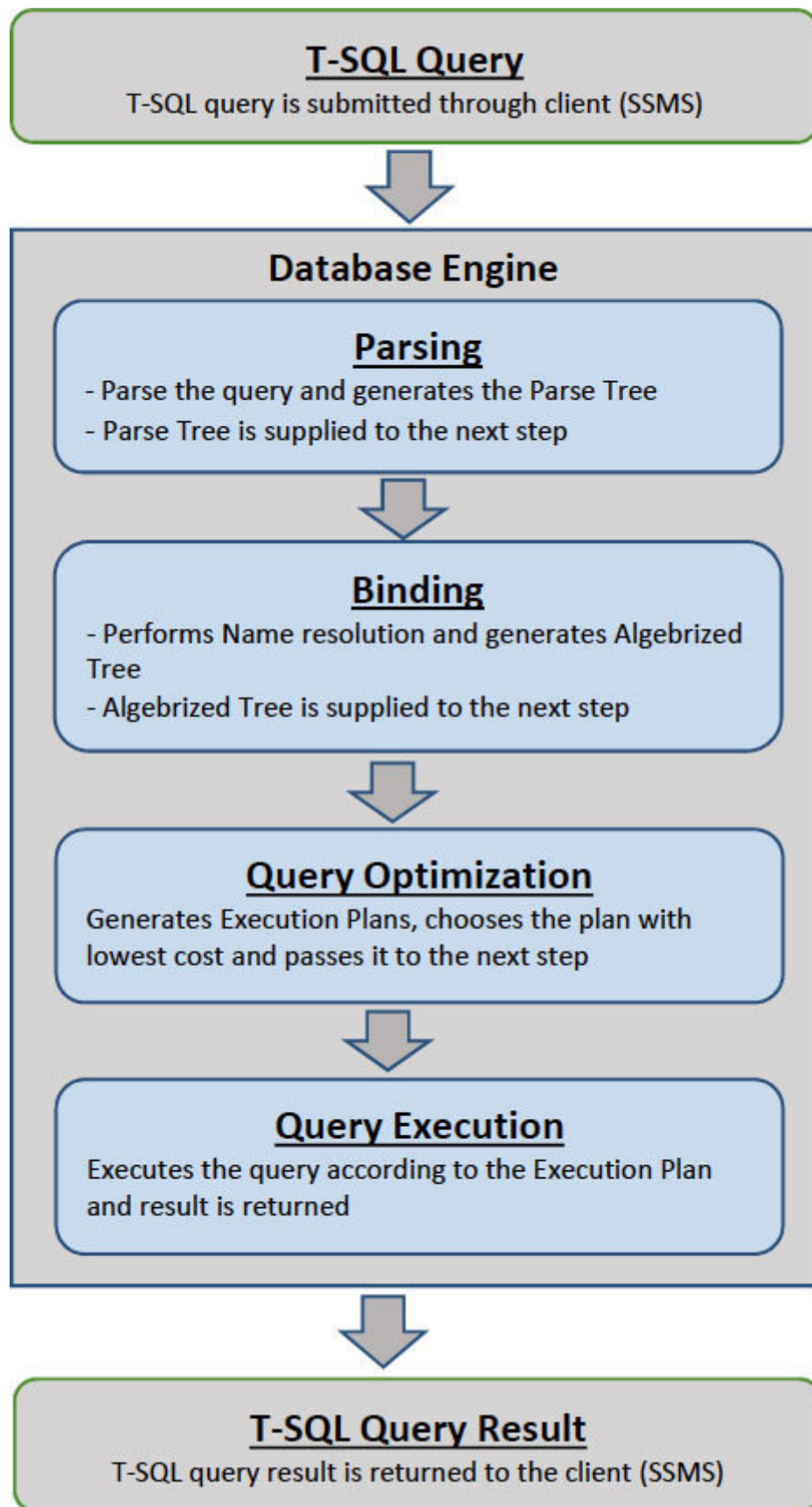


Figure 1.15: Query execution flow

The result pane can be seen by clicking the menu/shortcut key as shown in the following screenshot:

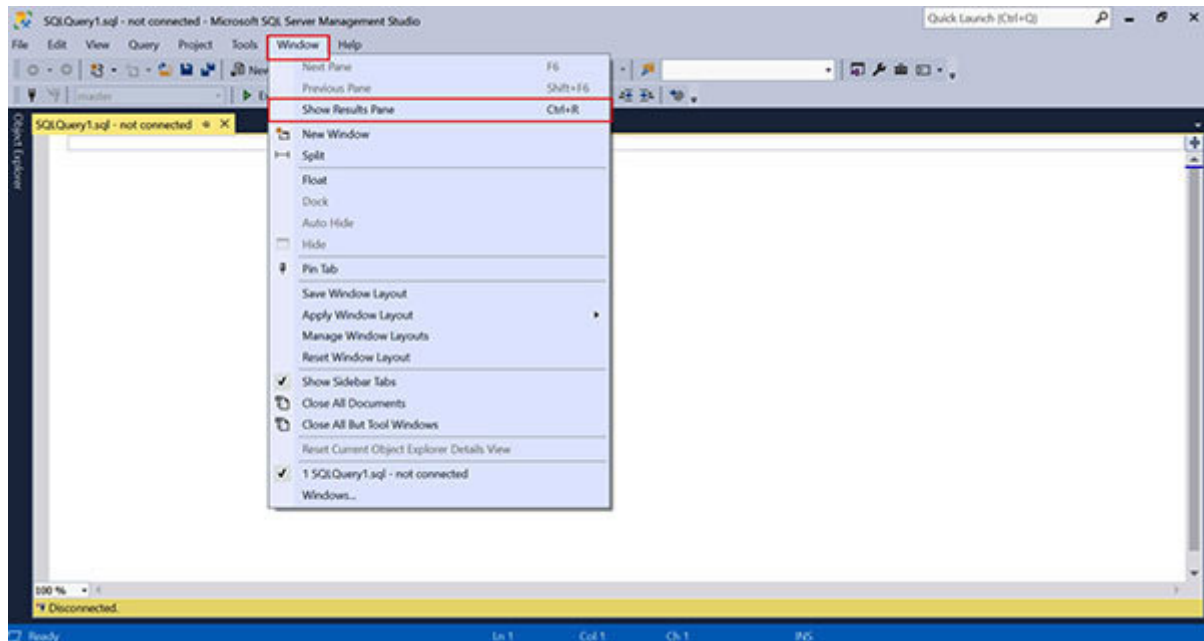


Figure 1.16: Window menu of SSMS

Results to Text (Ctrl + T)

If you want **Results to** then click on the toolbar button or click the *Ctrl + T* shortcut key as shown in the following screenshot. This feature returns the result in the result pane in the text format. You can go to the result pane, and search a particular text with the help of *Find* option +

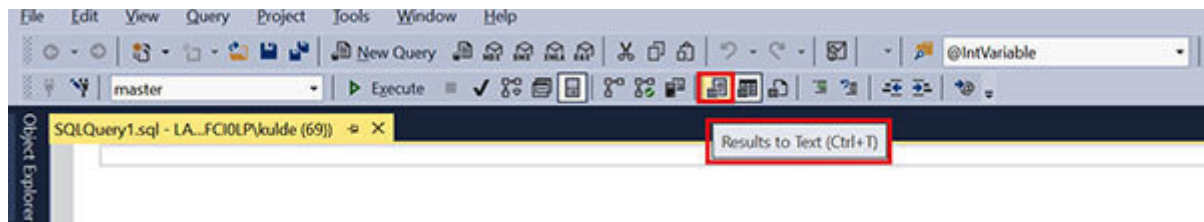


Figure 1.17: Result output to text shortcut in SSMS

Results to Grid (Ctrl + D)

If you want **Results to** then click on the toolbar button or click the *Ctrl* + *D* shortcut key as shown in the following screenshot. This is a default setting. Whenever you will run a query, the output will be returned in grid format in the result pane, unless you specifically choose any other option for the result. You can copy and paste the rows from the result to excel. You can copy the data from the result with or without header. There are two distinct options – **Copy** and **Copy with**. You can see these options when you right click on the result in the result pane. There are shortcuts too. *Ctrl* + *C* for and *Ctrl* + *Shift* + *C* for **Copy with**. **Copy** simply copies the row data, whereas **Copy with Header** copies the row data with column headers, as available in the result:

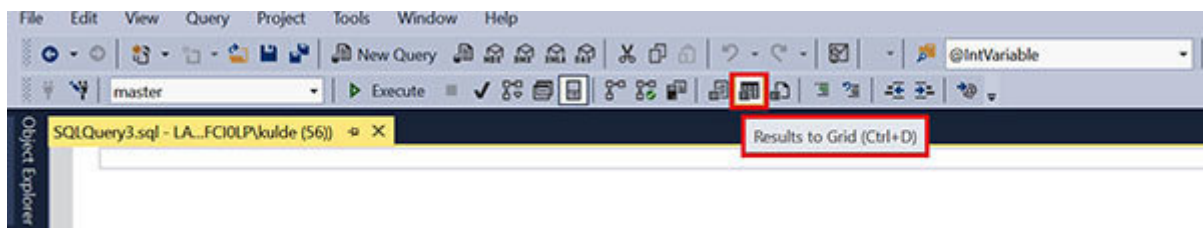


Figure 1.18: Result output to grid shortcut in SSMS

Results to File (Ctrl + F)

If you want **Results to** then click on the toolbar button or click the *Ctrl* + *F* shortcut key as shown in the following screenshot:

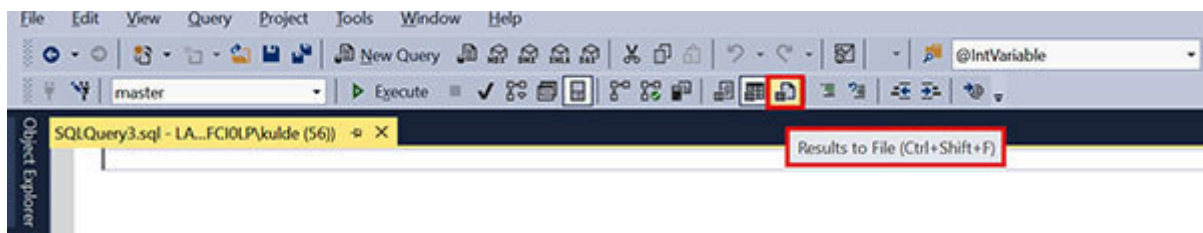


Figure 1.19: Result output to file shortcut in SSMS

Alternatively, you can also refer the **Query** menu for important commands as shown in the following screenshot:

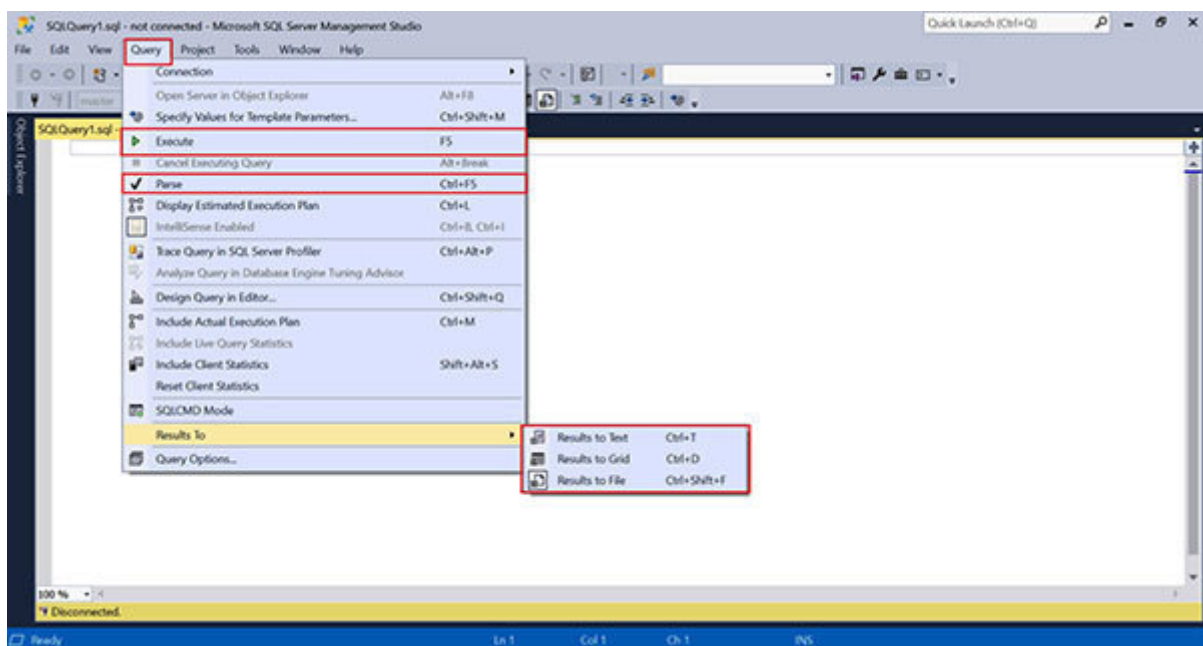


Figure 1.20: Result To menu options in SSMS

Comment out the selected lines (Ctrl + K + C)

Press *Ctrl* and then press *K* and This is shortcut key to comment out the selected lines of code. Alternatively, the mentioned toolbar button can also be used:

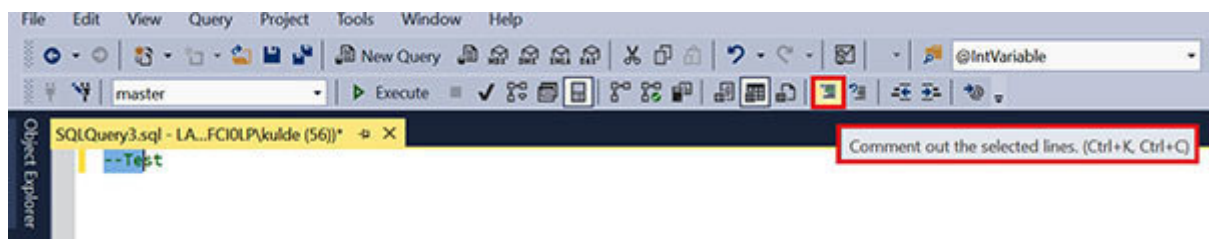


Figure 1.21: Commenting code in query editor of SSMS

Uncomment out the selected lines (Ctrl + K + U)

Press *Ctrl* and then press *K* and This is shortcut key to uncomment the selected lines of code. Alternatively, the mentioned toolbar button can also be used:

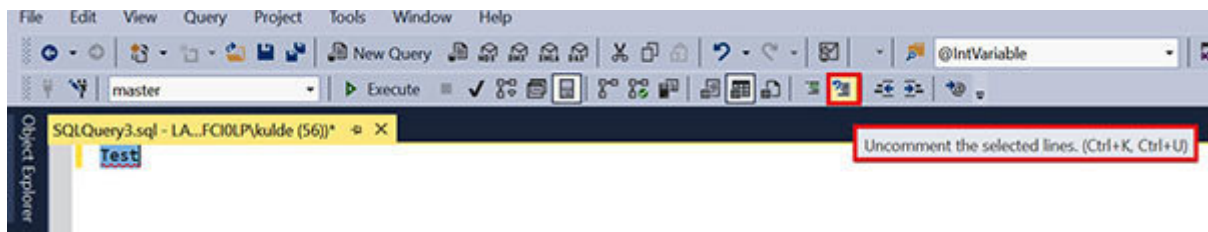


Figure 1.22: Uncommenting code in query editor of SSMS

Increase indent

Use this toolbar button to increase indent. Alternatively, *tab* key can also be used:

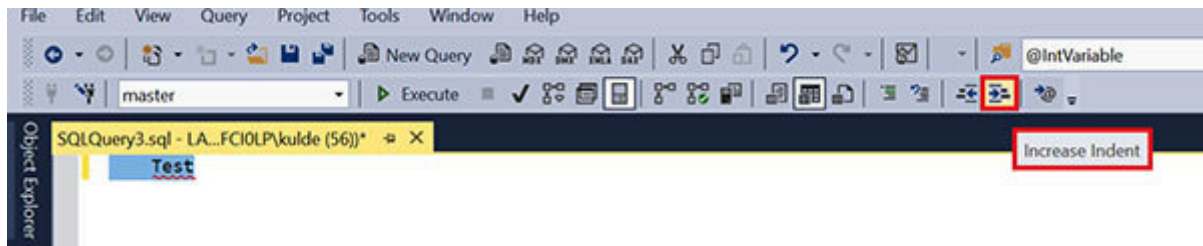


Figure 1.23: Increasing indent in query editor of SSMS

Decrease indent

Use this toolbar button to decrease indent. Alternatively, *shift + tab* shortcut key can also be used:

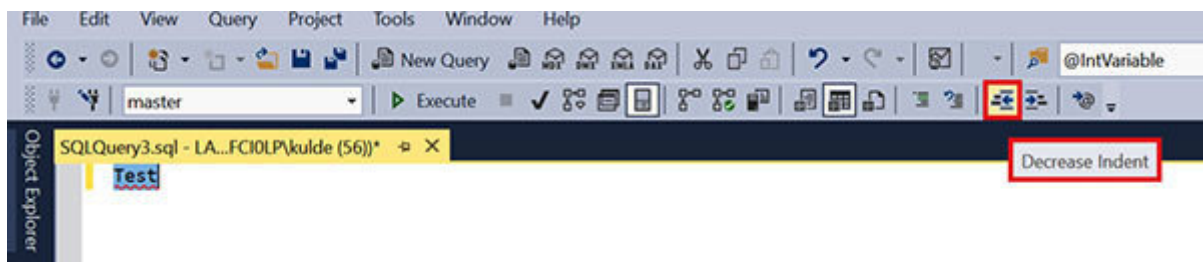


Figure 1.24: Decreasing indent in query editor of SSMS

Conclusion

In this chapter, we talked about database, tables, rows, and columns. We discussed about DBMS and RDBMS and understood ACID properties. We also discussed about SQL, T-SQL, and various types of T-SQL commands. We discussed about SQL Server physical architecture, version, and edition. We also got to learn about SSMS and its various options to get started with T-SQL querying.

We talked about the fundamentals of relational databases, T-SQL and SQL Server. We also learnt about SSMS which is the development tool for T-SQL. When you'll actually start practicing the T-SQL gradually, this is the tool that you'll use.

This chapter is the very first step to get the understanding of the pre-requisites to start with learning T-SQL. Without the knowledge of these basics, it will be like riding a motor bike without knowing how to drive a bicycle.

In the next chapter, we will deep dive more into tables and learn to create the table.

Points to remember

SQL Server is a relation database.

T-SQL is an extension to SQL and is proprietary language of Microsoft for SQL Server.

SSMS is the development tool also called IDE for developing and managing SQL Server databases. You'll use this tool for practicing the T-SQL

Multiple choice questions

Primary key allows duplicate values?

True

False

Which category UPDATE command belongs to?

DCL

TCL

DML

DDL

Answers

B

C

Questions

Which important commands are part of DDL?

What is the difference between *Ctrl* + *F5* and

What is C of ACID?

What is transaction?

What is entity?

What is the difference between referencing and referenced table?

Key terms

Query: Used interchangeably for T-SQL statements.

DDL: Stands for Data Definition Language.

DDL: Stands for Data Definition Language.

ER: Stands for Entity Relationship.

RDBMS: Stands for Relational Database Management System.

SSMS: Stands for SQL Server Management Studio.

Compile: It is another term of parse + in SSMS.

Run: It is another term of execute in SSMS.

Result window: It is another term of result pane + in SSMS.

Object Explorer: It is the important section within SSMS which you'll use very frequently and all your objects will reflect here.

Normalization: It is the process of removing data redundancy.

Redundant data: Repeated/duplicate data is called redundant data.

CHAPTER 2

Tables

Data in SQL Server and other relational databases are stored in tables in the form of rows and columns. The table is the starting point of any relational database. We can't query data unless we've tables.

In this chapter, we'll learn more about tables to create, modify, and delete them. We'll also learn about various other supporting features used while dealing with tables in SQL Server.

We've learned about the SSMS in [Chapter 1, Getting](#) It is an advanced tool shipped with **Graphical User Interface (GUI)** to create, modify, and delete the tables and other objects in SQL Server. You can do almost everything using GUI that you can do with T-SQL scripts. However, in this book, we'll only use T-SQL.

Structure

In this chapter, we will cover the following topics:

CREATE DATABASE

USE DATABASE

CREATE SCHEMA

Data types

CREATE TABLE

Three-part naming

ALTER TABLE

DROP TABLE

What is NULL?

Primary Key constraint

NOT NULL constraint

UNIQUE constraint

Primary Key versus UNIQUE constraint

DEFAULT constraint

CHECK constraint

Foreign Key constraint

CASCADING in Foreign Key

CREATE DATABASE

Since we are going to learn tables in this chapter. Hence before we create the table, it's mandatory to have a database that will hold the tables and other SQL Server objects.

Quoting here an important learning from [Chapter 1, Getting](#)

A database can have multiple file groups. A file group can have multiple data files. A data file can have multiple schemas. A schema can have multiple tables/objects. A table can have multiple partitions. A partition can have multiple rows and each row can have multiple columns. A page can hold one or more rows.

You can't create the table without the database. The table has to be created within a database. The database is the placeholder for tables and other database objects.

T-SQL is easy to understand language. The commands are simple and similar to giving general instructions.

CREATE DATABASE command is an instruction to SQL Server to create a database.

The syntax for creating a database in SQL Server is as follows:

```
CREATE DATABASE name>
```

With this syntax, the database will be created under the default location that is **DATA** folder of the SQL Server installation folder.

However, you can also specify the data and log file name and path using the following syntax. We've learned about data and log files in [Chapter 1, Getting](#)

```
CREATE DATABASE name>
```

```
ON
```

```
(NAME = file
```

```
FILENAME = file path>
```

```
)
```

```
LOG ON
```

```
(NAME = file name> ,
```

```
FILENAME = file path>
```

```
) ;
```

Example to create a database:

```
CREATE DATABASE Learn_T_SQL
```

```
ON
```

```
(NAME = Learn_T_SQL_data,
```

```
FILENAME = 'C:\Learn T-SQL\Learn_T_SQL_data.mdf'
```

```
)
```

```
LOG ON
```

```
(NAME = Learn_T_SQL_log,
```

```
FILENAME = 'C:\Learn T-SQL\Learn_T_SQL_log.ldf'
```

) ;

When you'll run the preceding example on SSMS, the result will look similar to as shown in the following screenshot:

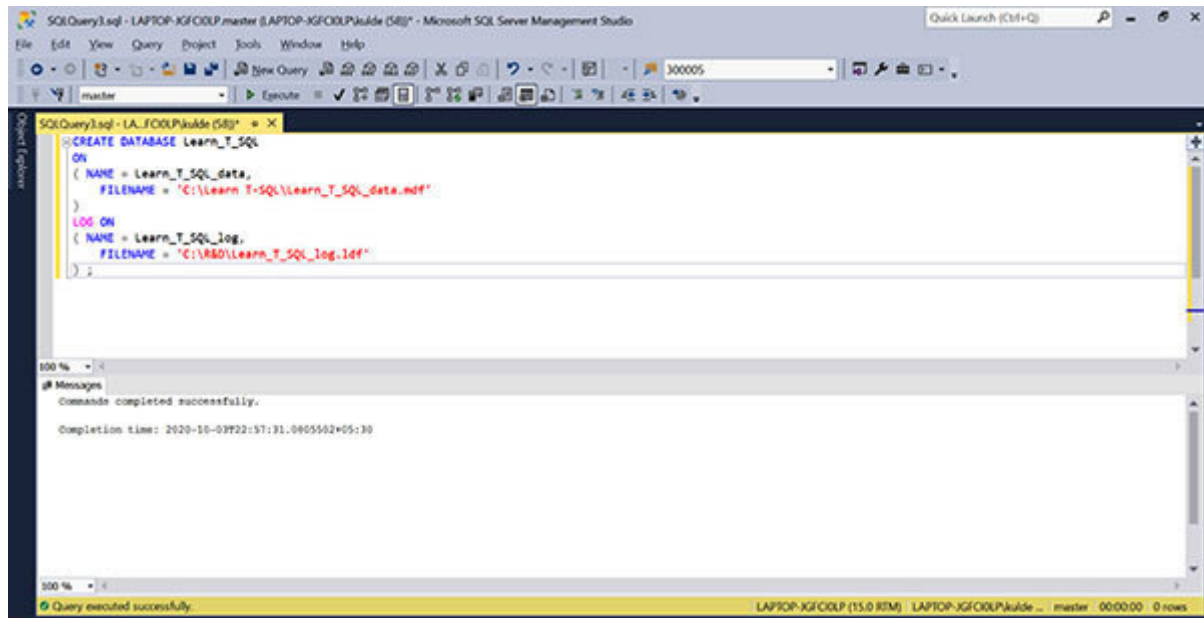


Figure 2.1: Execute Create Database Script

You can see the newly created database under Object Explorer as shown in the following screenshot:

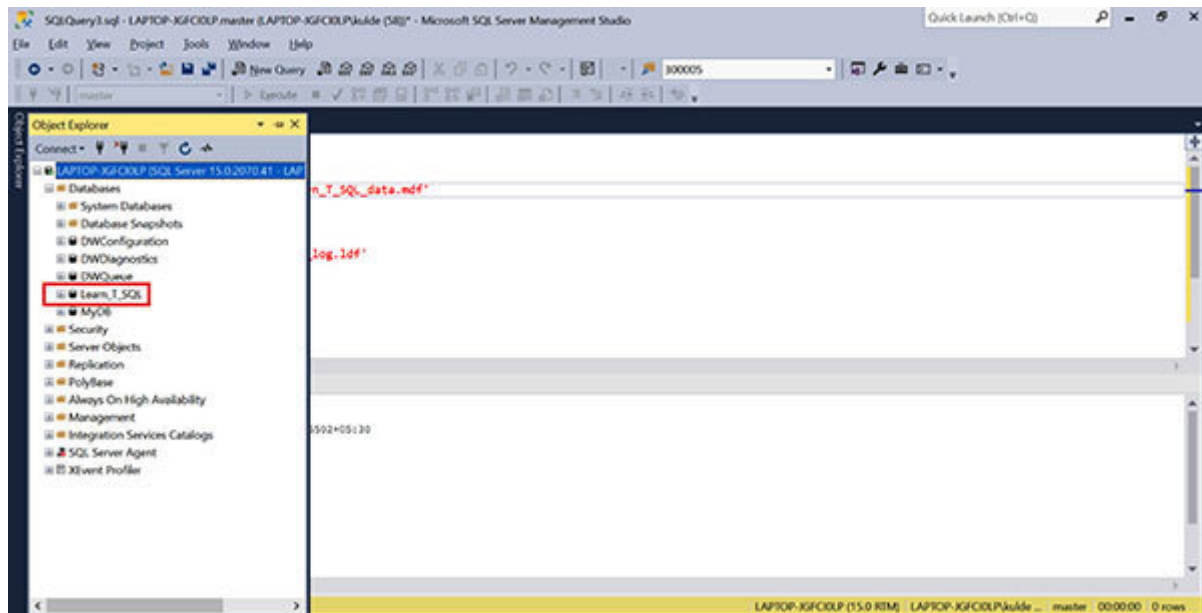


Figure 2.2: Database in Object Explorer

You can see the database files in file explorer as shown in the following screenshot:

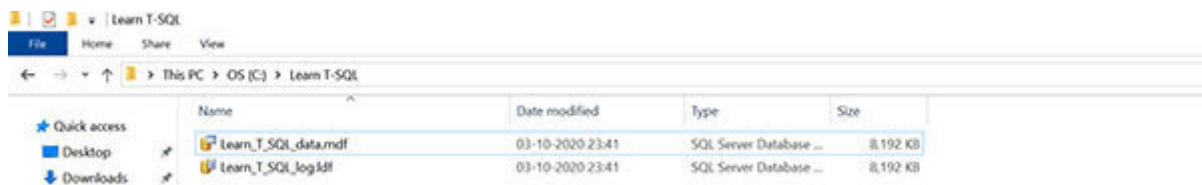


Figure 2.3: Database files in file explorer

USE DATABASE

USE DATABASE instructs SQL Server to point to the specified database. You would be wondering why it's needed at all.

An instance of SQL Server can hold multiple user databases.

As per the maximum capacity specifications for SQL Server, an instance of SQL Server can hold up-to 50 user databases. It can be found at the following URL:

<https://docs.microsoft.com/en-us/sql/sql-server/maximum-capacity-specifications-for-sql-server?view=sql-server-ver15>

We also know that computer works based on the instructions. Computer programs are nothing but a set of instructions.

Now, it's clear that if there are multiple databases then which database to use, has to be specified. **USE DATABASE** commands serve this purpose.

When you open the Query the default database selection is **master** as can be seen in the following screenshot:

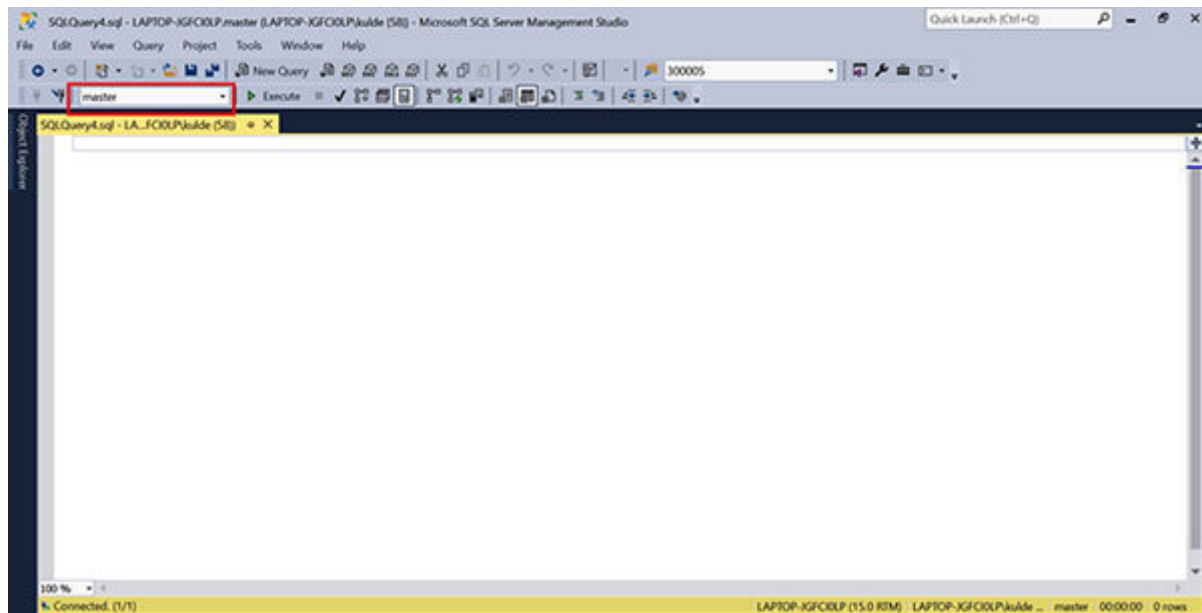


Figure 2.4: Database dropdown

As we already discussed, you will see multiple databases (including one we created upon clicking the drop-down button as can be seen in the following screenshot:

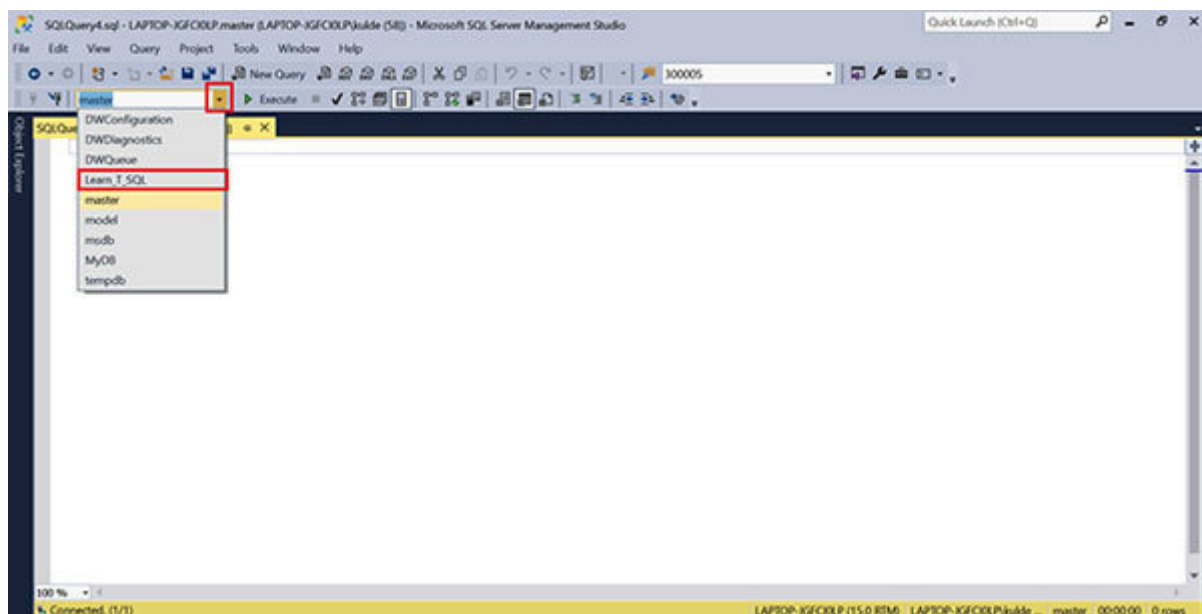


Figure 2.5: Database in the database dropdown

USE DATABASE name>

USE DATABASE Learn_T_SQL

I would like to tell you an interesting thing about SSMS. I'm talking about

Microsoft official documentation on IntelliSense says the following:

The editors in SQL Server Management Studio support Microsoft IntelliSense options that reduce typing, provide quick access to syntax information, or make it easier to view the delimiters of complex expressions.

IntelliSense by default is enabled in SSMS upon installation. However, SSMS offers configurable settings to enable and disable it. You can access this configuration by clicking on the **Tools** -> **Options** menu. A screen similar to [figure 2.6](#) will get populated. It will have various tabs at the left corner of the screen. You need to expand the **Text Editor** tab and then click on the **IntelliSense** option. Upon clicking on the **IntelliSense** option, you would see the relevant settings belonging to IntelliSense as highlighted in the following screenshot. The first setting is **Enable**. If this checkbox is ticked, then it means the IntelliSense is enabled. But if you will untick the checkbox, then it means the IntelliSense is disabled.

You can read more about IntelliSense at the following URL of Microsoft official documentation:

<https://docs.microsoft.com/en-us/sql/ssms/scripting/intellisense-sql-server-management-studio?view=sql-server-ver15>

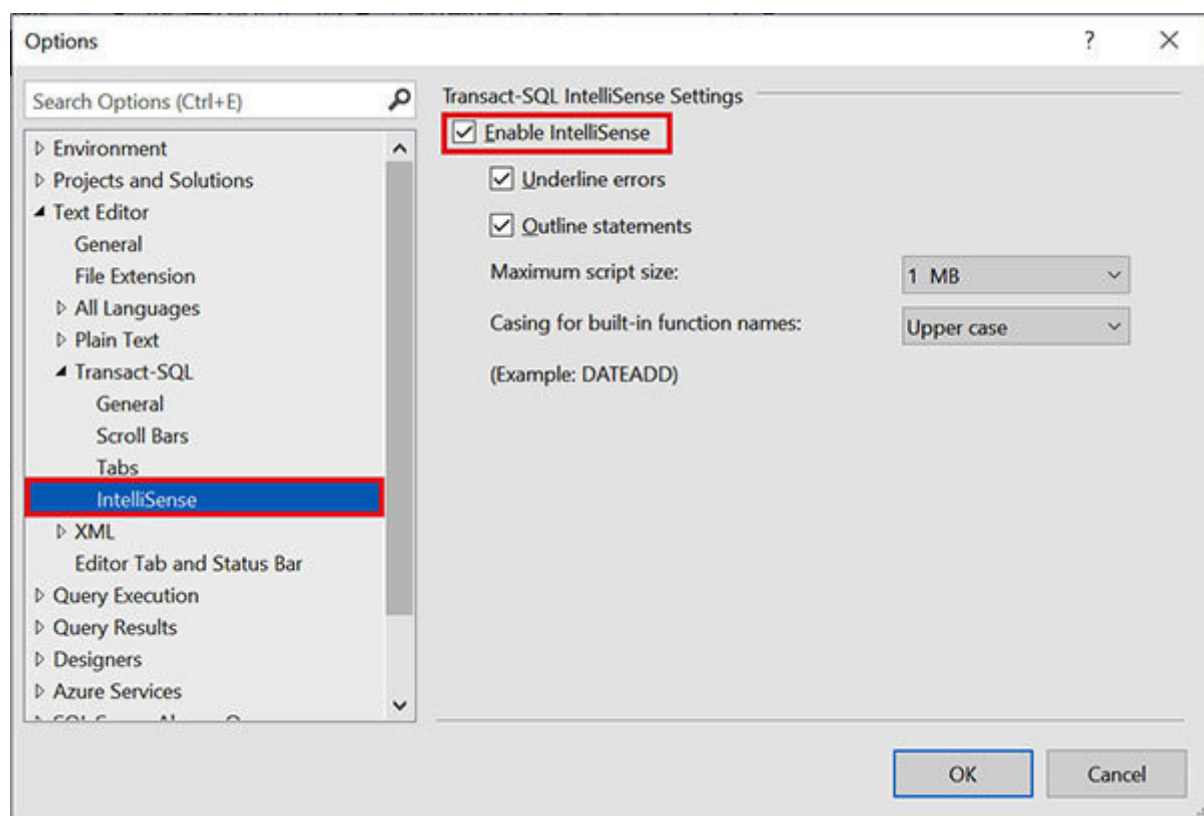


Figure 2.6: Options screen

As it can be seen in the following screenshot, when I typed three letters SSMS gave an auto recommendation of database name:

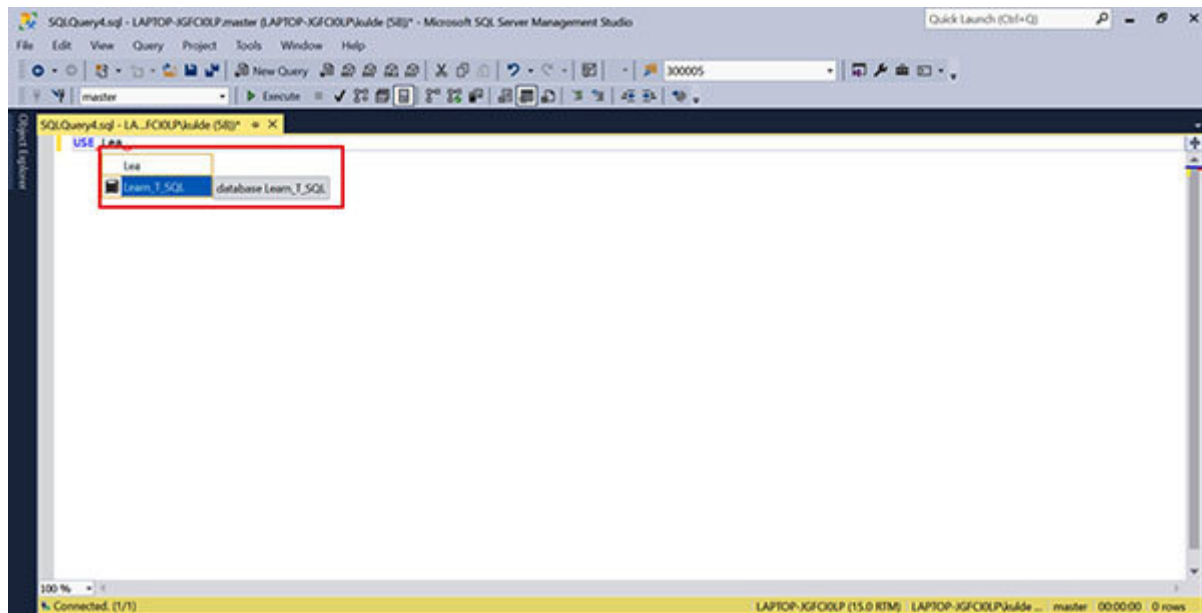


Figure 2.7: IntelliSense in query editor

Now when you will run the command, the database selection will change as shown in the following screenshot:

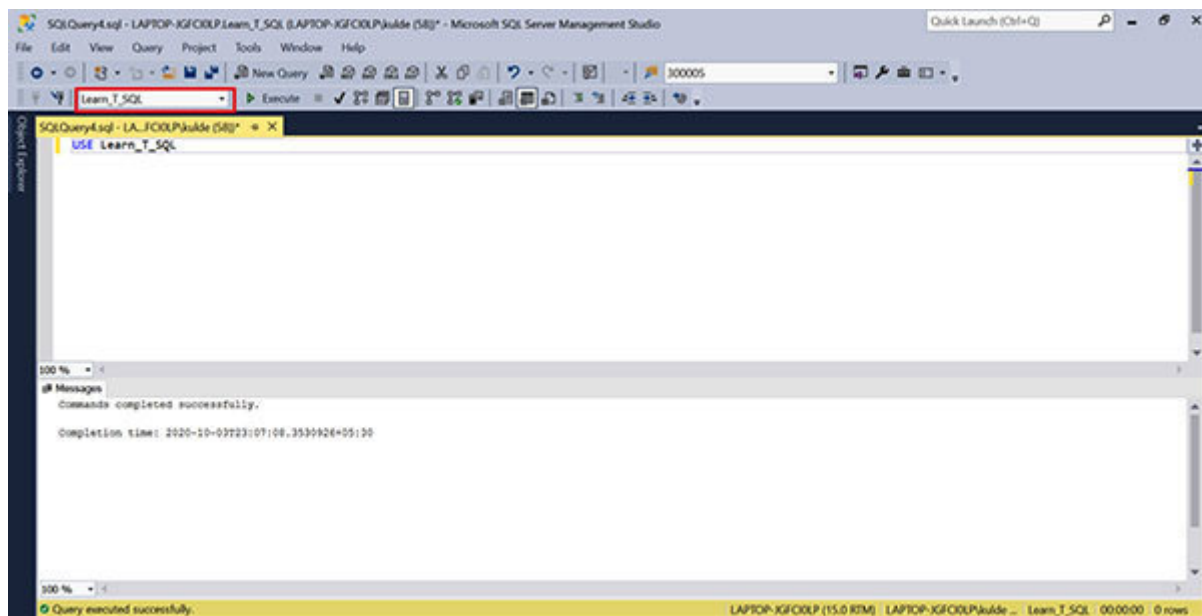


Figure 2.8: Execution of USE command

Congratulations on successfully creating the first database and writing the first query in SQL Server!

Since now we have created our database, we can now further deep dive into T-SQL.

CREATE SCHEMA

We talked about the schema in the previous chapter. Schema is a group of SQL Server objects. A database can have multiple schemas and a schema can have multiple objects. An object cannot belong to multiple schema.

Schema plays a vital role in grouping and organizing objects logically. It makes access control, script generation, code management, and release management easier.

As we discussed in the previous chapter, a database for an organization can have multiple schemas, each for individual departments. Each such schema can hold the objects belonging to the respective department.

Each user database has a default schema. If no schema is assigned while creating the objects then it's created under dbo schema. dbo stands for database owner.

Syntax:

`CREATE SCHEMA name>`

`CREATE SCHEMA Learn_T_SQL`

Data types

The Data type can be simply understood as the type of data. It represents the type of data that the object can hold.

When you create a table then you need to specify the data type of each column of the table. There could not be a column without data type.

Similarly, each local variable, parameter, and expression has a related data type. Whatever objects stores data, be it permanent or temporary, has a data type.

There are various kinds of data types. However, we'll discuss the widely used ones.

Data types are widely divided into multiple categories as covered in the following sections.

String

Each data type requires storage. From a storage point of view, data types are divided into fixed length and variable length data types:

Fixed length data type: Consumes a fixed storage size irrespective of whatever value is assigned. However, a data type does not accept the data beyond the upper storage cap of the data type.

For example, **Integer** is a fixed length numeric data type. It consumes four bytes irrespective of whether the value **1** is assigned or **1000** is assigned. However, there is an upper cap of The moment the value will cross the upper cap, SQL Server will raise an error.

Variable length data type: Consumes dynamic storage size depending upon the value/data. It also consumes an overhead storage size which generally differs for each variable length data type.

For example, **Varchar** is variable length string/alphanumeric data type. We can define the size of **Varchar** data type, such as **Varchar** In this case, it will consume *two bytes + (number of characters * 1 byte)*. Each character consumes 1 byte. 2 bytes are an overhead cost associated with variable length data types.

The overhead cost here is additional storage required by SQL Server to accommodate variable length datatypes.

If we store then it will cost 3 bytes.

For it will cost 4 bytes and so on.

But the moment the character length will exceed the upper cap of 10 characters, it would only accept the character up-to 10 characters and ignore the rest. However, it won't raise an error.

For example, for the value it will only accept **abcdefghij** without raising any error.

The following are the important fixed length string data types:

CHAR This is fixed length ANSI character data type. Consumes the bytes as defined in the length, irrespective of the value assigned. If the value is less than the length, then the space character will be appended at the end of the character.

For example, if the data type is defined as **Char (5)** and the value is assigned as then it will store

NCHAR This is fixed length Unicode character data type. Consumes the double the bytes as defined in the length, irrespective of the value assigned. If the value is less than the

length then the space character will be appended at the end of the character.

For example, if the data type is defined as **Char (5)** and the value is assigned as then it will store It will consume 2 bytes for each character. In this example, it will consume 10 bytes.

If you want to store a character or a text belonging to a language other than English then you will have to use the Unicode character data type. For example, the characters belonging to languages such as Chinese, Arabic, Urdu, Hindi, Japanese, and so on are Unicode characters.

The following are the important variable length string data types:

VARCHAR This is variable length ANSI character data type. Consumes the bytes as the number of characters in the assigned value, plus 2 bytes as overhead. Maximum limit is **2 VARCHAR (MAX)** can be used to accept the maximum value.

NVARCHAR This is variable length Unicode character data type. Consumes the double the bytes as the number of characters in the assigned value, plus 2 bytes as overhead. The maximum limit is **2 NVARCHAR (MAX)** can be used to accept the maximum value.

Binary.

Computers don't understand the natural language such as English, French, and so on. The computer only understands the binary. Binary is the combination of 1's and 0's. We won't discuss much about binary here.

When you'll work on databases, you'll also come across requirements where you'll receive the request to store and retrieve the data in binary format. The following are the binary data types:

BINARY Fixed length binary data type.

VARBINARY Variable length binary data type. Consumes the bytes as length of the data assigned/entered, plus 2 bytes as overhead.

The maximum limit is **2³¹ - 1** **VARBINARY (MAX)** can be used to accept the maximum value.

Numeric (non-decimal values)

As we discussed earlier, data could be of various types. The following are the fixed length numeric data types that are used to store non-decimal values. These data types do not accept decimal values, even if you'll assign them. In case a decimal value is assigned, it will be converted to non-decimal value automatically by SQL Server by ignoring the decimal part.

Each of these data types accepts values in different ranges. If you know the maximum limit of the expected data then choose the relevant data types explained as follows.

For example, if you know you won't get a value that will exceed **100** then choose **TINYINT**. There is no meaning in choosing the bigger data types. It won't make sense to spend 4 bytes of storage to store a value that can be accommodated in 1 byte.

Will you spend 100\$ to buy a cup of coffee which you can get in 20\$? You can if you have excess money!

I was giving a real-world example, please don't take it seriously. It's your money, *you can decide how much to pay for your coffee!*

Always remember, nothing is free in this world. There is some cost associated with everything, in the databases too. The sizable

data type requires more space. More is the storage requirement; more you'll end up spending on storage and memory.

Before you decide on the data types, know the maximum range of expected values. Proper use of the data types optimizes the storage requirements that in turn optimizes the cost and performance. Following are the various integer data types, their space requirements, and range of values:

Accepts values ranging from 0 to 255. Consumes 1 byte.

Accepts values ranging from -32,768 to 32,767. Consumes 2 bytes.

Accepts values ranging from -2,147,483,648 to 2,147,483,647.
Consumes 4 bytes.

Accepts values ranging from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. Consumes 8 bytes.

Numeric (decimal values)

These data types accept decimal values.

Precision mean number of digits in a number (including both left and right of decimal point).

Scale means the number of decimal places (right of decimal point).

For example, **100.25** has precision of **5** and a scale of

Fixed precision and scale numeric data Both of the following data types are exactly the same, but with different names. I've hardly used decimal. Numeric had served all my purposes so far. Default Precision for both of these data types is minimum precision is and maximum can be

These data types are widely used for dealing with decimal values. They offer good accuracy.

Fixed precision and scale means the precision and scale will always be the same (as defined in the data type) in all the cases.

For example, for **decimal (4, 2)** or **numeric (4,** The following will be accepted values:

1, 1.00, 10.99, 99.99

If you will supply then it will become

If you will supply then it will be treated as

If you will supply then it will be treated as

DECIMAL (,)

NUMERIC (,)

The scale must be always less than or equals to precision else SQL Server will throw an error. If the scale equals to precision, then you can only assign decimal value up to the number of decimal places as defined in scale. For example, if Numeric (2, 2) data type is defined then only values up to 0.99 can be assigned. The moment you will assign 1.00, SQL Server will throw an error.

Approximate floating_point numeric data types

Following data types do not offer good accuracy. For example, if you'll run a calculation that uses these data types then the decimal portion of the result may be approximate (less accurate). It's also possible that if you'll run the same calculation multiple times then you may get different results.

You should not use this data type if accuracy is important for you. For example, it should not be used for the price, banking, and financial domain data, and so on which demands accuracy. This data type is majorly used for scientific calculations or approximate values, and so on.

The default precision for **Float** is **15** digits.

You can control precision using the following syntax:

FLOAT (n)

If **n** is between **1** and then precision will be **7** digits.

If **n** is between **25** and then precision will be **15** digits.

If **n** is **0** or beyond SQL Server will not accept it, and will prompt an error.

Date and time

The followings are the data types that deal with data of type date and/or time. Each of these data types has a specific purpose, so choose wisely based on the types of data you expect.

For example, if you know that you'll only get date (without time) then choose **DATE** data type. But if you know that you'll get a date with time, then use amongst **SMALLDATETIME** or depending upon the minimum and maximum range of the data you expect.

Please go through the following explanation of each data type and choose wisely:

Accepts only date.

Accepts only time.

Consumes 4 bytes and accepts **YYYY-MM-DD** Also, the range of this data type is **1900-01-01** through You can't store the data prior to **1900-01-01** and post

Consumes 8 bytes and accepts **YYYY-MM-DD** Where **nnn** can be understood as a nanosecond. It's more precise to be millisecond. The range of this data type is **1753-01-01** through

Boolean

This is very simple to understand data type amongst all. When somebody tells you to give you an answer in either or or then you are expected to answer in either of them or This is called or All of these are different terminologies used to refer the same concept.

Accepts **1** means **True** and **0** means

More on data types

More on data types can be read over the following URL of Microsoft official documentation:

<https://docs.microsoft.com/en-us/sql/t-sql/data-types/data-types-transact-sql?view=sql-server-ver15>

CREATE TABLE

We've already discussed that data is stored in tables in the form of rows and columns. We'll now learn to create the tables using T-SQL.

CREATE TABLE command is used to create tables in SQL Server.

```
CREATE TABLE name>.name>.name>
(
[Col1]    type>
, [Col2]  type>
, [Col3]  type>
, [Col4]  type>
:      :
:      :
, [Coln]  type>
)
```

[] can be used with any object such as table name, column name, data type, stored procedure name, function name, view name, and trigger name. Specifying the name within a square bracket is the best practice to avoid the name clash with the SQL Server reserved keywords. If you will use any of the reserved keywords as object names and you have not covered it within square brackets, then SQL Server will throw an error.

You cannot specify the object name such as table name, column name, and so on having space in between. If you wish to have the object name with spaces in between then you need to enclose the object name within For example, column name Customer Name won't work, but [Customer Name] will work perfectly fine. There can be maximum 1024 columns in a table.

Three-part naming

We talked about the database and schema. We also understood that tables/other objects have to be created in a database. We also understood that each table has to be assigned to a schema. If no schema is specified then the table is created under the default schema

Similarly, to access the tables and other objects, you need to specify an object belong to *which schema and the schema belong to which database?*

If you won't specify the schema, then the default **dbo** schema will be considered. If you won't specify the database then the database chosen in the Query Editor will be considered.

name>.name>.

name> is called **three-part** Three-part naming represents the database, schema, and object name:

The first part is the database name.

The second part is the schema name

The third part is the object name. Table is also an object.

Default name> is current database and default name> is If these options are not specified then SQL Server will treat that the object is to be created under/accessed from the dbo schema within the current database. If the object belongs to the current database, then you do not need to specify the three-part naming. Specifying just the two-part name which includes name>. name> should be good enough. It is always a good practice to use the two-part naming for every object.

The following is an example to create a table using T-SQL:

```
CREATE TABLE [Test]
(
  [ID]      INT
,
)
```

The result after executing the preceding query in SSMS will look similar to as shown in the following screenshot:

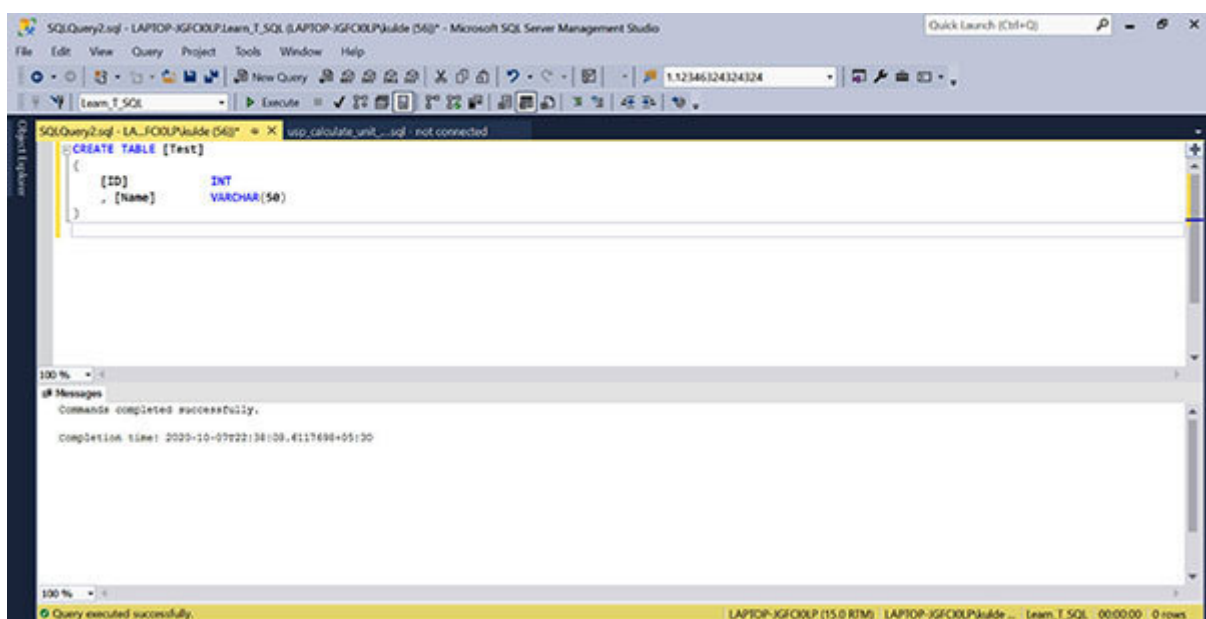


Figure 2.9: Executing CREATE TABLE script

With the help of Object Explorer, we can expand the database to find our newly created table, as can be seen in the following image:

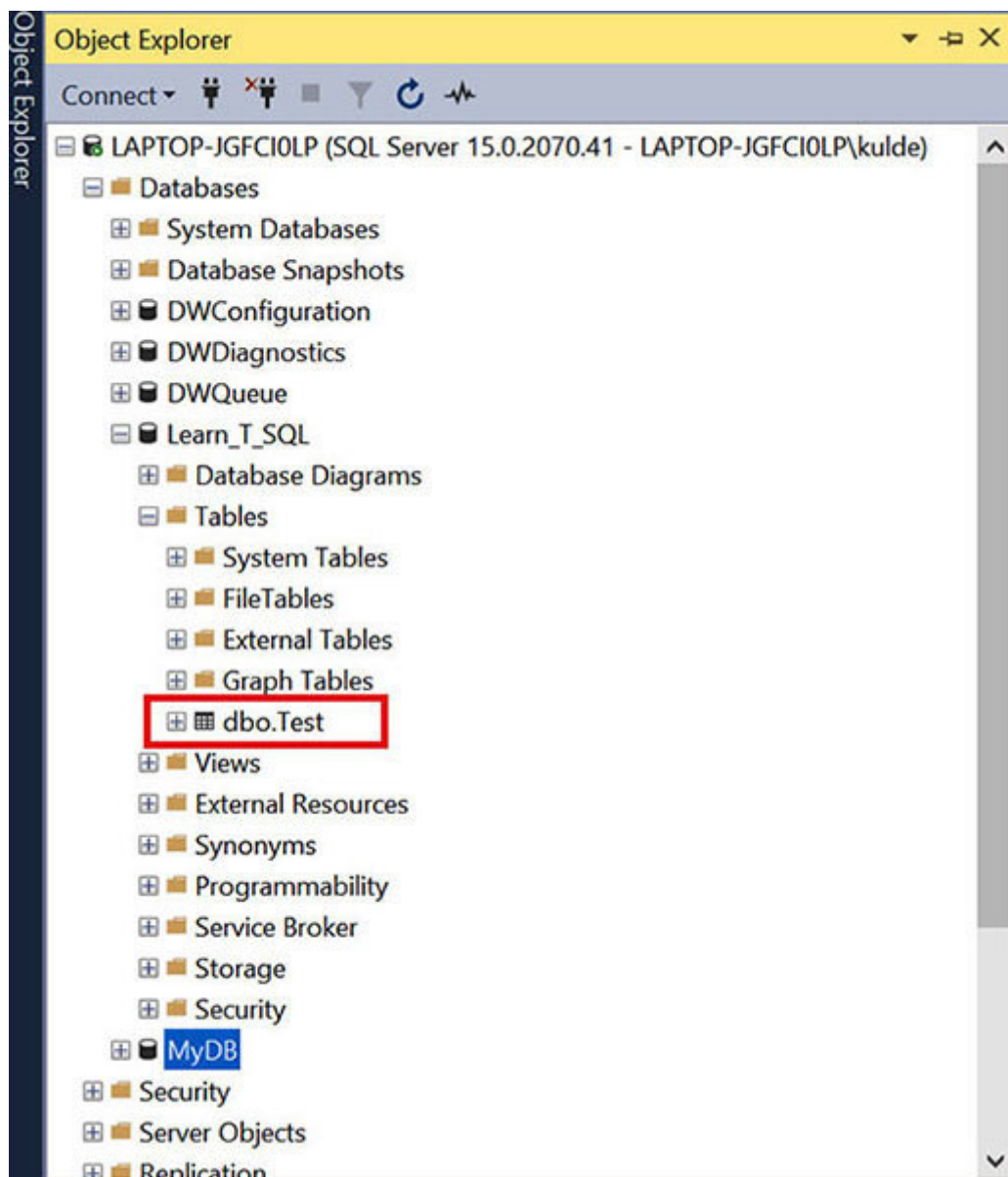


Figure 2.10: Table in Object Explorer

If you notice, the **Test** table was created under the **dbo** schema.

Why?

Because we haven't specified the schema name and that is why it was created under the default schema

ALTER TABLE

ALTER TABLE command is used to modify a table definition. The table definition is also termed as table structure or table schema. We'll use these terminologies interchangeably.

You can use this command to add new columns and constraints, remove/drop existing columns and constraints, change the data type of existing columns, change the length of the data type, change the precision/scale of the data type, and so on.

Syntax to add new

```
ALTER TABLE  
ADD name> type>
```

```
ALTER TABLE [Test]  
ADD [Address] INT
```

Executing the query will look like as shown in the following screenshot:

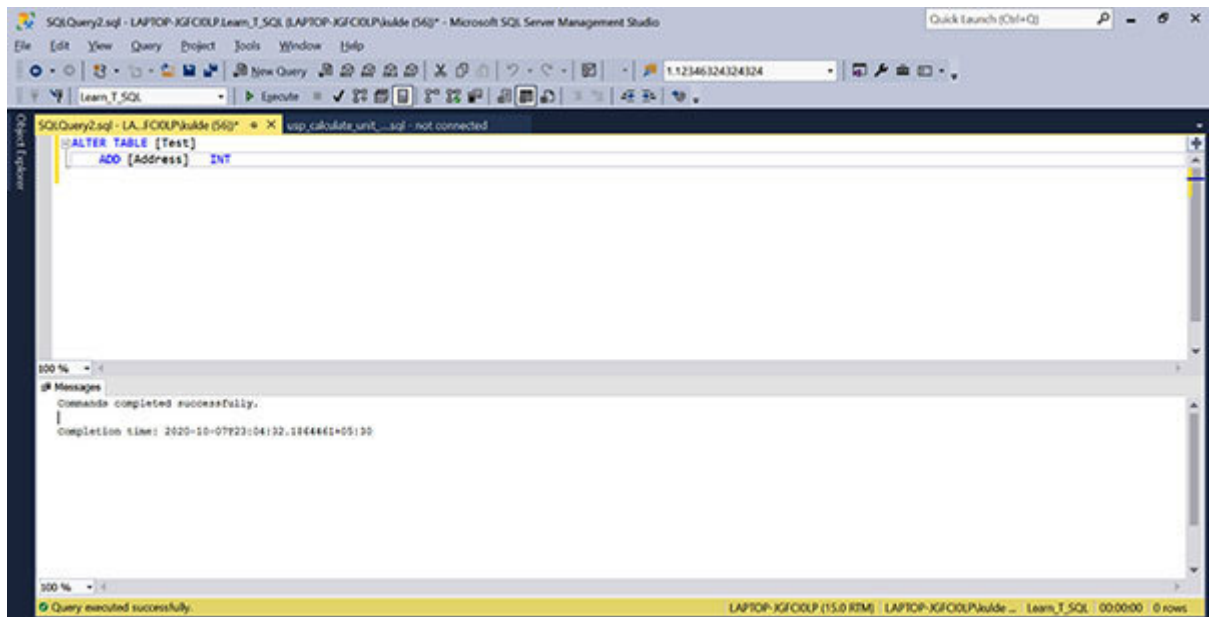


Figure 2.11: Executing ALTER TABLE script

Right click on the database in Object Explorer, and click on the **Refresh** option to see the changes you've made to the table as shown in the following screenshot:

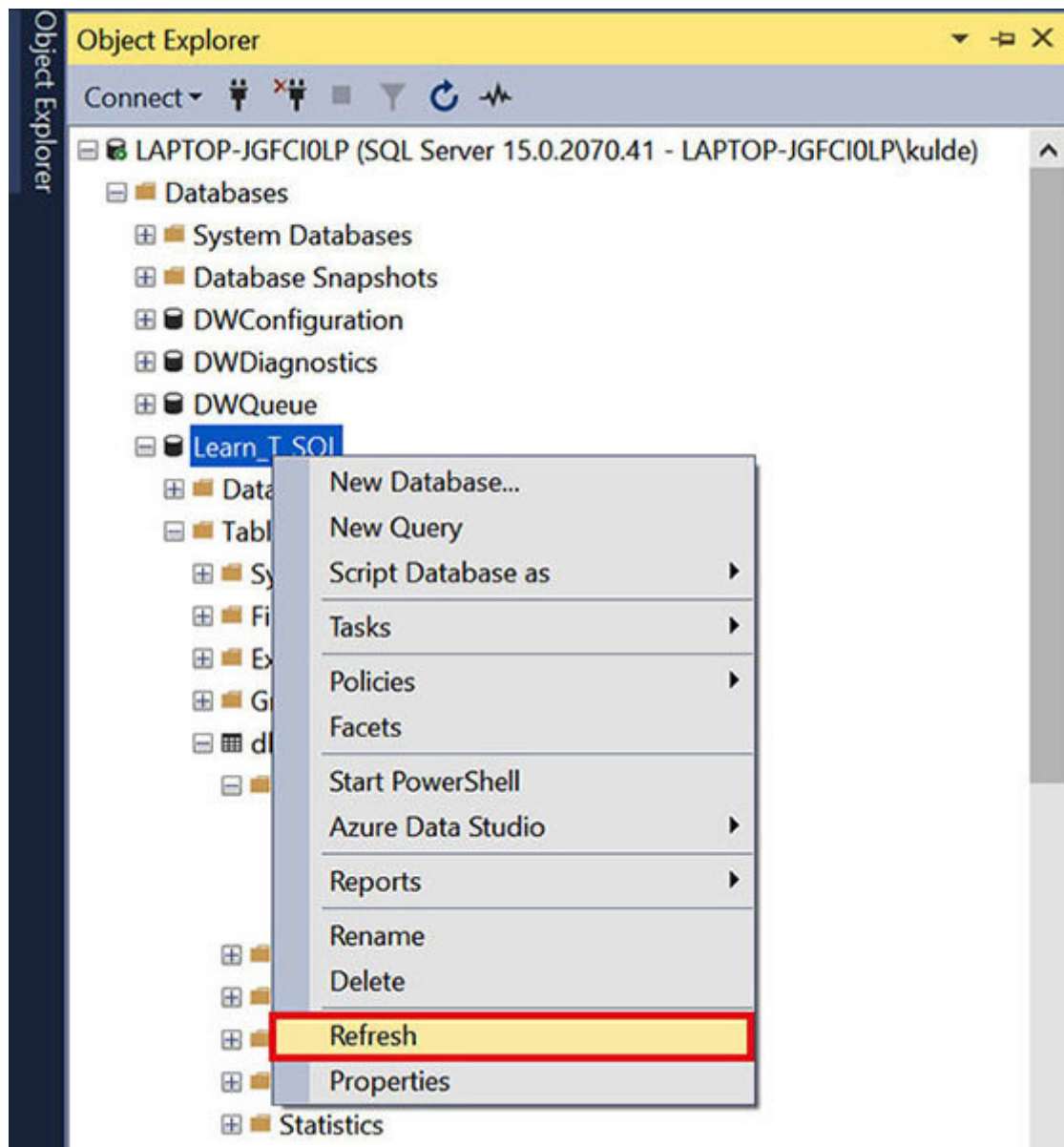


Figure 2.12: Refresh database in Object Explorer

You'll see the newly added column in the Object Explorer as shown in the following screenshot:

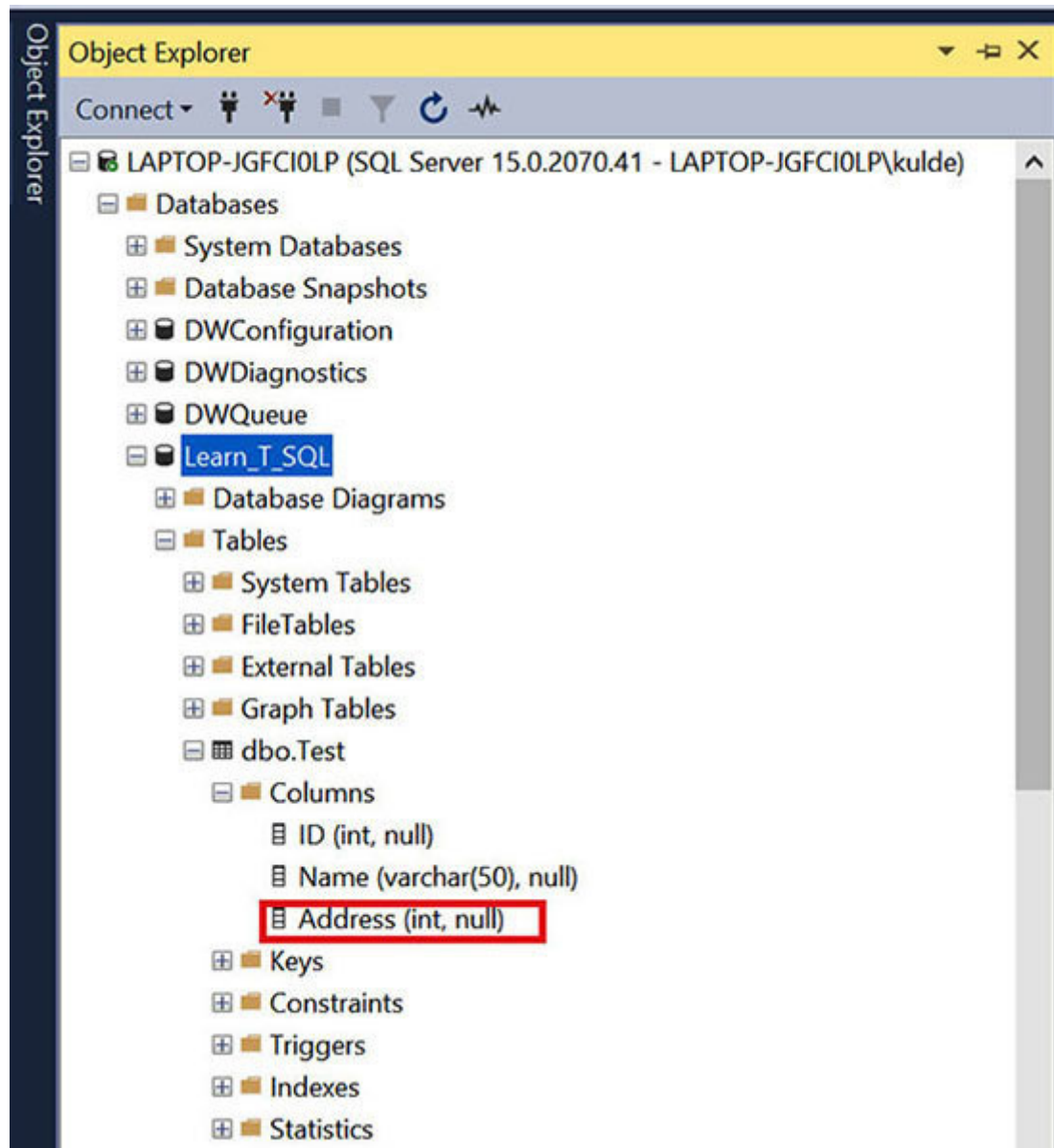


Figure 2.13: Column reflecting in Object Explorer

Syntax to modify existing

ALTER TABLE

name>

ALTER COLUMN name> data type> () / (,)

ALTER TABLE [Test]

ALTER COLUMN [Address] VARCHAR(100)

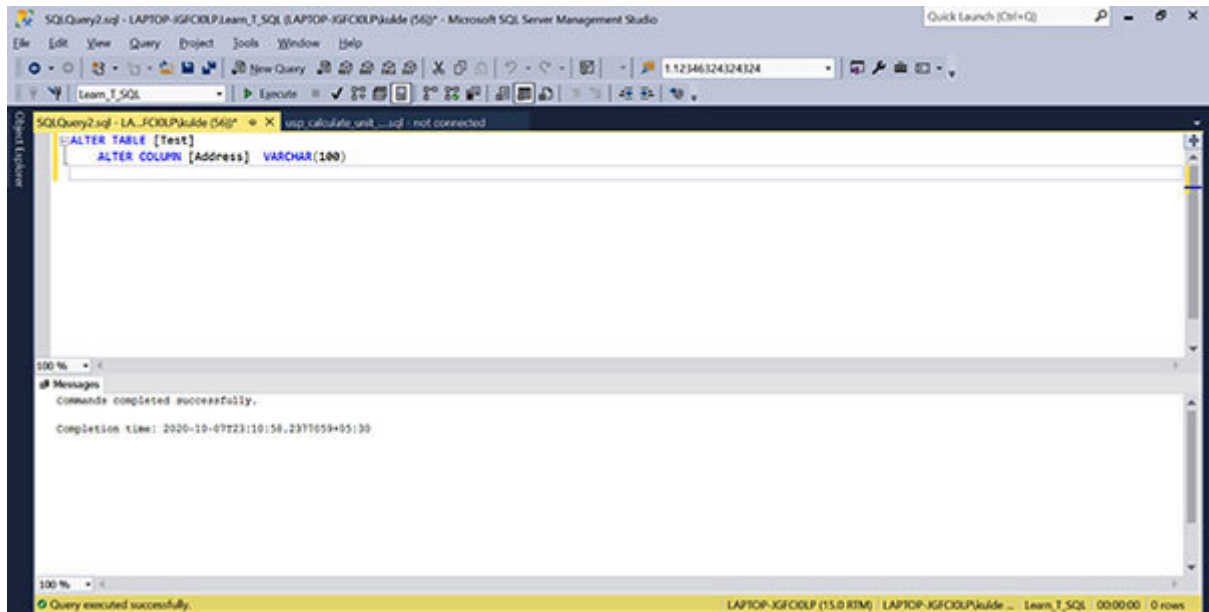


Figure 2.14: Executing ALTER TABLE script to modify the datatype of a column

Refresh the database in the Object Explorer to see the changes you've made to the table. Changes will reflect as shown in the following screenshot:

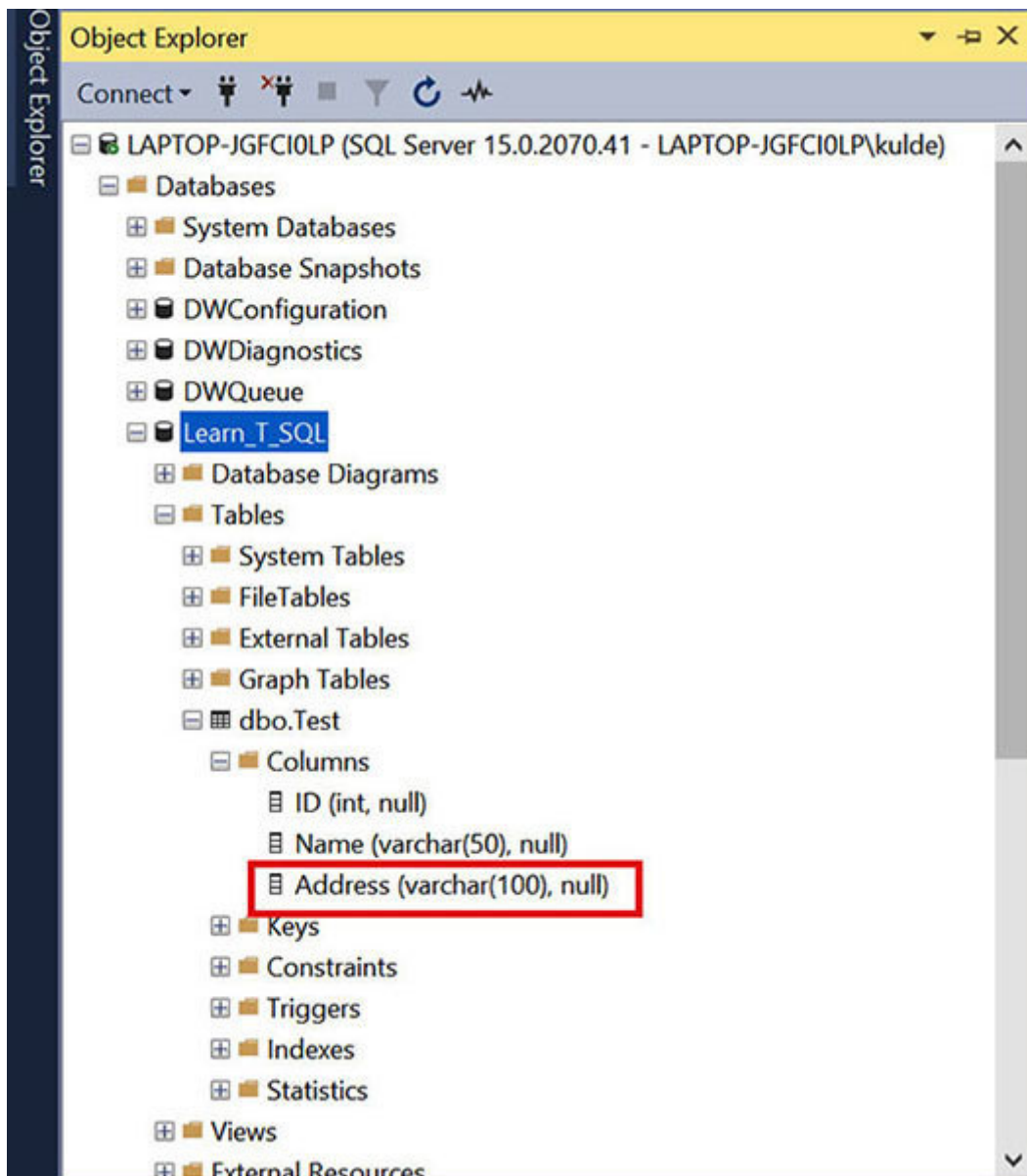


Figure 2.15: Column with new datatype reflecting in Object Explorer

Syntax to drop existing

ALTER TABLE

name>

DROP COLUMN name>

Example

```
ALTER TABLE [Test]  
DROP COLUMN [Address]
```

Executing the query will look like as shown in the following screenshot:

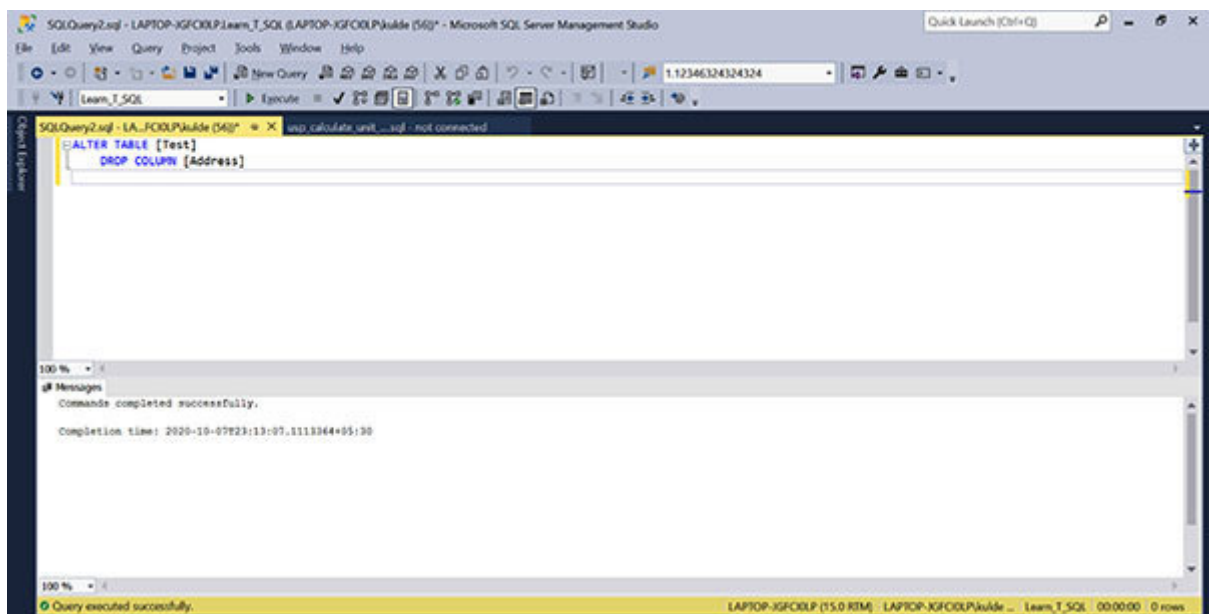


Figure 2.16: Executing *DROP COLUMN* script to remove a column

Refresh the database in Object Explorer to see the changes you've made to the table. Changes will reflect as shown in the following screenshot:

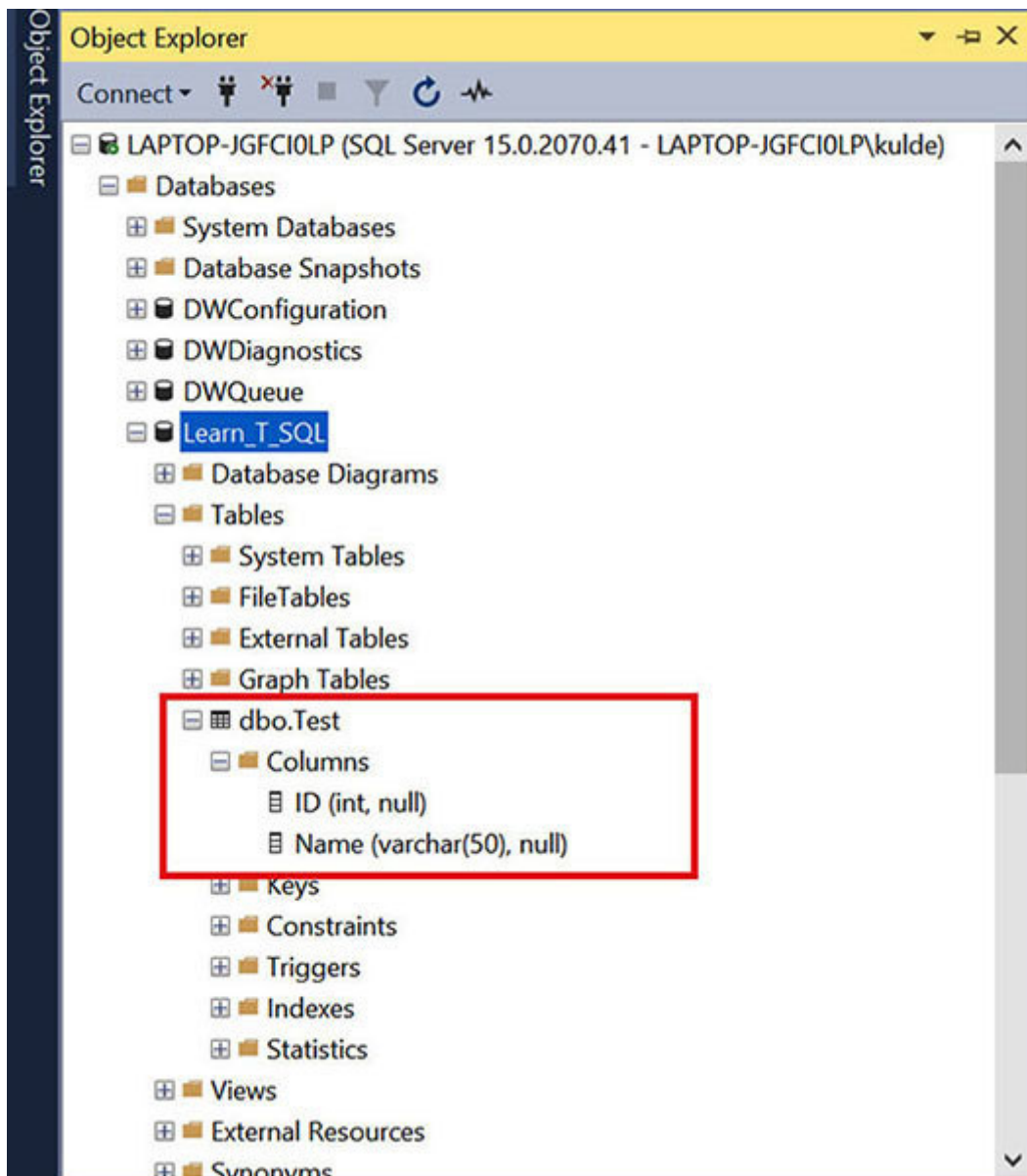


Figure 2.17: Column removed is not reflecting in Object Explorer

You can't change the data type of the column to another data type if the data residing in the column is not compatible with the new data type. For example, if current data type is and column holds alphanumeric data then you can't change it to int or any other numeric data types. If you want to do so then you'll have to remove the data from that column and then only you can do it. Similarly, if the column contains

duplicate data, then you can't add the Primary Key constraint. We'll talk about Primary Key constraint in the subsequent topic.

DROP TABLE

DROP TABLE command is used to drop/delete a table permanently. It deletes table definition along with the data.

DROP TABLE

DROP TABLE [Test]

Executing the query will look like as shown in the following screenshot:

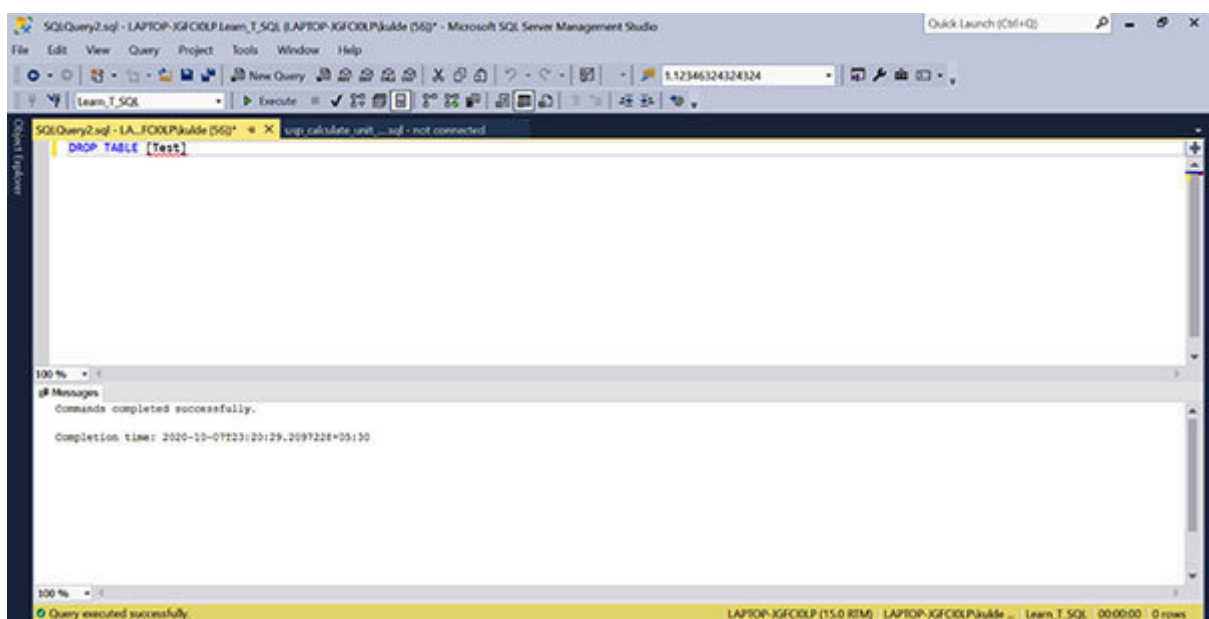


Figure 2.18: Executing DROP TABLE script

Refresh the database in Object Explorer to see the changes you've made to the table.

Test table is removed from the database. Changes will reflect as shown in the following screenshot:

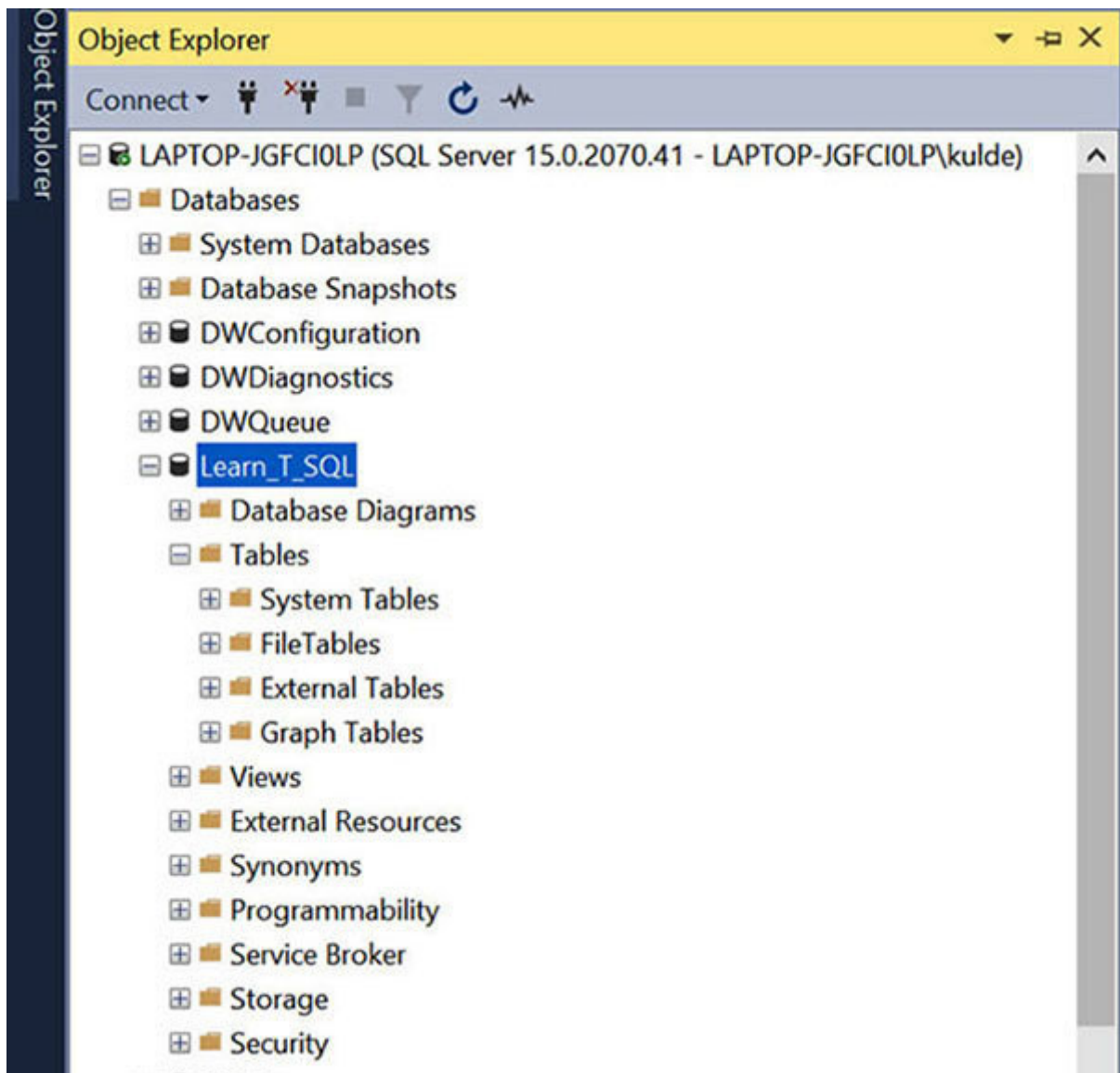


Figure 2.19: Table is not reflecting in Object Explorer

If the table has Primary Key which is referenced as Foreign Key in another table, then you cannot drop the table unless the corresponding Foreign Key data from another table is deleted/removed.

What is NULL?

NULL in SQL Server means If you don't know the value then you can specify NULL. If there is **NUMERIC** data type, then you can only assign numeric values. For **STRING** data type you can assign alphanumeric values. For the data type, you can only assign date/time values. For the **BOOLEAN** data type, you can only assign

NULL can be assigned to any data type and will have the same meaning everywhere that is It is like water. Water becomes the color/taste of the material mixed with it. Similarly, NULL does not have a specific data type. It becomes part of the data type to which it's assigned and has the same meaning all across that is

It is to be noted that NULL cannot be treated similar to the empty string (") of string data types, and 0 of numeric data types.

Primary Key constraint

In [Chapter 1, Getting](#) we talked about constraint and Primary Key in a general sense. We'll now understand more about Primary Key and learn to apply it.

Primary Key is a type of constraint which makes each row unique. A column on which Primary Key constraint is applied can't have duplicate values.

Primary Key does not accept the NULL value. That means you cannot assign NULL value to the column(s) forming Primary Key.

There are three different methods as follows to include Primary Key when creating a new table, either is OK to use. However, *Method 1* is not recommended since it doesn't include the name of the Primary Key.

If you do not specify the name of constraint, it is left to the SQL Server to decide the name. It is not the right choice. If you will run the same query (as mentioned in *Method* on different environments (different databases), the name of Primary Key constraint created might be different.

You may not want the same object having different names amongst various instances. Automated database comparison won't work in this case and lead to manual efforts for comparing objects.

Method 3 is the recommended and ideal approach. Because if you want to create a composite Primary Key, you can't create it using *Method 1* and *Method 3* is the only way to create both single column and composite Primary Key.

Composite Primary Key is a kind of Primary Key created on multiple columns. In order to create a composite Primary Key, add the columns in comma separated format within the bracket.

Method 1

In this method, you don't need to provide a name to the Primary Key. SQL Server will decide the name and automatically assign to it:

```
CREATE TABLE [Test]
(
[ID] INT PRIMARY KEY
,
)
```

Executing the query will look like as shown in the following screenshot:

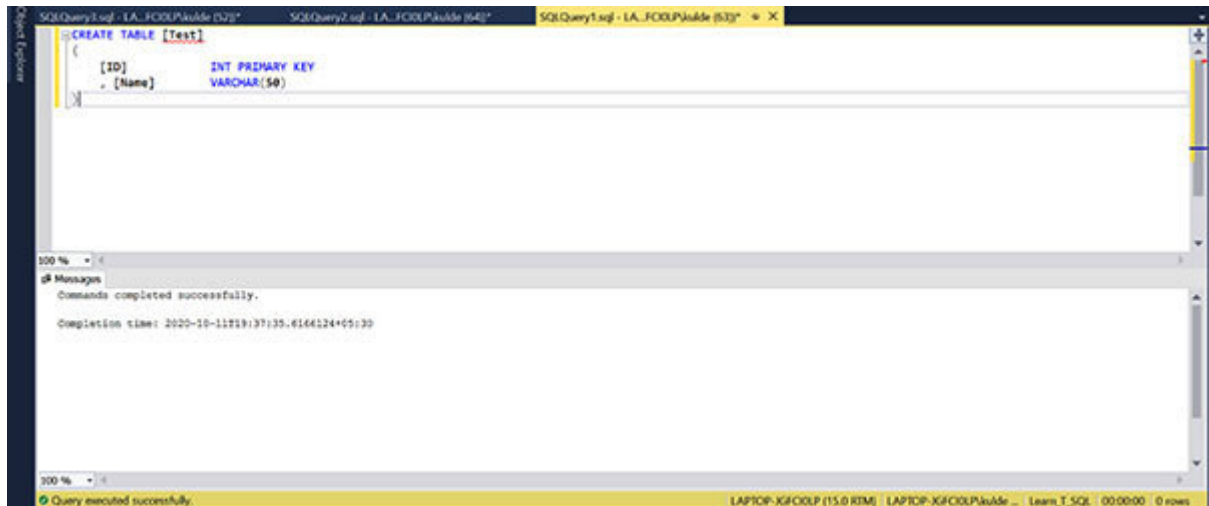


Figure 2.20: Executing script of CREATE TABLE with PRIMARY KEY but without specifying primary key name

The Primary Key is created along with the table as shown in the following screenshot. It is also to be noted that the name of the Primary Key was automatically assigned by SQL Server. When you'll run the same query as mentioned in *Method 1* on your own database, the name of Primary Key may be different:

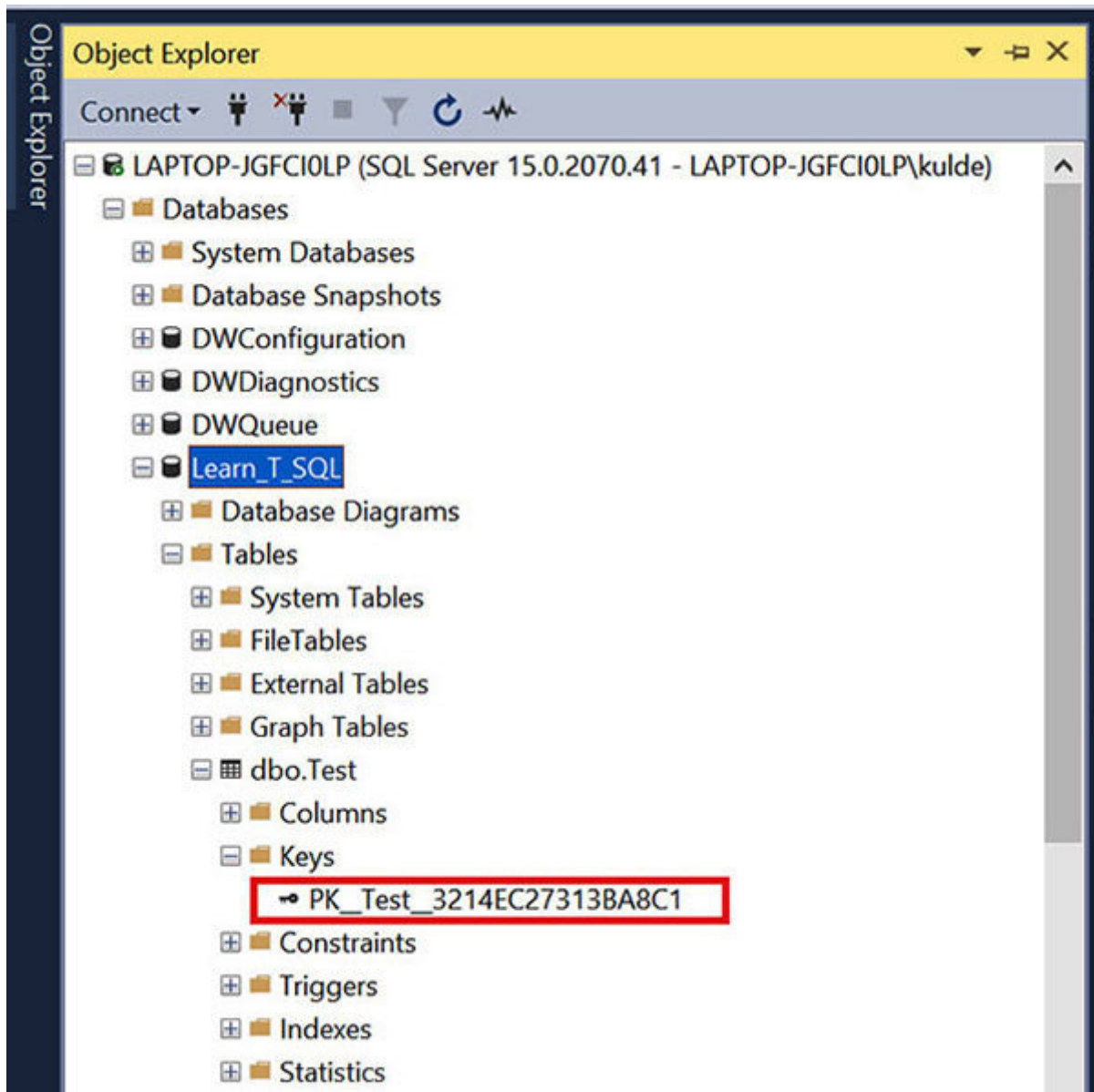


Figure 2.21: Primary Key with name assigned automatically is reflecting in the Object Explorer

Method 2

In this method, you assign the Primary Key constraint along with the Primary Key name in the column definition itself. As you can see in the following example, the Primary Key constraint is assigned immediately after the column data type.

This method offers control on the name to be assigned to the constraint. However, you can't create a composite Primary Key using this method:

```
CREATE TABLE [Test]

(
[ID] INT
CONSTRAINT PK_Test_ID PRIMARY KEY
,
)
```

Since we already have created a table hence in order to create the table with the same name again, we need to drop the existing table first. That is why in the following screenshot you can see a **DROP TABLE** statement followed by **GO** command.

GO command is used to separate two T-SQL statements. It's must if you are running multiple DDL statements together in the same session. Each DDL statement should be followed by the **GO** command.

Executing the query will look like as shown in the following screenshot:

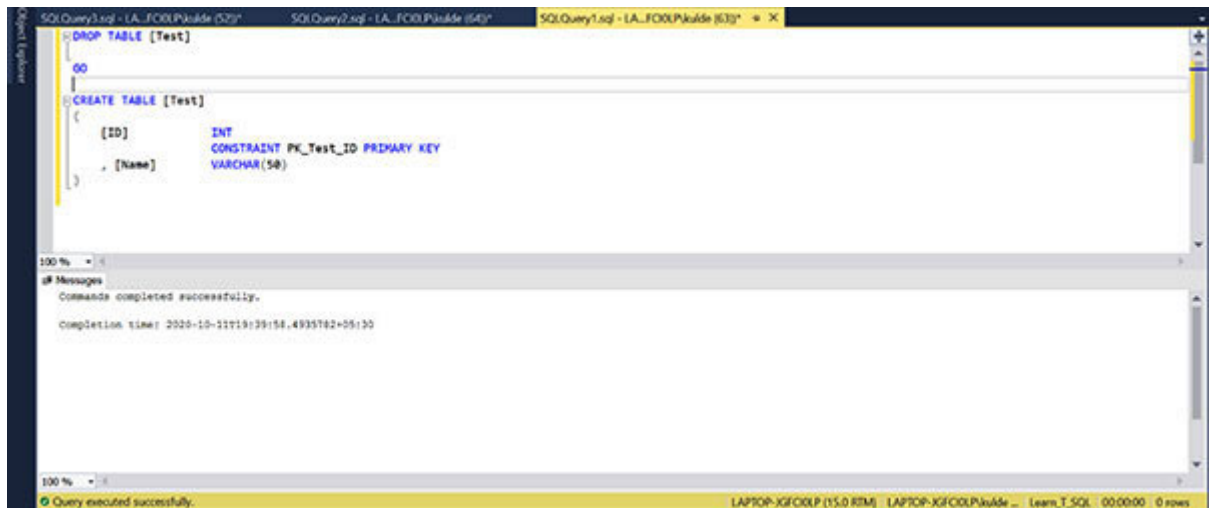


Figure 2.22: Executing script of CREATE TABLE with PRIMARY KEY and with a specific primary key name

As you can see in the following screenshot, the table is created along with Primary Key. It is also to be noted that the name of the Primary Key is same as we assigned. If you'll run the same query on your own database, the name of Primary Key will be same:

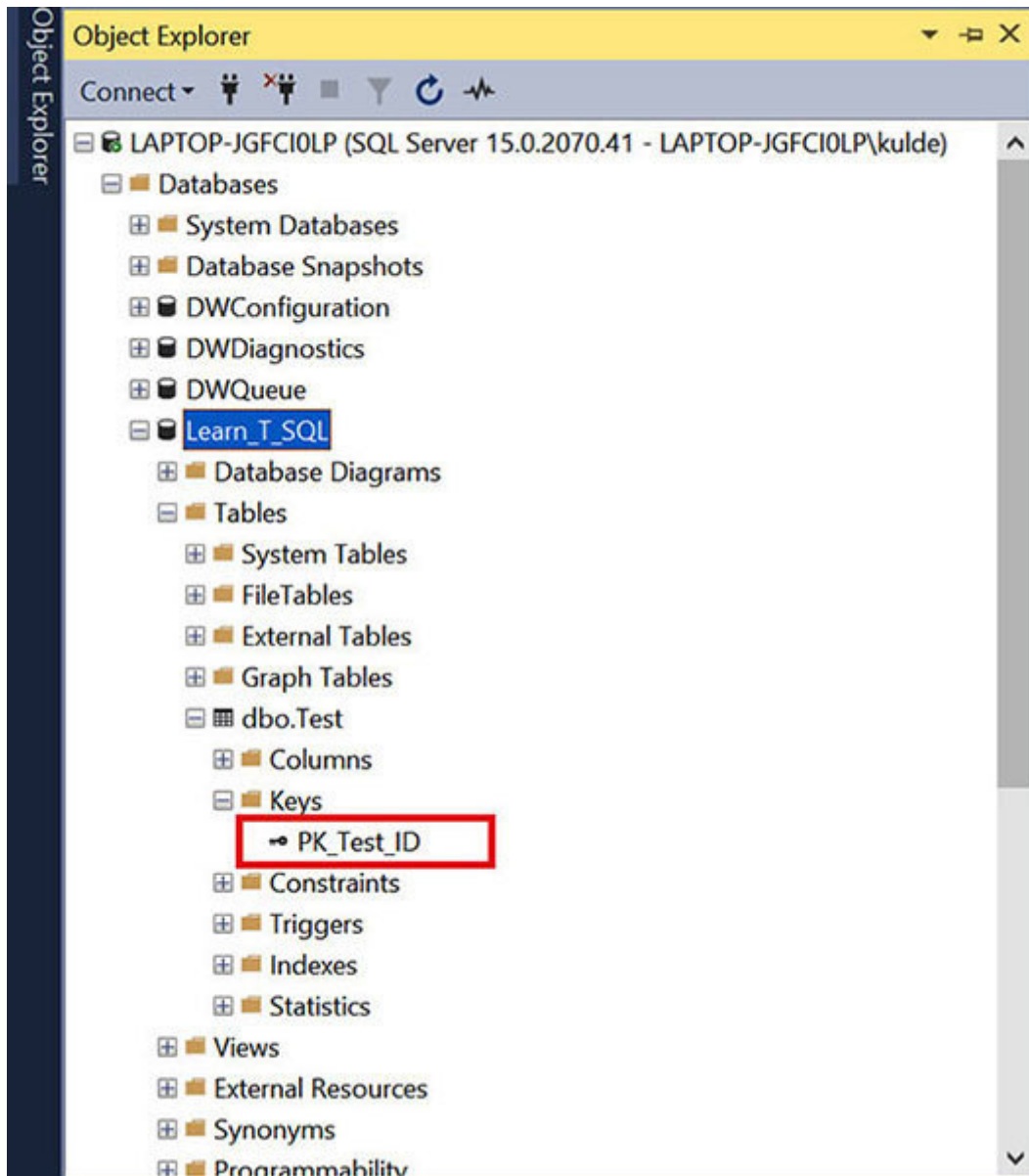


Figure 2.23: Primary Key with specified name is reflecting in the Object Explorer

Method 3

In this method, you define the Primary Key constraint along with the Primary Key name separately after column definition. Using this method, you can also create a composite Primary

Key. Just add the required set of columns in comma separated format within parenthesis after keyword **PRIMARY**

```
CREATE TABLE [Test]
(
[ID]    INT
,
, CONSTRAINT PK_Test_ID PRIMARY KEY ([ID])
)
```

Executing the query will look like as shown in the following screenshot:

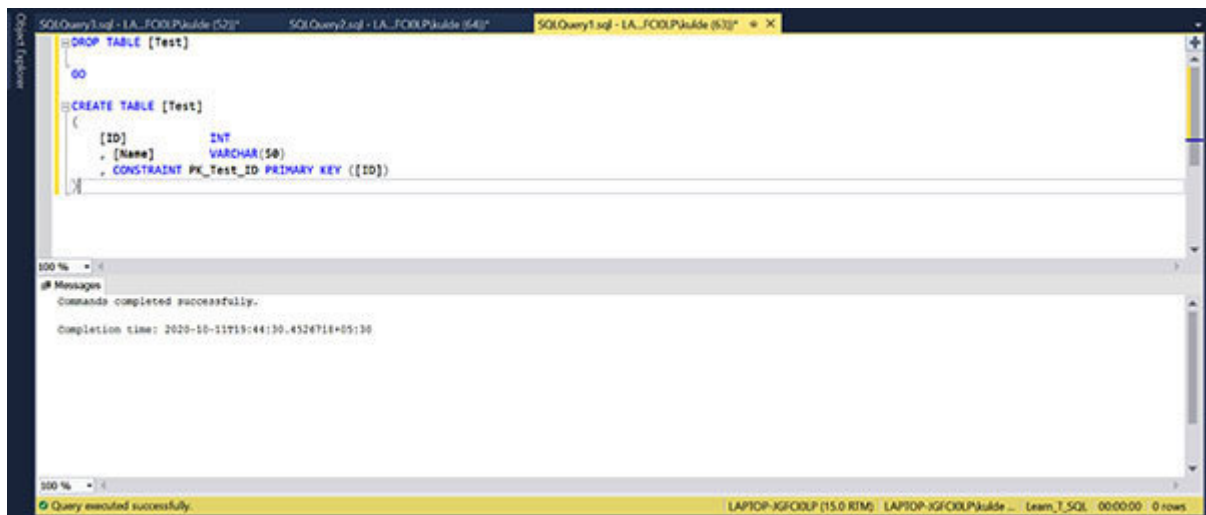


Figure 2.24: Executing script of CREATE TABLE with PRIMARY KEY with key name and column specified

As you can see in the following screenshot, the table is created along with the Primary Key. It is also to be noted that the name of the Primary Key is the same as we assigned. If you'll run the same query on your own database, the name of the Primary Key will be the same:

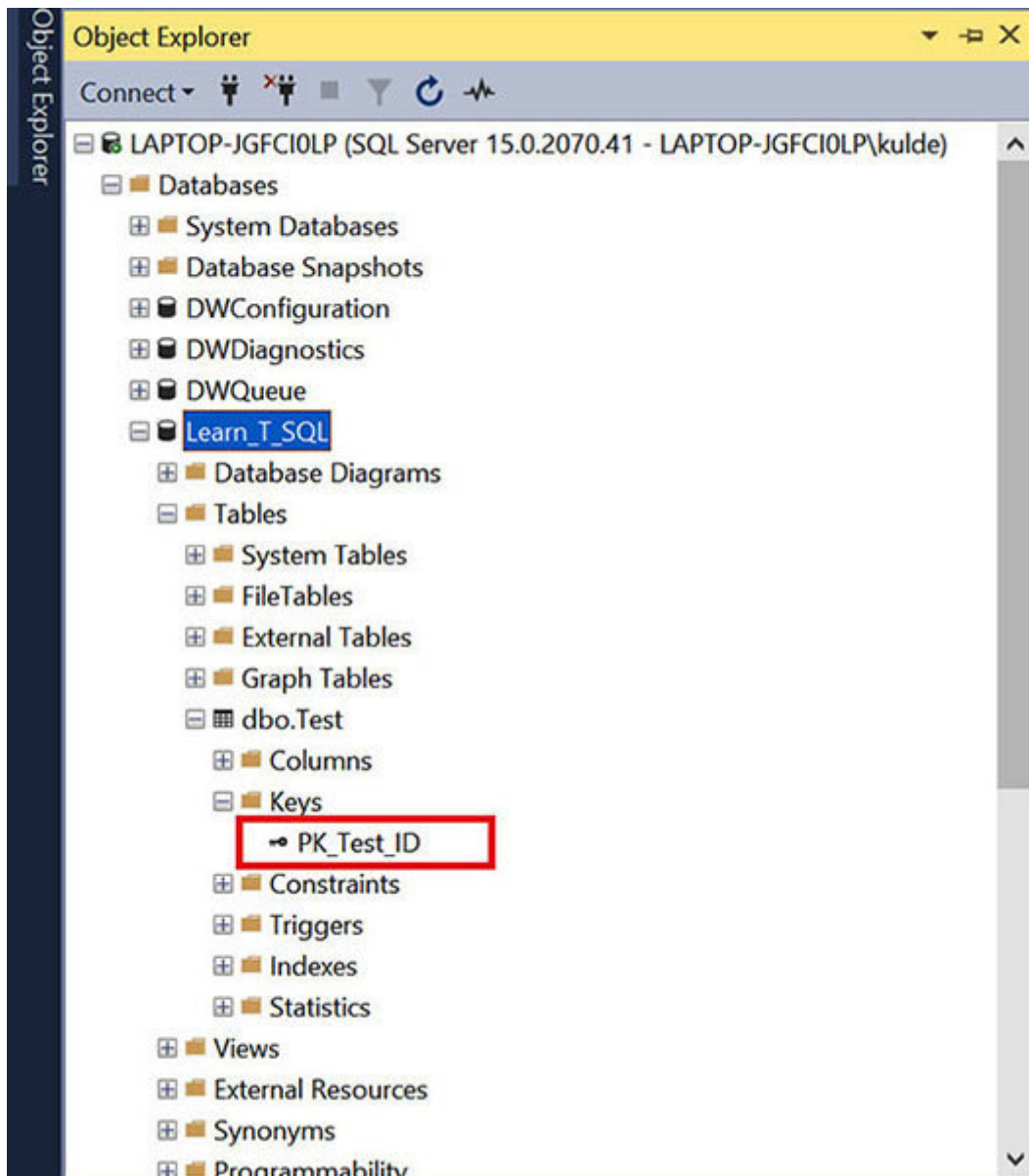


Figure 2.25: Primary Key with specified name is reflecting in the Object Explorer

PK_Test_ID in both of the aforesaid methods is the name of the Primary Key that is created on the ID column. PK is the alias for Primary Key, Test is the name of the table, and ID is the name of the column of which the Primary Key is

created. This is general naming convention for Primary Key. However, this is not the compulsion. You can name constraints your way. But this method helps identify the details of the constraint with the name.

Let us see how to add Primary Key Constraint on the existing table:

```
ALTER TABLE [Test]  
ADD CONSTRAINT PK_Test_ID PRIMARY KEY ([ID])
```

Since we have already created Primary Key **PK_Test_ID** on **ID** column of **Test** table, hence in order to create the Primary Key on the same table again with the same or different name, we need to drop the existing Primary Key constraint first. That is why in the following screenshot you can see **ALTER TABLE ... DROP CONSTRAINT** statement followed by the **GO** command.

A table cannot have more than one Primary Key.

Executing the query will look like as shown in the following screenshot:

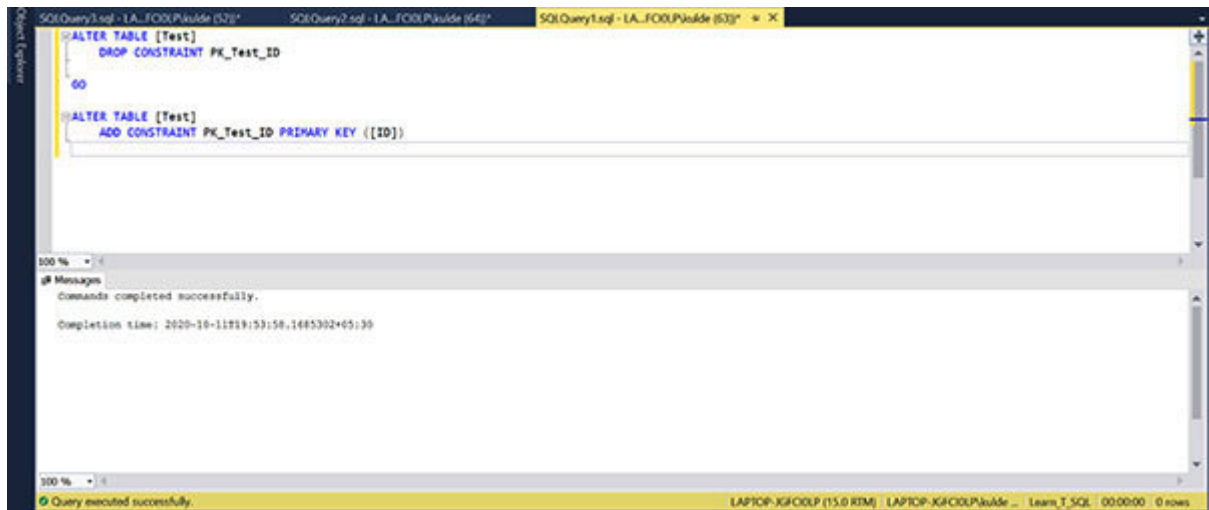


Figure 2.26: Executing ALTER TABLE script to add primary key

You'll find the Primary Key created under the **Keys** folder of the respective table folder as shown in the following screenshot:

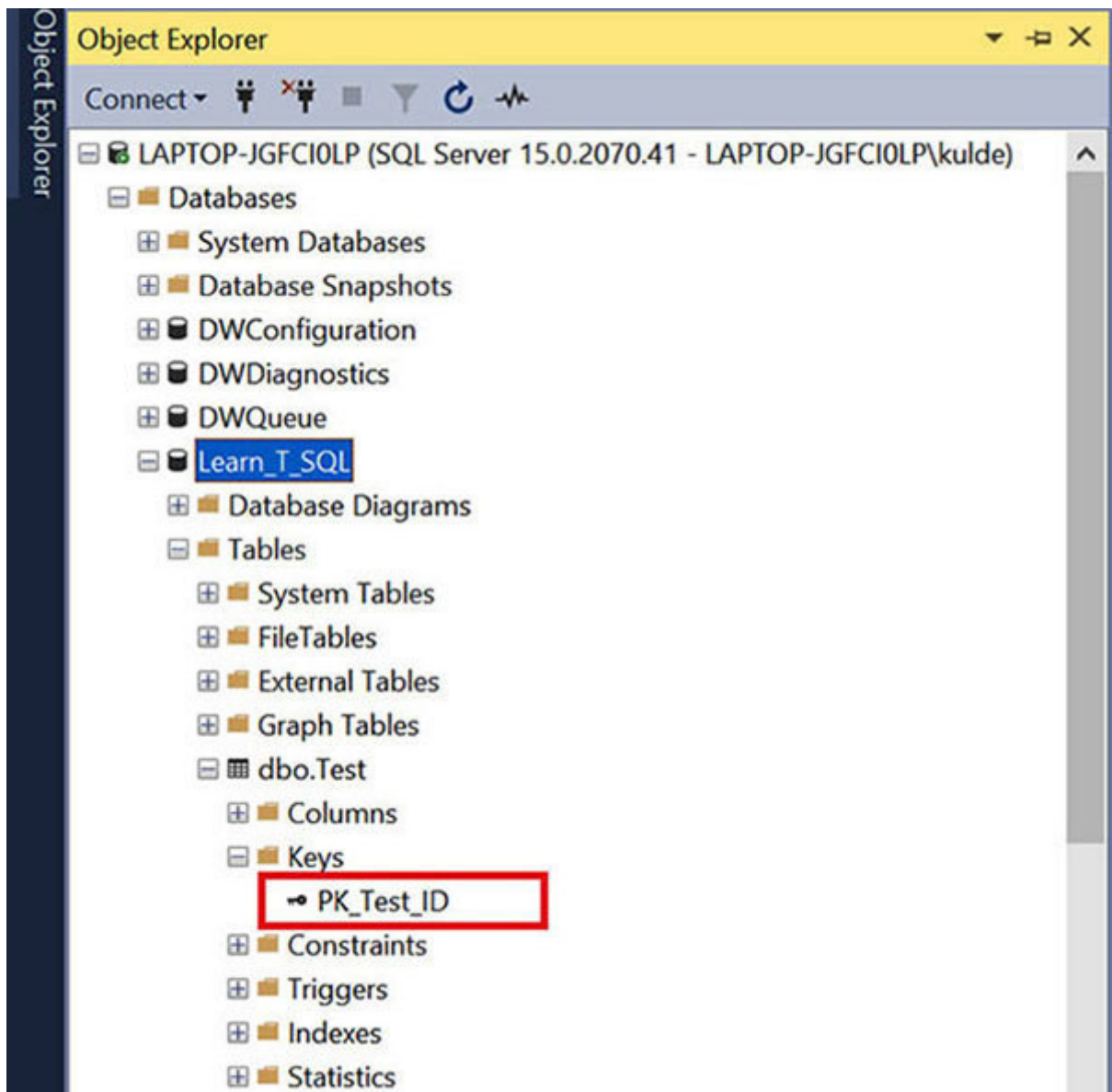


Figure 2.27: Primary Key created with ALTER TABLE script is reflecting in the Object Explorer

NOT NULL constraint

Not Null constraint can be created in the same way the Primary Key constraint is created. This constraint restricts the NULL values. It means you cannot specify the NULL value in a column (using INSERT or UPDATE statement) if it is assigned a NOT NULL constraint. In case a NULL value is assigned, the SQL Server will throw an error.

In the following example, we've assigned the **NOT NULL** constraint to the **Name** column. So, the **Name** column will not accept NULL values. If you do so, SQL Server will throw an error.

Alternative for NULL, in this case, would be " (opening and closing single quotes). It means nothing for String data types.

The following is an example to assign the NOT NULL constraint to a column while creating a new table.

```
CREATE TABLE [Test]
(
  [ID] INT
  , NOT NULL
  , CONSTRAINT PK_Test_ID PRIMARY KEY ([ID])
)
```

The following is an example to assign the NOT NULL constraint to an existing column of a table.

```
ALTER TABLE [Test]  
ALTER COLUMN NOT NULL
```

UNIQUE constraint

UNIQUE constraint is another important constraint. It helps to maintain the data integrity. UNIQUE constraint is used to enforce the uniqueness of the data on a column, or on the set of columns.

This is how you can add **UNIQUE** constraints while creating a new table:

```
CREATE TABLE [Test]
(
  [ID] INT
,
  , CONSTRAINT UQ_Test_ID UNIQUE ([ID])
)
```

This is how you can add **UNIQUE** constraint on an existing table:

```
ALTER TABLE [Test]
ADD CONSTRAINT UQ_Test_ID UNIQUE ([ID])
```

Primary Key and Unique constraints both can't be created on the same set of column(s) of a table. The UNIQUE

constraints can also be made composite similar to the Primary Key.

Since we have already created Primary Key **PK_Test_ID** on the **ID** column of the **Test** table, hence we need to drop the existing Primary Key constraint first before creating the **UNIQUE** constraint on the same column That is why in the following screenshot, you can see **ALTER TABLE ... DROP CONSTRAINT** statement followed by the **GO** command.

You can also see the **ALTER TABLE ... ALTER COLUMN** command to make the **ID** column nullable. This is important to bring the **ID** column to the original state (before the Primary Key constraint was created).

When you make column(s) Primary Key then NOT NULL constraint automatically gets created on those set of column(s), irrespective you specify it or not. Whereas, UNIQUE constraint doesn't create the NOT NULL constraint, and it accepts the NULL value.

Executing the query will look like as shown in the following screenshot:

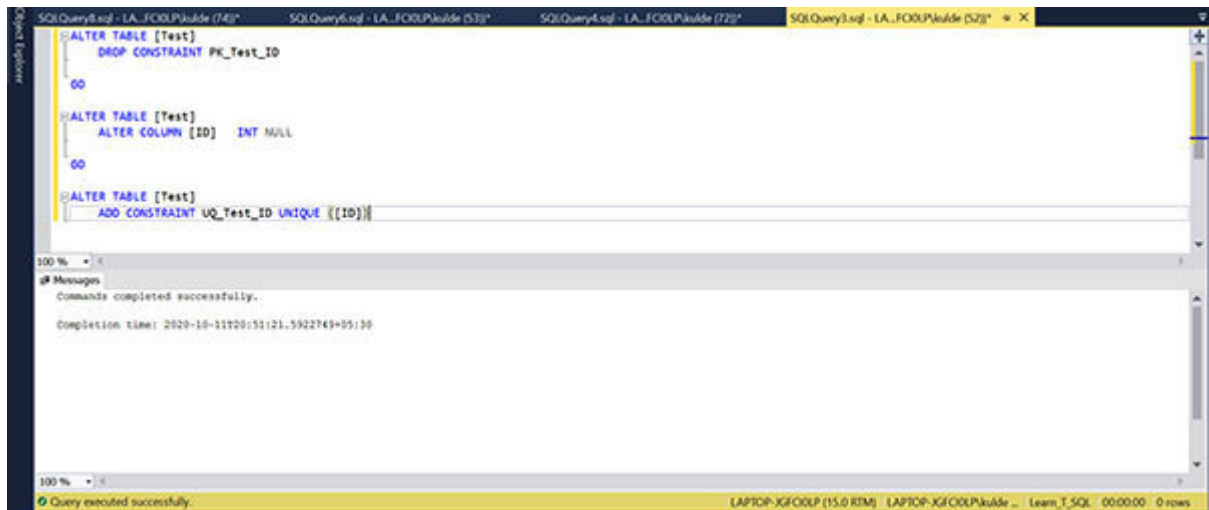


Figure 2.28: Executing ALTER TABLE script to add unique key

You'll find the unique key created under the **Keys** folder of the respective table folder, as can be seen in the following screenshot:

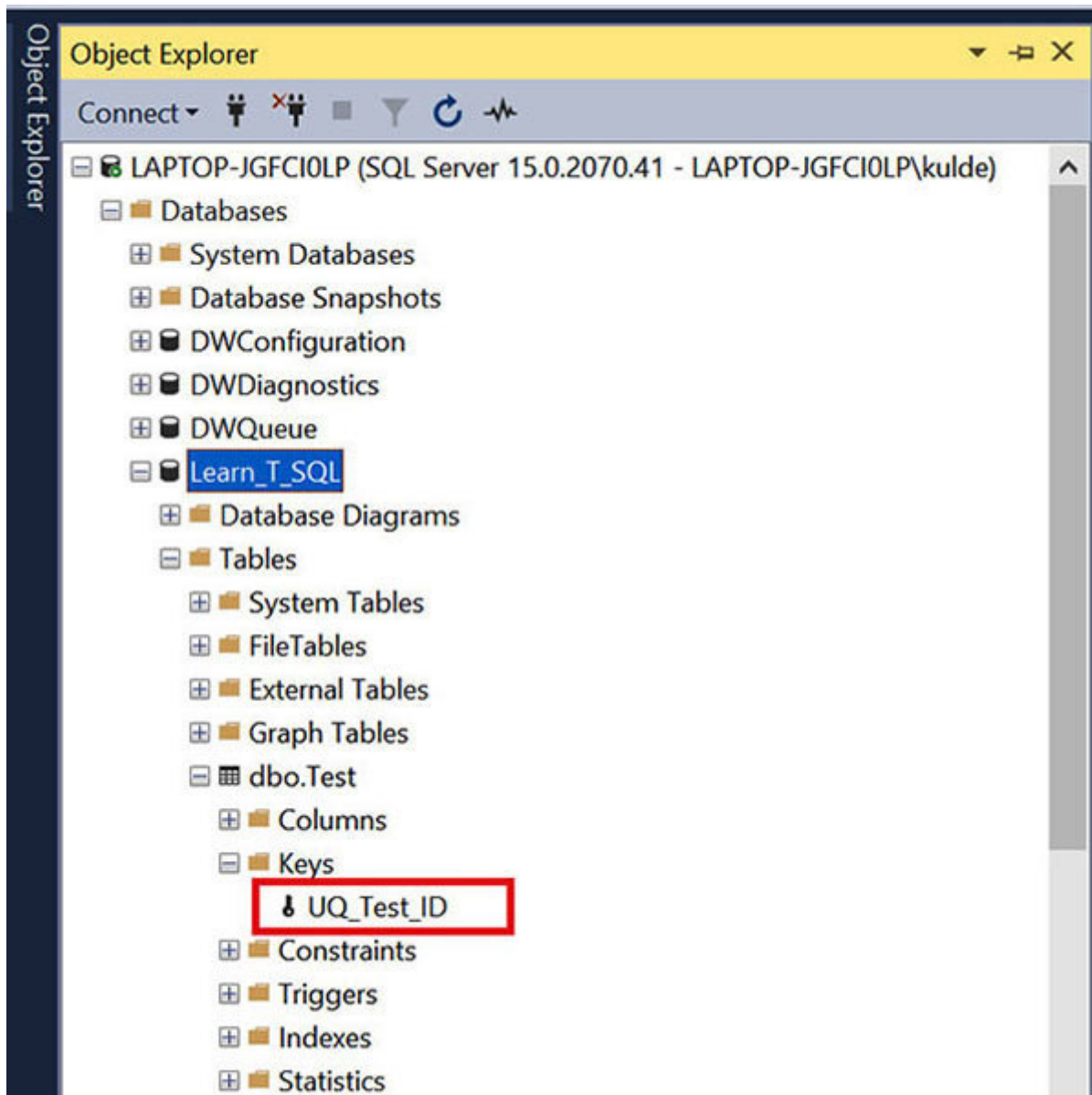


Figure 2.29: Unique Key created with ALTER TABLE script is reflecting in the Object Explorer

Primary Key versus UNIQUE constraint

Primary Key and UNIQUE constraint behavior are almost the same. Both of these constraints can be created on a single column or the combination of columns. Although there are certain differences as follows:

The purpose of the Primary Key constraint is to uniquely identify each record, whereas the **UNIQUE** Key constraint enforces uniqueness on a column or set of columns.

Only one Primary Key constraint can be created on a table, whereas a table can have multiple **UNIQUE** Key constraints.

Primary Key constraint by default creates a clustered index on the Primary Key columns, and with the name of the Primary Key constraint. Although, there is also a provision to specify whether the Primary Key should create a clustered or a non-clustered index. If the type of index is not specified then by default a clustered index would be created; whereas, UNIQUE Key constraint by default creates a unique non-clustered index. Although, provision is there to specify whether the unique key should create a unique clustered or a unique non-clustered index. We'll talk about indexes more in [Chapter 3](#).

Primary Key creates NOT NULL constraint on the column(s) forming Primary Key, whereas UNIQUE constraint doesn't

creates the NOT NULL constraint.

Primary Key column(s) do not accept the NULL values, whereas the UNIQUE constraint column(s) allow the NULL values provided such combination is unique. It means if the UNIQUE constraint is on one column then only one row can have NULL. Similarly, the UNIQUE constraint on two columns can have the following combination of NULL values.

values.
values. values.
values.
values.

Table 2.1: *Null values in the unique constraint columns*

DEFAULT constraint

The DEFAULT constraint is a special kind of constraint that is used to assign a default value to the column (if no value is assigned to it).

Let us add one more column to our table **Test** to accept gender. Gender could be or **Do not wish to**

In our example of the DEFAULT constraint, we'll assign **Do not wish to mention** as a default value to the **Gender** column. So, if the **Gender** column is not included in the then the default value **Do not wish to mention** will be assigned to the respective rows being inserted.

If a column is Nullable, it means it accepts the NULL value, and it doesn't have a DEFAULT constraint. All the rows inserted in such a column will have NULL if it is not part of the INSERT statement.

This is how **DEFAULT** constraint can be created while creating a new table:

```
CREATE TABLE [Test]
(  
[ID] INT
```

```
,  
,  
, CONSTRAINT PK_Test_ID PRIMARY KEY ([ID])  
, CONSTRAINT DF_Test_Gender not wish to  
)
```

But, since we have an existing table, we'll add a new column and **DEFAULT** constraint on it:

```
ALTER TABLE [Test]
```

```
ADD [Gender] CONSTRAINT DF_Test_Gender not wish to
```

If you have the column already available, then the **DEFAULT** constraint can be created as follows:

```
ALTER TABLE [Test]  
ADD CONSTRAINT DF_Test_Gender not wish to FOR  
[Gender]
```

You'll find the **DEFAULT** constraint created under the **Constraints** folder of the respective table folder as shown in the following screenshot:

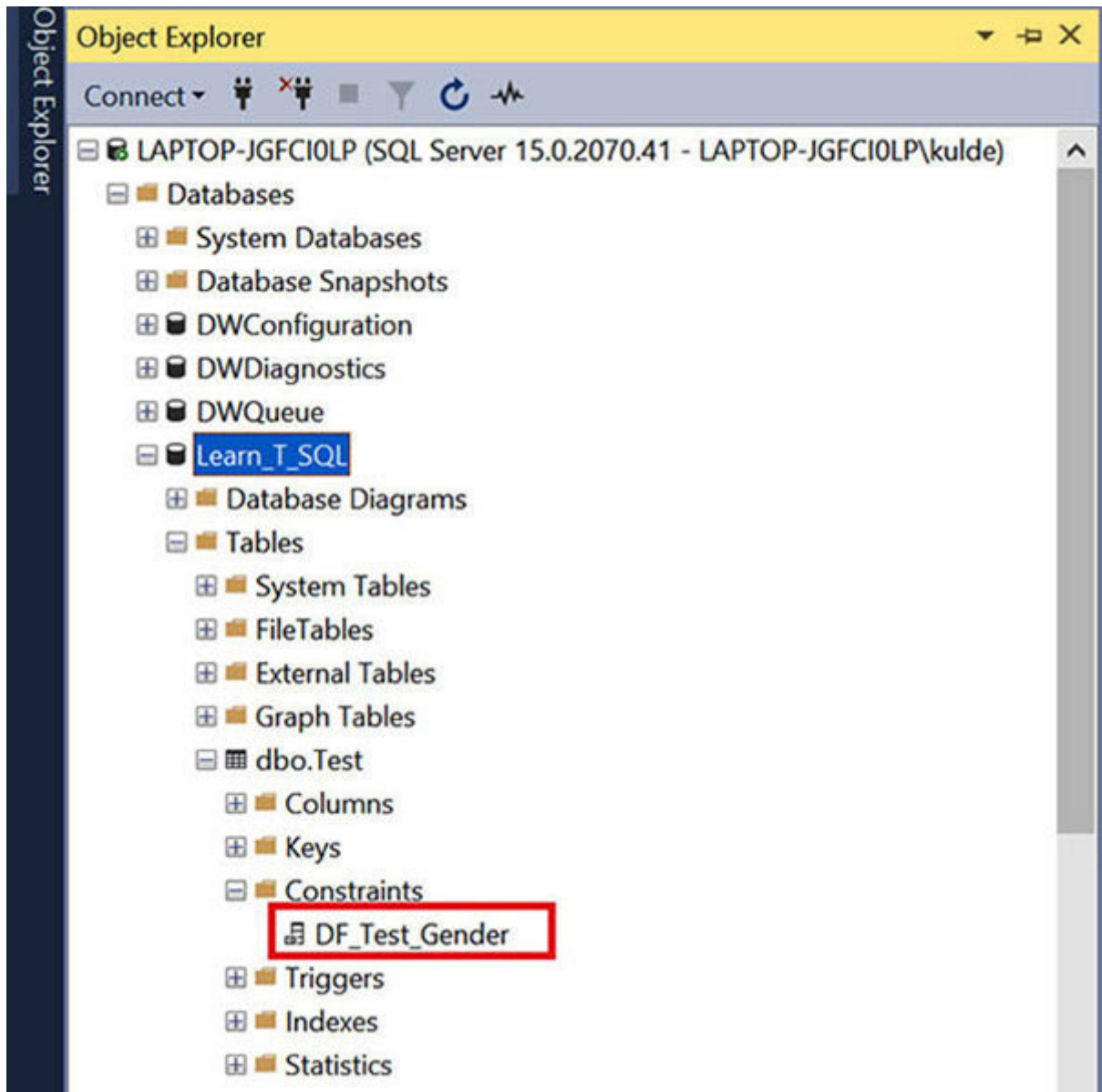


Figure 2.30: Default constraint is reflecting in Object Explorer

CHECK constraint

CHECK constraint is another special kind of constraint that is used to specify the rule of the accepted values in the column.

For example, if we want to restrict the **Gender** column to only accept the values **Female**, or **Do not wish to** then it can be achieved with the help of CHECK constraint as shown in the following example. If any other value will be supplied other than the ones specified, an error will be thrown.

```
ALTER TABLE [Test]
ADD CONSTRAINT CK_Test_Gender IN 'Do not wish to
```

If you wish to add **CHECK** constraint at the time of table creation then it can be done as follows:

```
CREATE TABLE [Test]
(
[ID]    INT
,
,
, CONSTRAINT PK_Test_ID PRIMARY KEY ([ID])
, CONSTRAINT DF_Test_Gender not wish to
, CONSTRAINT CK_Test_Gender IN 'Do not wish to
)
```

You'll find the **CHECK** constraint created under the **Constraints** folder of the respective table folder, as can be seen in the following screenshot:

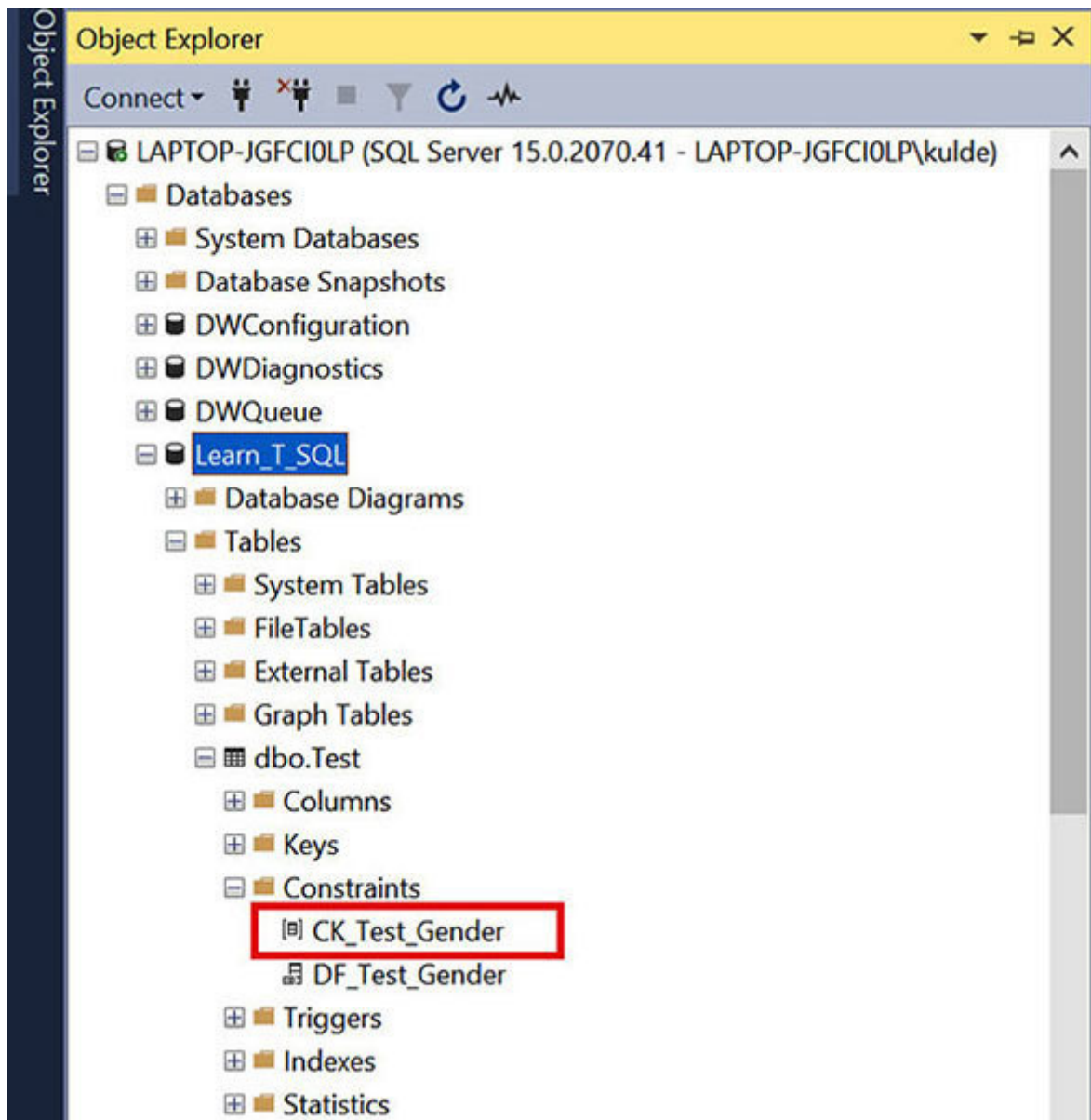


Figure 2.31: Check constraint is reflecting in Object Explorer

Foreign Key constraint

In [Chapter 1, Getting](#) we also talked about Foreign Key in the general sense. We'll now understand more about Foreign Key and learn to apply it.

The Foreign Key is a type of constraint which references to another table. It makes sure that the value being inserted or updated in the Foreign Key column exists in the referenced column of another table (called Primary Key). Referenced table column should have Primary Key or UNIQUE constraint defined on it.

The Foreign Key constraint column(s) can have the NULL value. But Primary Key constraint column(s) cannot.

Primary Key table (referenced table) is treated as a *parent* and Foreign Key (referencing table) is treated as

A table can have self-reference which means Foreign Key referencing to the Primary Key of the same table.

If a table having Primary Key (parent) is referenced in another table as Foreign Key (child) then parent table can't be dropped. Similarly, the parent row can't be deleted unless the

child is deleted. So first the child is to be deleted and then the parent.

The following is an example of self-referencing Foreign Key constraint. In the following example, we can see how to create the Foreign Key at the time of creating the table:

```
CREATE TABLE [Test]
(
[ID] INT
,
,
, [Parent_ID] INT
, CONSTRAINT PK_Test_ID PRIMARY KEY ([ID])

, CONSTRAINT DF_Test_Gender not wish to
, CONSTRAINT CK_Test_Gender IN 'Do not wish to
, CONSTRAINT FK_Test_Parent_ID FOREIGN KEY ([Parent_ID])
REFERENCES [Test] (ID)
)
```

The following is the example to add a new column **Parent_ID** (since we don't have it) along with the **FOREIGN KEY** on it:

```
ALTER TABLE [Test]
ADD [Parent_ID] INT CONSTRAINT FK_Test_Parent_ID
FOREIGN KEY ([Parent_ID]) REFERENCES [Test] (ID)
```

The following is an example to add a **FOREIGN KEY** constraint on the existing column of a table.


```
ALTER TABLE [Test]
ADD CONSTRAINT FK_Test_Parent_ID FOREIGN KEY
([Parent_ID]) REFERENCES [Test] (ID)
```

In our example, we've created the Primary Key and Foreign Key on the same table. However, you can have them on different tables. The Primary Key can be on one table (Parent table) and Foreign Key on another table or in multiple tables (Child tables). The syntax will remain the same.

You'll find the Foreign Key created under the **Keys** folder of the respective table folder as shown in the following screenshot:

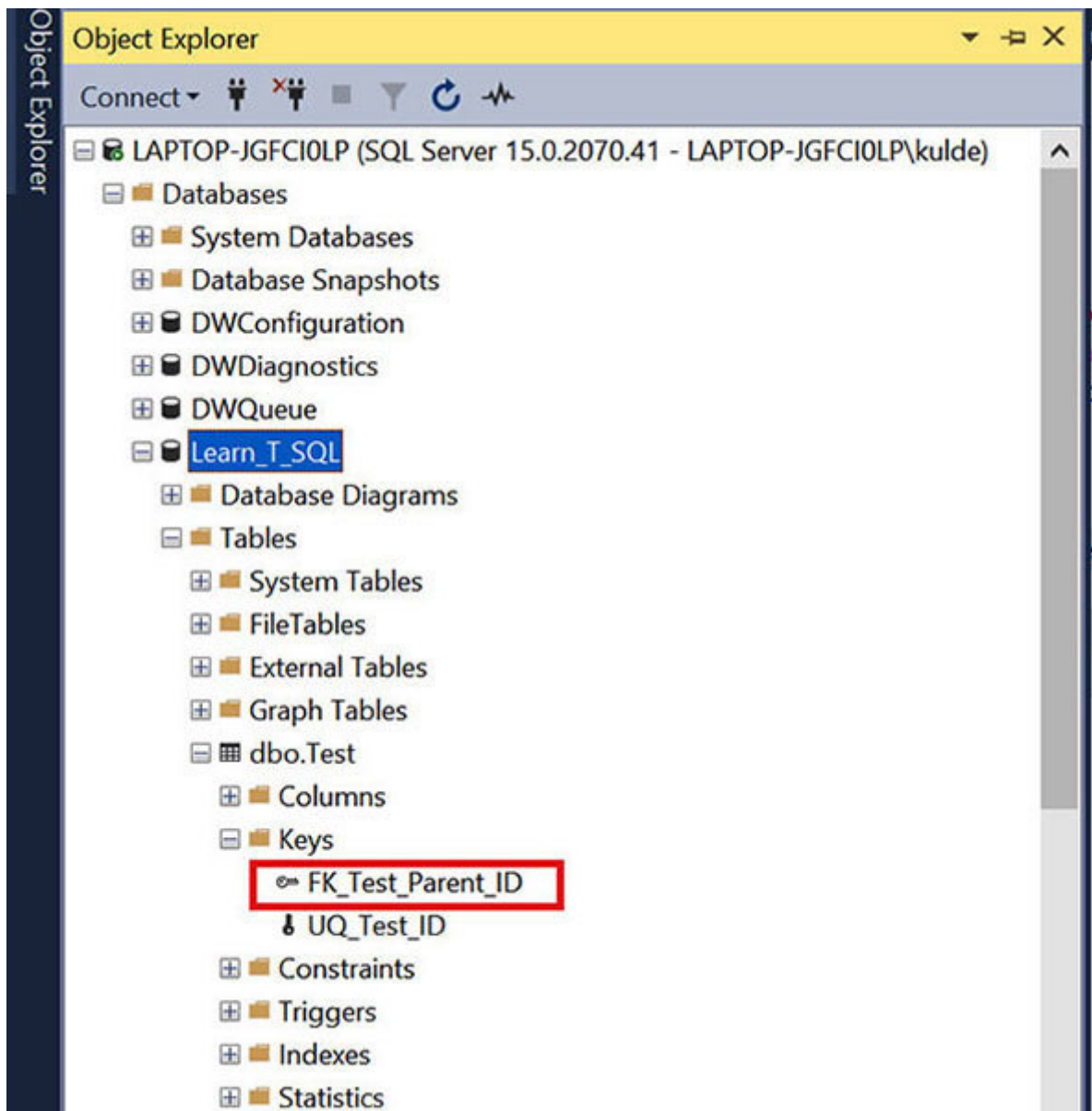


Figure 2.32: Foreign Key is reflecting in Object Explorer

CASCADING in Foreign Key.

Cascading is the process of replicating the changes performed on the parent table to the child tables. There are two kinds of cascading available with Foreign Key: **ON DELETE** and **ON**

ON DELETE CASCADE deletes the child rows from the child tables whenever referencing parent rows are deleted.

ON UPDATE CASCADE updates the Foreign Key column of the child rows (with the modified Primary Key) in the child tables, whenever the Primary Key is updated in the referencing parent rows.

Either **ON DELETE CASCADE** or **ON UPDATE CASCADE** or both can be defined in Foreign Key:

```
ALTER TABLE [Test]
ADD CONSTRAINT FK_Test_Parent_ID FOREIGN KEY
([Parent_ID]) REFERENCES [Test] (ID)
ON DELETE CASCADE
ON UPDATE CASCADE
```

Conclusion

In this chapter, we talked about tables and various prerequisites to create a table such as data types and constraints. You've also learned to create, modify, and delete tables. You've also learned to implement various constraints.

We also discussed that no object (including table) can be created without a database. All the objects are to be created within a database. We also talked about schema and three-part naming.

In the next chapter, we'll learn about indexes.

Points to remember

You can't create the table without the database. The table has to be created within a database. The database is the placeholder for tables and other database objects.

Default **name>** is the current database as selected in the query editor.

Each user database has a default schema. If no schema is assigned while creating the objects then it's created under the **dbo** schema. **dbo** stands for database owner.

You can't change the data type of the column to another data type if the data residing in the column is not compatible with the new data type. For example, if the current data type is **varchar** and column holds alphanumeric data then you can change it to **int** or any other numeric data types. If you want to do so then you'll have to remove the data from that column and then only you can do it.

If the column contains the duplicate data, then you can't add the Primary Key constraint.

If table has Primary Key which is referenced as Foreign Key in another table, then you cannot drop the table unless the

corresponding Foreign Key data from another table is deleted/removed.

When you make column(s) Primary Key then NOT NULL constraint automatically gets created on those set of column(s), irrespective you specify it or not. Whereas, UNIQUE constraint doesn't create the NOT NULL constraint, and it accepts the NULL value.

If a column is nullable that means it accepts a NULL value and it doesn't have a DEFAULT constraint. If the column is not included in the **INSERT** statement, then by default NULL will be assigned to the respective rows being inserted.

If a table having Primary Key (parent) is referenced in another table as Foreign Key (child) then the parent table can't be dropped. Similarly, the parent row can't be deleted unless the child is deleted. So first the child is to be deleted and then the parent.

The Primary Key could be a Unique Key, whereas the Unique Key can't be a Primary Key.

Multiple choice questions

You want to create a column that will have numeric data without decimal and will not have values more than 500. Which data type amongst the following should you choose?

SMALLINT

DECIMAL

BIGINT

TINYINT

You want to store the currency conversion rate with a maximum value Which data type amongst the following should you choose?

NUMERIC

DECIMAL

FLOAT

NUMERIC(12,6)

You've two tables Student and Student table has StudentID column which has unique data but no Primary Key is specified on it. StudentResult table also has StudentID column. Can you create Foreign Key on the StudentResult table?

Yes

No

Which of the following statements are true for Primary Key?

Primary Key accepts duplicate values.

Primary Key can be created on the nullable column.

Primary Key should be created on non-nullable columns having unique values.

Primary Key cannot be created on a set of columns.

Answers

A

D

B

C

Questions

Which data types should be used for decimal values if accuracy is important?

If you got a data and has a column for Character length of address varies for each row in your data but has a maximum of 300 characters. The address will always be in the English language. What would be the data type that you will choose for **Address** column?

If you are creating a table to store the data for a survey. This survey also has a section where the user needs to read the privacy statement and confirm with either **Yes** or There could not be another value? You've to decide the data type to store the user confirmation. Which data type will you choose?

Extending the scenario of *Question* Suppose the user can confirm with either or Now which data type and constraints will you choose?

What are various cascading options available with Foreign Key?

If you have data that include an attribute holding only the date (without time). Which data type will you choose?

Can you have more than one Primary Key on a table?

Can you have more than one Foreign Key on a table?

What is NULL and what is the purpose of NOT NULL constraint?

Why CHECK constraint is used?

Key terms

dbo: Stands for Database Owner. It's also the default schema of each database.

Precision: Means the number of digits in a number (including both left and right of the decimal point).

Scale: Means the number of decimal places (right of decimal point).

Three-part naming: It is a method to specify the object name in SQL Server. The first part is database name. The second part is schema name. The third part is object name.

Boolean: It is the terminologies to represent Bit or Binary data.

Null: It is equivalent to If you don't know the value then it's NULL.

CHAPTER 3

Index

We learnt to create the table in [Chapter 2, Table](#). A table holds multiple rows and columns. The number of rows in the table can grow to a huge count.

Consider you have been told to reach an address without the city being conveyed to you. *How would you find an address?* Yes, you can find it, since it exists on the earth. But you must agree, it would not be easy. There would be a lot of struggles. It will also take a lot of time. Wandering through the world to find an address will never make sense. So, it is better to have the relevant reference of the address such as country, state, city, area, locality, ZIP/PIN, apartment, and so on to locate the address quickly.

The database is no different than the real-world scenarios. If you have to fetch specific records from the billions of rows of a table, it will take a lot of time to search without a proper reference. Indexes maintain the relevant pointers of the records to locate the data quickly, without scanning the whole set of data.

Structure

In this chapter, we will cover the following topics:

Index

Types of indexes

Rowstore

Clustered index

Non-clustered index

◇ Covering index

◇ Filtered index

Unique index

Columnstore

Clustered columnstore index

Non-clustered columnstore index

Filtered columnstore index

DROP INDEX

Indexes in practical

Objective

You'll learn about the indexes. It plays a crucial role in making effective use of the databases. Data is returned quickly with indexes. Imagine you need an urgent report, but the query is running for hours. Indexes improve the performance of the read queries.

In this chapter, you'll get to know the various indexes in SQL Server and how to create and drop them. You will also learn the practical use-case of indexes.

[Index](#)

Word *index* is very familiar to us. Don't you think so? We have all known this terminology since our childhood.

We all must have touched the book at least once in our lifetime. Every book has an Index, the relevant chapters, and topics in each chapter. Same thing you'll find in this book as well. Index in a book has the page number of the Chapters. A chapter contains the structure, which depicts the topics along with the sequence.

Just imagine, if you have the book of around 500 similar to this book and you are asked to search the topic *create* Without an index in the book, *what do you think would you be able to search it?* If yes, then *how soon?*

I'm sure there won't be a book without an index. The relational database has leveraged the indexes in the same manner. Its core purpose is to fetch the data quickly. As we learnt in [Chapter 1, Getting](#) the fundamental of data storage in SQL Server is page. The data is written in the pages called **data** Similarly, there is another kind of page called the index page used to store the index entries. All the index entries are written in the index pages.

For understanding purposes, you can relate the data file (.mdf) and files with the book. When a request is submitted to SQL Server to fetch any data, it first scans through the index page to find the location and reference of the data pages to pick and return the data from the relevant pages.

If you remember, we also understood in [Chapter 1, Getting Started](#) about the size of a page. The size of each page is 8 KB. So, if the database size is 100 GB, then it means there are approximately **1,31,07,200** Each table is assigned extents. Each extent is composed of **8** Now, you can easily relate to the importance of the index. Without the index, SQL Server has to scan through all the pages associated with the table. It'll result in more time taken to execute a query.

Indexes in SQL Server follow B-tree structure. More on B-tree, index architecture, and index design guide can be read at the following Microsoft official documentation URL:

<https://docs.microsoft.com/en-us/sql/relational-databases/sql-server-index-design-guide?view=sql-server-ver15>

We understood the index. Now let's understand the various workloads for which SQL Server database can be used. You came across the new terminology It can be understood as the purpose for which the SQL Server database is being used. Generally, the relational database workload is divided into two categories – **OLTP** and The full form of OLTP is **Online Transaction Processing** and the full form of OLAP is **Online**

Analytical The name itself represents the purpose of each of these workloads.

OLTP is also referred to as transactional workload. It refers to the normal system including windows, web, and mobile applications used for serving the day-to-day work. Examples of OLTP workload are banking systems, ERP systems, shopping websites, stock trading applications, and so on.

Similarly, OLAP is also referred to as analytical workload. When you come across the term analytics, it always refers to a system that is different from the transactional workload. The system falling under this category is generally not used for day-to-day work. Such systems are used to get the insights of the transactional data into analysis and decision-making purposes.

For example, a company into online shopping may want to know which city had the maximum sales or which products were in demand. Similarly, the company may also want the analytics at various other levels, in order to analyze how the business is doing. At the same time, data from the analytical workload can be used by the company to make critical decisions for the business such as which city to target more, which product to launch, which product needs more marketing, and so on.

Indexes as we understood so far speed up the query execution. However, it is to be noted that they only speed up the data retrieval. They do not speed up the insert the

operations. Instead, they add some overhead to insert the operation. **INSERT** statement on a table without any index will run faster as compared to with indexes.

At the same time, indexes may speed up the update and delete operations if the query has a filter clause, by helping the query to trace the rows to be updated or deleted quickly, whereas there will be some overhead in the write part of these update and delete operations. This overhead is due to additional work that is required to be performed by SQL Server to maintain index entries. Overall, update and delete statements will be benefited too with indexes if it has a where clause.

We'll now discuss the most commonly used indexes in SQL Server. SQL Server indexes are generally divided into two categories: **rowstore** and We'll understand each of these in the following topics.

Rowstore indexes

You must now be well versed with OLTP workloads. Rowstore indexes are traditional indexes. They are used to speed up the retrieval of data by allowing queries to locate the row data quickly with the help of an index, instead of scanning the entire table. Rowstore indexes are logically organized as a table with rows and columns, and the data are also stored in the form of rows and columns. There are two kinds of rowstore indexes: **clustered** and Both of these kinds of indexes have a specific use.

Rowstore indexes are the best fit for the queries retrieving data by searching for a particular value.

Clustered index

Clustered indexes are organized in B-Tree structure, and maintains the entire rows of a table in its leaf node. Most importantly, the rows stored in the data pages of the leaf node are sorted, as per the sort order defined on the clustered index key. Data are stored in the sorted order only once and that is with the clustered index. If the table does not have the clustered index, then data won't be sorted, even though there are other non-clustered indexes on the table. A table without a clustered index is called a heap table.

Figure 3.1 depicts the structure of a clustered index, and an example of how the search works with the clustered index.

Clustered indexes follow the B-Tree structure in which there are three kinds of nodes. It is advised to relate the below explanation with the image depicted in figure

Root Node: It is an index page. It holds the range of clustered index keys and the child page id (belonging to intermediate nodes) in which these clustered index keys are further narrow down.

Intermediate Nodes: These are also index pages. There could be multiple levels of the intermediate nodes. The levels of the intermediate nodes will depend upon the number of rows in the table and the size of the clustered index key in bytes. The B-Tree structure is managed internally by the SQL Server engine, and we don't have any control over it. In [figure](#) you can see only one level of the intermediate node, but it can further narrow down to multiple levels. The final level of the intermediate node will have the child page ID of the data page in which the data resides.

Leaf Nodes: These are the data pages that hold the actual data. The leaf level of clustered index holds the data pages that contain the sorted data rows ordered by the clustered index key columns.

A table in SQL Server cannot have more than one clustered index. The reason is simple because clustered index leaf node stores the data in the sorted order of the clustered index key column(s). *Does it make sense to have multiple copies of the same data, or simply the redundant data?* Obviously No! That's why there may be the restriction of just one clustered index. This is quite logical too.

Suppose we've a table **Customer** that has thousands of customer records. If we want to read a specific customer record then clustered index makes it easier and quicker. As it is depicted in the following figure, upon submitting the select request, the row is located and fetched in two steps.

Whereas, in the absence of a clustered index, data are stored in the heap. Heaps do not follow any B-Tree structure. Heap is simply a bunch of data pages holding the table data. If we've to perform the same operation of reading a customer from the list of thousands of the customers spread across hundreds of the data pages, SQL Server engine will have to scan through all the pages individually to return the requested data. This shall be certainly a time-consuming process and would take more time. However, such kinds of reading requests can be optimized even without clustered index by using the non-clustered indexes. We'll talk about non-clustered indexes in the upcoming topic:

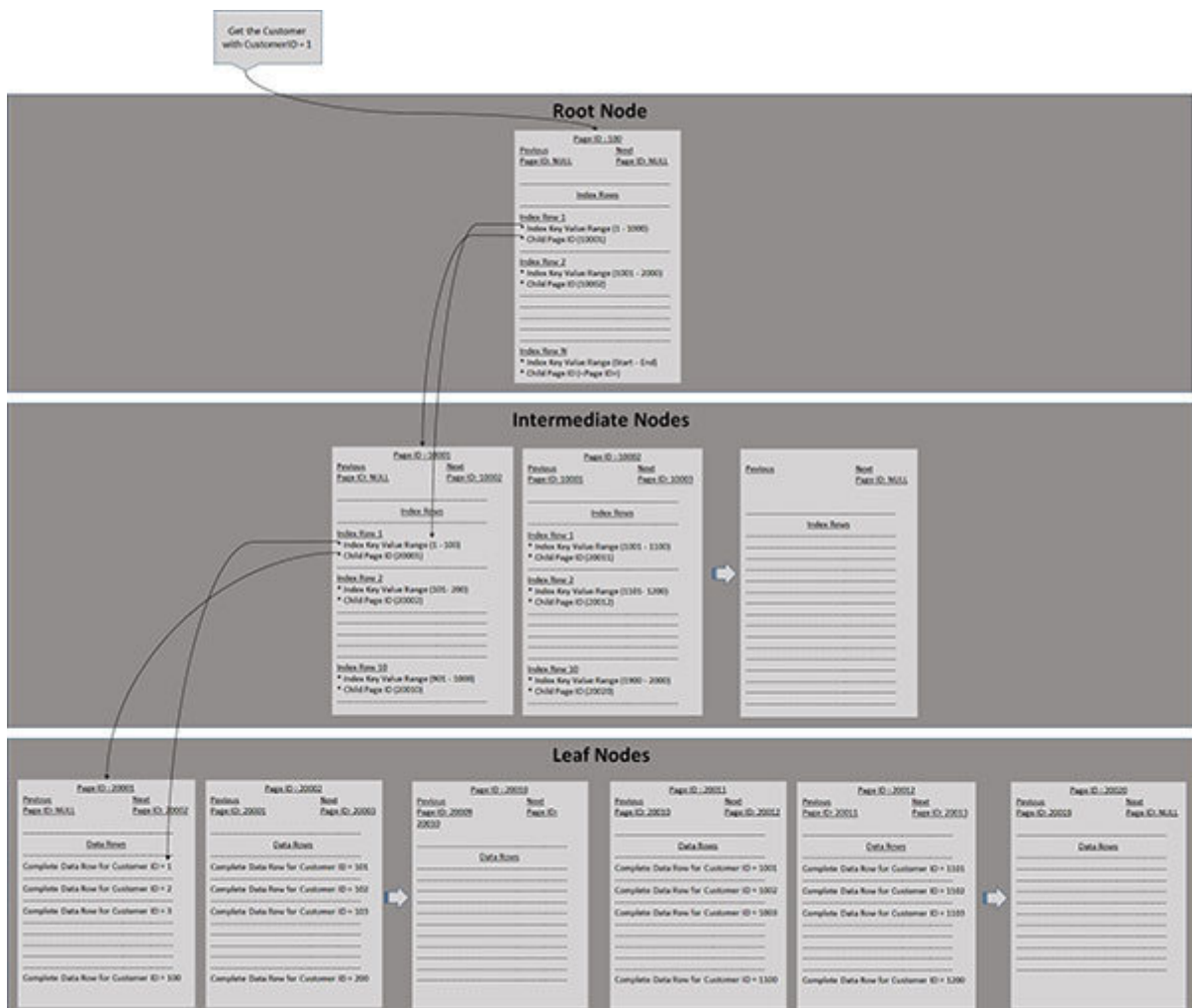


Figure 3.1: *Clustered index*

Index creation is one aspect, but using it is another. You may have a clustered index or another bunch of non-clustered indexes, but if you've not written the query appropriately then you won't be benefitted from the indexes.

In order to make use of the available indexes (clustered and non-clustered indexes), it is important to write the query in such a way that it uses the index key columns in the filter condition condition), and join condition. The columns in the filter and join condition should follow the same sequence as defined in the index.

For example, if an index is created on *Col1*, and *Col3* columns, try to follow the same sequence in the **WHERE** clause as well. The first column in the index is the second is and the third is Try to match the sequence in the filter condition as well. However, it is not mandatory and not always possible.

If not all the columns of an index are possible to include in the filter or join condition, but starting few columns of the index is included, in the same sequence as defined in the index, the index shall be used. For example, if the filter or join condition includes *Col1* column, or *Col1* and *Col2* columns, the index shall be used. Whereas if only the **Col2** column is used, the index shall not be used.

We'll talk about filter conditions condition) in [Chapter 4](#), and joins in [Chapter 6, Join, Apply, and](#)

One more important aspect of the indexing is the Cardinality simply refers to the number of unique values in a column. More unique values mean the **good** Less unique values mean the **poor**

Always, try to use the columns with the good cardinality in your indexes. If there are multiple columns in your index then columns with good cardinality should lead and the rest shall follow. Columns with good cardinality should be first in the list, and the column with the poor cardinality should be last in the list.

The data in the heap is stored in the order in which the rows are inserted into the table. Heaps do not store the sorted data as clustered indexes do.

Qualities of a good clustered index key are as follows:

It should be as *narrow* as possible in terms of the number of bytes it stores.

It should be *unique* else SQL Server will add a *uniquefier* to make it unique. The clustered index key is always unique. If you will not explicitly choose the right columns with unique values for it, SQL Server will add 4-byte *uniquefier* to make it

unique. You can create the clustered index on columns having duplicate values, but it doesn't imply you should always create it. It is better we do our job than leaving the SQL Server engine to do it. If SQL Server does it for us, then it would be an overhead.

It should be ideally, never updated like an identity column. The clustered index has to sort the data before it is written to the data pages. So, if the clustered index key values themselves are changing then SQL Server will have to sort the entire table again in order to maintain the sorted data in the leaf level.

It should be like an identity column to avoid fragmentation and improve the write performance.

There are two ways SQL Server chooses to use the index: **index seek** and **index scan**. Index seek means SQL Server can trace the matching records based on the index key. The index key is formed based on the values of the columns on which the index is created. In the case of index seek, only the qualifying pages are being fetched, whereas index scan means SQL Server is not able to trace the matching records based on the index key; hence it has to scan through every row in the table. It can be easily derived from the explanations of both seek and scan. Seek would generally perform better on a medium to a large dataset. If the dataset is smaller, then the scan may perform better than seek.

You can relate the preceding explanation of seek and scan with a simple example of a book that has just 10 You may be able to trace a topic of the book quickly by scanning through all the pages, instead of first tracing the topic in the index and then navigating to the topic.

The general practice says that each table should have a clustered index. Clustered indexes are always a good choice if the filter condition of the query is using the columns on which the clustered index has been defined. It can benefit select, update, and delete queries. There are other benefits of the clustered index such as they enable the SQL Server to lock the specific row while running the update and delete statements, whereas, in the absence of the clustered index, the entire range of rows being scanned will be locked.

The clustered index also has a performance overhead. As we already understood, and **delete** statements performing write operation would take more time as compared to without clustered index. This overhead is caused due to an additional step of maintaining the indexes, and obviously, the sorting involved.

Always remember, indexes are meant for optimizing the read requests. You may find it useful for read loads (more SELECT queries and fewer INSERT, UPDATE and DELETE queries). So, if the database is write intensive (more INSERT, UPDATE and DELETE queries and fewer SELECT queries) then it is advisable to have fewer indexes as much possible. Although,

in few scenarios, the update and delete queries may be benefited if they have filter conditions on a clustered index column(s). It will also enable to lock the specific rows if there is a strong filter condition restricting the specific rows. If there are too many rows to be updated or deleted then SQL Server may choose clustered index scan, and it would not benefit in performance. But it will surely slow up the speed of the query execution.

Syntax to create clustered index:

```
CREATE CLUSTERED INDEX name>  
ON
```

```
name> (1 name> 2 name> .... n name>
```

The explanation of the syntax to create the clustered index is as follows:

CREATE CLUSTERED INDEX is the keyword to be specified to create a clustered index

name> to be replaced with the name of index. You have the liberty to assign the name as you wish. However, keeping it as per the naming convention best practices is always a good option.

name> to be replaced with the actual table name on which the clustered index is to be created.

(1 name>, 2 name>, ... n name>) is called **clustered index key** and each column in the key is called **clustered index key**. Each column in the key has to be assigned the sort order. If the sort

order is not defined then the default ascending sort order will be considered by SQL Server. When you'll create the clustered index then you need to specify the name of columns on which you desire to create the clustered index.

Non-clustered index

Non-clustered indexes as opposed to the clustered index do not store the copy of row data. It maintains a pointer to the clustered index or heap to locate the rows data.

[Figure 3.2](#) depicts the structure of a non-clustered index, and an example of how the search works with the non-clustered index.

Non-clustered indexes also follow the B-Tree structure in which there are three kinds of nodes. It is advised to relate the following explanation with the image depicted in [figure](#)

Root Node: It is an index page. It holds the range of non-clustered index keys and the child page ID (belonging to intermediate nodes) in which these non-clustered index keys are further narrow down.

Intermediate Nodes: These are also index pages. It is an optional node in the case of non-clustered indexes. If the dataset is small then the B-Tree shall end-up only in two nodes – **root** and **leaf** nodes. There could be multiple levels of the intermediate nodes. The levels of the intermediate nodes will depend upon the number of rows in the table and the size of the non-clustered index key in bytes. The B-Tree structure is managed internally by the SQL Server engine, and we don't have any control over it. In [figure](#) you can see only

one level of the intermediate node, but it can further narrow down to multiple levels.

Leaf Nodes: These are also index pages. It holds the pointer to the clustered index key or heap RID (key). These keys are the pointer to the row located in the data pages of the clustered index or heap:

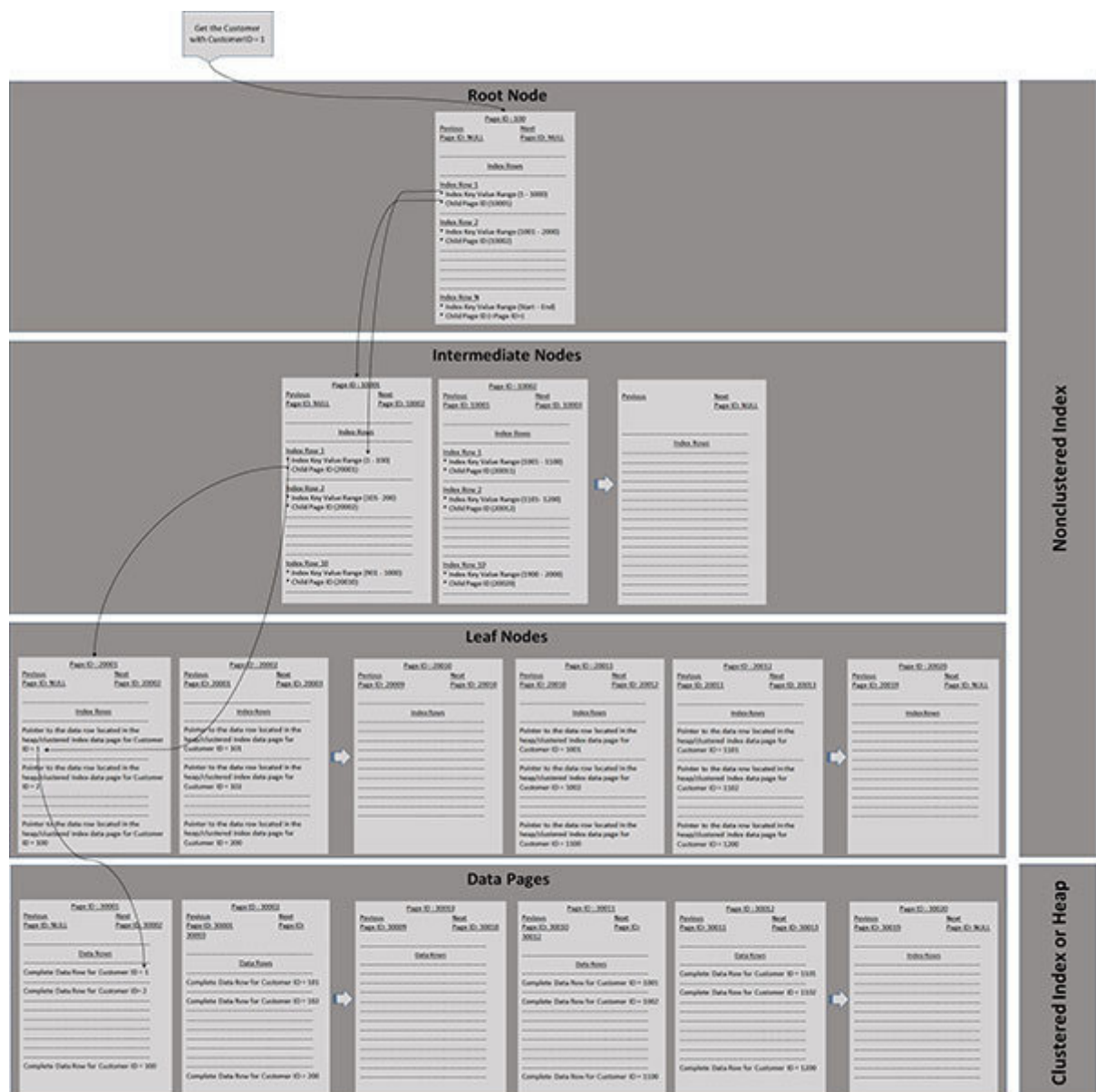


Figure 3.2: Non-clustered index

The index seek behavior of the non-clustered index is different from that of the clustered index. Clustered index seek do not have to locate the data pages as it already has the entire row data with it. Whereas, non-clustered index seek has to perform an additional step called **bookmark lookup** which is of two types - **key lookup** and **RID**

If the table has a clustered index, and the non-clustered index is able to trace the index entries after seek, then it will fetch the data from the clustered index leaf node data pages based on the clustered index (key) reference it holds. It is called **key**

Whereas, if the table is a heap table (without a clustered index) then the data is fetched from the heap data pages based on the heap RID (key) reference it holds and is called **RID**

Heap RID (key) and clustered index (key) are the pointers to the data pages in clustered index or heap. These pointers include the information to locate the data row. The information includes – database file, allocation unit, page ID, and slot number, and so on to locate the rows quickly.

Always remember, key lookup is when the table has clustered index. RID lookup is when the table does not have clustered index (heap table). Bookmark lookups (key and RID lookup) can be very expensive if it has to deal with the large number of rows.

The syntax to create non-clustered index:

```
CREATE NONCLUSTERED INDEX name>  
ON
```

```
name> (1 name> 2 name> .... n name>
```

The explanation of the syntax to create non-clustered index is as follows:

CREATE NONCLUSTERED INDEX is the keyword to be specified to create a non-clustered index.

name> to be replaced with the name of index. You have the liberty to assign the name as you wish. However, keeping it as per the naming convention best practices is always a good option.

name> to be replaced with the actual table name on which the non-clustered index is to be created.

(1 name>, 2 name>, ... n name>) is called **non-clustered index key** and each column in the key is called **non-clustered index key** column. Each column in the key has to be assigned the sort order. If the sort order is not defined then the default ascending sort order will be considered by SQL Server. When you'll create the non-clustered index then you need to specify the name of columns on which you desire to create the non-clustered index.

More on clustered and non-clustered can be read from the following URL of Microsoft official documentation:

<https://docs.microsoft.com/en-us/sql/relational-databases/indexes/clustered-and-nonclustered-indexes-described?view=sql-server-ver15>

From a performance stand point clustered index seek is always better than non-clustered index seek. But we can't create the clustered index on all the filter criteria. It is not possible and not logical too. That is why SQL Server has come up with *covering* a solution to avoid the key and bookmark lookups.

Covering index

The covering index is not different than the non-clustered index. It is also a non-clustered index. The only difference is it includes other columns as well apart from ones used in the index key. Covering index helps avoid the key and bookmark lookup by storing the additional columns values. Hence whenever you'll request to retrieve the data and if the columns requested are part of the index key or **INCLUDE** clause, there won't be a key or bookmark lookup.

INCLUDE clause is used to cover the additional columns and make a non-clustered index a covering index. It is to be noted that the columns used in the index key and **INCLUDE** clause should not clash. That means once you've mentioned a column in index key, you can't use the same column in the **INCLUDE** clause and vice versa.

For example, if the table has ten columns. *Col1*, ..., The index is created on *Col1*. There is a select query that returns *Col2* and *Col3* and it has a where clause on *Col1*. In this case, there will be index seek. But there will be an additional bookmark lookup as well. As we already discussed, non-clustered index does not store the row data, instead, it stores the pointers to the row data. In this case, *Col2* and *Col3* are not part of the index. Also, these columns are not meant for filtering, so it does

not make sense to add them in the index key. So, if we want to avoid the bookmark lookup in this case, then these two columns – *Col2* and *Col3* can be covered in the index using the **INCLUDE** clause, and the index will become the covering index. There won't be any bookmark lookup after the covering index is created, as far as any other columns (excluding *Col1*, and are not included in the **SELECT** query.

The syntax to create non-clustered covering index is as follows:

```
CREATE NONCLUSTERED INDEX name>  
ON
```

```
name> (1 name> 2 name> .... n name>  
INCLUDE (a name>, b name>, .... z name>)
```

The explanation of the syntax to create non-clustered covering index is as follows:

CREATE NONCLUSTERED INDEX is the keyword to be specified to create a non-clustered index.

name> to be replaced with the name of the index. You have the liberty to assign the name as you wish. However, keeping it as per the naming convention best practices is always a good option.

name> to be replaced with the actual table name on which the non-clustered index is to be created.

(1 name>, 2 name>, ... n name>) is called **non-clustered index key** and each column in the key is called **non-clustered index key**. Each column in the key has to be assigned the sort order. If the sort order is not defined then the default ascending sort order will be considered by SQL Server. When you'll create the non-clustered index then you need to specify the name of columns on which you desire to create the non-clustered index.

INCLUDE is used to make the non-clustered index a covering index by including the additional columns, other than ones used in the index key.

(a name>, b name>, ... z name>) is called covering columns or included columns. Columns you wish to cover or include have to be specified here. You can't specify the column that has been already used in the index key.

You can read more about covering indexes at the following URL of Microsoft official documentation:

<https://docs.microsoft.com/en-us/sql/relational-databases/indexes/create-indexes-with-included-columns?view=sql-server-ver15>

Filtered index

The filtered index is also a non-clustered index, but here you can specify the filter condition clause) in the index definition. You can leverage filtered index when you need to query a specific set of records and wish to store the index data only for the relevant dataset.

In case you create a normal non-clustered index, it applies to the entire table, but with the filtered index you can limit it to the relevant dataset with the help of a filter clause. Indexes offer performance benefits but at the cost of additional disk space and few other overheads such as more time taken during write operations, and so on, as we already discussed. The filtered index is the best example of utilizing a feature by maintaining a balance in such a way that it benefits us but with less overhead.

The syntax to create non-clustered filtered index is as follows:

```
CREATE NONCLUSTERED INDEX name>  
ON
```

```
name> (1 name> 2 name> .... n name>  
INCLUDE (a name>, b name>, .... z name>)  
WHERE name>  
AND name>
```

The explanation of the syntax to create non-clustered filtered index is as follows:

If you'll compare the syntax of the filtered index with covering index then you will observe only one difference and that is an additional **WHERE** clause in the index definition. The filtered index can be also without an **INCLUDE** clause. It means a filtered index may or may not be a covering index:

CREATE NONCLUSTERED INDEX is the keyword to be specified to create a non-clustered index.

name> to be replaced with the name of index. You have the liberty to assign the name as you wish. However, keeping it as per the naming convention best practices is always a good option.

name> to be replaced with the actual table name on which the non-clustered index is to be created.

(1 name>, 2 name>, ... n name>) is called **non-clustered index key** and each column in the key is called **non-clustered index key**. Each column in the key has to be assigned the sort order. If the sort order is not defined then the default ascending sort order will be considered by SQL Server. When you'll create the non-clustered index then you need to specify the name of columns on which you desire to create the non-clustered index.

INCLUDE is used to make the non-clustered index a covering index by including the additional columns, other than ones used in the index key.

(a name>, b name>, ... z name>) is called **covering columns** or **included** Columns you wish to cover or include have to be specified here. You can't specify the column that has been already used in index key.

WHERE is the keyword used to specify the filter condition to make the non-clustered index a filtered index.

AND can be used to combine multiple filter conditions.

You can read more on filtered indexes at the following URL of Microsoft official documentation:

<https://docs.microsoft.com/en-us/sql/relational-databases/indexes/create-filtered-indexes?view=sql-server-ver15>

Unique index

When you create a clustered or non-clustered index, it does not enforce the uniqueness. It means the index key columns can hold duplicate values. However, if you wish to enforce the uniqueness on the index key columns, then SQL Server enables us to tag the index as **UNIQUE**. It makes the index, be it clustered or non-clustered, a unique index. **UNIQUE** keyword with a clustered index makes a unique clustered index. **UNIQUE** keyword with a non-clustered index makes a unique non-clustered index.

You can even create a unique index without clustered or non-clustered specified to it. Such indexes will be created as a unique non-clustered index.

The syntax to create unique index is as follows:

```
CREATE UNIQUE INDEX name>  
ON
```

```
name> (1 name> 2 name> .... n name>
```

The syntax is almost same to what we have discussed so far for clustered and non-clustered indexes. There is only one difference that differentiates other indexes with unique index and that's the **UNIQUE** keyword. Specify this keyword to make an index the unique index.

You can read more on unique indexes at the following URL of Microsoft official documentation:

<https://docs.microsoft.com/en-us/sql/relational-databases/indexes/create-unique-indexes?view=sql-server-ver15>

Columnstore indexes

Columnstore indexes are recent offerings of SQL Server and were introduced in Microsoft SQL Server 2012. They are primarily introduced to improve the performance of the OLAP (data warehouse) workload.

Columnstore indexes as opposite to the rowstore indexes, stores the data in columnar format. The data stored in columnstore indexes are highly compressed. Data compression can be achieved up to *10 times* by using the columnstore indexes. Data warehouse workload queries can perform better with up to 10 times the performance benefits over the traditional rowstore indexes.

Compression plays an important role. Since data are highly compressed, hence more data can be accommodated in a page, as compared to rowstore indexes. More data in a page means more pages can be retained in memory. More pages in the memory mean less calls to disk that means less IO. Less IO will always result in a better performance.

Columnstore indexes are not meant for locating the data quickly, as rowstore indexes are used. They can outperform in the case of analytical workloads. They can also be used in OLTP workload for analytical queries such as and so on.

Rowstore indexes have a specific purpose, so does the columnstore indexes have. We cannot form an opinion, just because one outperforms in some specific scenario. Both of these types of indexes can be also used together in a table.

Clustered columnstore index

The concept and purpose of clustered columnstore index are no different than the clustered index. Clustered columnstore indexes too maintain the second copy of the table, as clustered indexes do. Obviously, the storage internals and their behavior with database engines are different.

Although you can create both columnstore and rowstore indexes on a table. However, you can't create a clustered columnstore Index if the table already has a clustered index. You can create only one clustered index whether it is columnstore or rowstore. You cannot create both.

There is one major difference as far as T-SQL script is concerned to create it. In rowstore clustered index, we need to specify the index key in the index definition, whereas we need not in clustered columnstore index.

You would be wondering why?

As far as I can relate, it's because rowstore index stores the data row-wise hence it needs an identification for each row. Whereas, columnstore index stores the data column-wise hence it does not care to have a row identification.

The syntax to create clustered columnstore index is as follows:

```
CREATE CLUSTERED COLUMNSTORE INDEX name>  
ON
```

name>

The syntax to create the clustered columnstore index is very simple, as compared to clustered index (rowstore). You just need to specify the index name and table name and *the job is done!*

Non-clustered columnstore index

Non-clustered columnstore indexes too came into existence for the same purpose as the non-clustered index. Indexes, be it columnstore or rowstore, exists to make the queries run faster. But *which query*? This is something the differentiating factor for their applications. So far, we understood, rowstore indexes are meant for OLTP to locate the data quickly, and columnstore indexes are meant for OLAP (analytical queries). Columnstore indexes give the best results in the case of analytical queries on the huge number of rows. You would not observe the performance difference with less number of rows. But the difference can be observed as the data grows.

Although, it'll be repetitive as we already discussed in clustered columnstore index. Still, it's worth mentioning, storage architecture of the non-clustered columnstore index and non-clustered index (rowstore), and their behavior with the database engine is different.

The syntax to create non-clustered columnstore index is as follows:

```
CREATE NONCLUSTERED COLUMNSTORE INDEX name>  
ON
```

```
name> (1 name>, 2 name>, .... n name>)
```


If you compare the syntax of non-clustered columnstore index with non-clustered index (rowstore) then it is almost the same, with few differences as follows:

It cannot have more than 1024 columns. Technically speaking, it's not logical too to have these many columns in a non-clustered columnstore index.

You cannot create it as a constraint-based index. It is possible to have unique constraints, primary key constraints, and foreign key constraints on a table with a columnstore index. Constraints are always enforced with a row-store index. Constraints cannot be enforced with a columnstore (clustered or non-clustered) index.

You cannot modify it using the **ALTER INDEX** statement. To change the non-clustered index, you must drop and re-create the columnstore index instead.

INCLUDE keyword cannot be used in non-clustered columnstore index, as it can be used with non-clustered index (rowstore) to make it a covering index. In short, you cannot create the covering index with the non-clustered columnstore index.

ASC or **DESC** keywords cannot be specified on the index columns for sorting the index. Columnstore indexes are ordered according to the compression algorithms.

As we already discussed, you cannot create the covering index with the non-clustered columnstore index, however, it is very much possible to create a filtered index.

Filtered columnstore index

Filtered columnstore index too is similar to that of filtered rowstore index. You need to specify the filter condition clause) in the index definition to make it a filtered columnstore index. The rest of the explanations would be the same as what we have already discussed in the filtered rowstore index.

The syntax to create filtered columnstore index is as follows:

```
CREATE NONCLUSTERED COLUMNSTORE INDEX name>  
ON
```

```
name> (1 name>, 2 name>, .... n name>)
```

```
WHERE name>
```

```
AND name>
```

The syntax doesn't require an explanation. It's almost similar to the filtered rowstore index. Limitations and differences we've talked about in the non-clustered columnstore index would apply here as well. Always remember, filtered indexes are always non-clustered, be it rowstore or columnstore.

For further reading on columnstore indexes, you can refer to the following URL of Microsoft official documentation:

<https://docs.microsoft.com/en-us/sql/relational-databases/indexes/columnstore-indexes-overview?view=sql-server-ver15>

DROP INDEX

We have talked about various kinds of indexes and have also learnt to create them. You must also be interested to know *how to drop them?* You would be amazed to know that there is a common syntax to drop all kinds of indexes. The syntax is as mentioned as follows:

DROP INDEX name> **ON**

name>

The syntax is self-explanatory and does not need an explanation. Still would explain it a bit:

DROP INDEX is the keyword to instruct SQL Server to drop an index.

name> has to be replaced with the actual name of the index you wish to delete.

ON is the keyword to specify the table name on which the index exists.

name> has to be replaced with the actual table name on which the index exists.

Although, SQL Server offers ALTER INDEX to modify the indexes, but it has multiple limitations. That's why ALTER INDEX is not

used widely. If you wish to modify an index then you can drop and then create it. It will have the same effect and this approach doesn't have any limitations too. This approach can be used with all types of indexes.

Indexes in practical

It is a very vast subject and it cannot be covered in a topic or a chapter. The book does not intent to cover the indexing strategies and performance tuning. It'll be a separate book altogether. However, I'll make an attempt to make you understand how the indexes work and how to identify the candidate for an index.

The candidates for an index are hidden in the requirement itself. For example, you get the requirement *Get me the customer details of the provided Customer **Customer ID*** becomes our search criteria. Similarly, you get the requirement, *Get me the details of all the loan defaulters of the month of Jul 21 from the Mumbai and Branch* becomes our search criteria. Indexes are meant for the search, and depend upon the search criteria. *Customer ID* of the first example and *Month* and *Branch* of the second example can become the index candidate.

Let's consider we have a table similar to shown as follows for *Purchase* In the following example, we've considered just five records but the actual may can have millions and billions of rows:

rows:
rows:

rows:
rows:
rows:
rows:

Table 3.1: *PurchaseOrder table*

Suppose there is a requirement to fetch and **Amount** by The query will look like as follows:

```
SELECT OrderID, TransactionDate, ProductID, Amount
FROM PurchaseOrder
WHERE CustomerID = 1
```

This is an example where we need to locate the row quickly. So, we can make use of rowstores indexes, specifically a non-clustered index on Non-clustered index is preferred on the columns having duplicate values, or having foreign key references. However, it is to be noted that more the duplicate values the index columns will have, the lesser will be the performance benefits.

You can also choose a non-clustered covering index on **CustomerID** by covering all other non-index key columns of the **SELECT** statement in the **INCLUDE** clause. Non-index key column means the column that is already part of the index key (for example, **CustomerID** in this case), and clustered

index key columns. Every non-clustered index holds the pointer to the clustered index, so you need not to include such columns in the covering index.

Covering non-clustered index will outperform the simple non-clustered index. But it will have some overhead too. The write operations (insert, update, and delete) on the table would involve additional work of maintaining the additional columns data in the index, and hence take more time to complete. More columns in the index, more will be index size, and higher will be the index page count. Index maintenance would also require more time for covering the non-clustered index as compared to the simple non-clustered index.

If the filter condition is changed from the **CustomerID** to **OrderID** as follows then clustered index will be the best fit. You may want to create a clustered index on **OrderID** column. This index will outperform all other indexes if the filter is specified on the **OrderID** column:

```
SELECT OrderID, CustomerID, TransactionDate, ProductID,  
Amount  
FROM PurchaseOrder  
WHERE OrderID = 1
```

The fraction of rows in a table having the same values for the index key defines the index selectivity. A highly selective index has few rows for each index entry. A highly selective index represents good cardinality. Whereas, a low selective index represents poor cardinality.

The columns chosen for the index keys should be of the good cardinality to make an index highly selective. If the index selectivity or cardinality is poor then there won't be any performance benefits of having the index in-place.

There are other factors that matter while defining an index. They matter because it matters to SQL Server Database Engine and Optimizer to decide *whether to use an index?* If then *how?*

Always remember, the following points play a crucial role in defining the index and at the same time to help SQL Server choose it. They are as follows:

Filter condition and/or join conditions.

The sequence of columns in the filter and/or join conditions.

Column list in the

Type of operations such as simple **SELECT** or analytical operations and so on).

The first two points that pertain to filter and/or join conditions, relate to the index key. The index key is the columns on which the index has been created. So, the columns in the index and the sequence in which they are

placed have to be the same as the filter and/or join condition. If this satisfies then SQL Server chooses to seek the index called **index seek** operation. Otherwise, it'll will scan the index called **index scan** operation.

Index seek could be on a clustered index or on the non-clustered index. If it's a clustered index seek then *it's always good news!* A clustered index doesn't have to perform bookmark lookup to locate the data entries, since it also has a copy. But if it's a non-clustered index then there will be an additional step of bookmark lookup – key lookup (if table has clustered index) or RID lookup (if table is a heap table without clustered index). Now, you would be wondering *why and when bookmark lookup happens?*

Bookmark lookup is always related to the third point that is the column list in the These columns are not part of the index definition so the non-clustered index has to perform additional steps to locate the data entries. Bookmark lookup will be only performed for the columns that are not part of the index definition. This is where the covering index plays the role. You include the missing columns in the index definition and the index becomes the covering index. Now there won't be bookmark lookups – key and RID lookups.

Similarly, the filtered index is also dependent on the filter condition of the query.

So far, we've talked about the rowstore indexes. Now, let's understand when and where columnstore indexes are used.

Refer [table 3.1](#) and suppose the requirement is to fetch the sum of amount and average of orders for all the orders:

```
SELECT AS TotalAmount  
, AS AvgAmount  
FROM PurchaseOrder
```

This is one of the possible use-case where you can create the non-clustered columnstore index on the **Amount** column. This is a very simple scenario, whereas the situation in reality could be much more complicated.

Conclusion

Indexes are the performance booster of the T-SQL queries. You need them when you are working on production databases. If the data size is huge then indexes become savior to honor the data request on time. T-SQL queries are not only executed from SSMS but can be triggered from the application such as web app, mobile app, windows app, and so on too.

Would you be interested to wait for minutes for your mobile app to respond you? Seriously, We do not have patience and time today. We need everything quickly. How many times have you waited for any website which is just loading and loading? Yes, you got it right. Indexes make your queries run faster which in turn helps the application run faster.

In the next chapter, we'll understand the various DML statements in T-SQL, and also learn to implement them.

Points to remember

Clustered indexes both rowstore and columnstore maintain another copy of the table.

You cannot create more than one clustered index on a table.

Table without a clustered index is called heap table.

Bookmark lookups are of two types – **key** and **RID** lookup.

Key lookup occurs on a table with a clustered index.

RID lookup occurs on a table without a clustered index.

Bookmark lookups are performance killers and can be avoided with covering index.

Rowstore indexes are meant for locating the data quickly on OLTP workload, whereas columnstore indexes are meant for analytical workload for performing quicker analytics.

Clustered columnstore index does not have an index key as opposite to the clustered index (rowstore).

You cannot specify the sort order in the columns of the non-clustered columnstore index.

You cannot create covering index with columnstore indexes. It can be only created with rowstore indexes.

Both rowstore and columnstore indexes support the filtered index.

Multiple choice questions

When key lookup does occur?

With index scan.

If there is no clustered index.

With index seek and if the table has clustered index.

If the index key does not match with the filter predicates of the query.

Which index would be the best fit for the analytical queries on tables having a large set of data?

Clustered index.

Columnstore indexes.

Filtered index.

Covering index.

Answers

C

B

Questions

When you should create a filtered index?

What is bookmark lookup and how to avoid it?

What is a heap table?

Can you create columnstore covering index?

What is RID lookup and when it occurs?

What cardinality?

When the index is said to have good cardinality?

Why you cannot create more than one clustered index on a table?

Key terms

Heap table is a table without a clustered index.

Rowstore is a variety of index in which the data is stored row-wise. This type of indexes has an index key. It is generally divided into clustered and non-clustered indexes. However, the non-clustered indexes have further classification such as covering and filtered indexes. A non-clustered index that is neither covering nor filtered is termed simply as a non-clustered index.

Columnstore is another variety of index in which is the data is stored column-wise. There are two types of columnstore indexes – clustered and non-clustered. However, non-clustered index can also be created as a filtered columnstore index.

Bookmark lookups occur in the case of rowstore indexes and that too with a non-clustered index. It is the process by which the data entries are located. Each non-clustered index stores the clustered index key or the reference of the heap table to locate the data entries.

Key lookup is a kind of bookmark lookup that locates the data entries from the clustered index.

RID lookup is also a kind of bookmark lookup that locates the data entries from the heap table.

CHAPTER 4

DML

The database is just a placeholder like a cupboard. It is nothing if it doesn't have any data. Imagine, you bought a branded, spacious and costly cupboard and kept it somewhere in your house and never used it. *Does it make sense to have an empty cupboard?*

We are sure the answer would be No unless collecting cupboards is your hobby. We buy a cupboard so that we can store and retrieve the articles of our interest. Similarly, in the case of databases, it becomes necessary to have a mechanism to add, modify, delete, and most importantly read data already stored. DML serves this purpose. DML stands for **Data Manipulation**

We understood DML in [Chapter 1, Getting](#) We learnt to create the table in [Chapter 2,](#) Now, in this chapter, we'll further deep dive into DML and learn various kinds of DML statements.

Structure

In this chapter, we will cover the following DML statements:

INSERT statement

SELECT statement

UPDATE statement

DELETE statement

Magic tables (inserted and deleted tables)

Objective

After studying this chapter, you'll be able to insert, update, and delete records as well as to read records using You'll be able to filter the records with the help of where clause. You can implement the paging in the dataset.

INSERT statement

INSERT statement is used to create a new record. When you'll create a table, the intention will be to have the data in the table. You can add records only with the help of

Since we are going to learn various DML statements hence we need to have few tables that we can use across this chapter and other upcoming chapters. We'll create tables similar to what we discussed in [Chapter 2, Tables](#) in *tables*

Remember, we created the **Learn_T_SQL** database in [Chapter 2](#). We'll refer to the same database across the book. We'll create three tables namely: and

Following is a DDL script to create a table named **Customer** in the chosen database. You need to select the **Learn_T_SQL** database in the query window, and the table will be created in it when you'll execute the query as follows:

```
CREATE TABLE Customer
(
CustomerID    INT NOT NULL
, NOT NULL
, NOT NULL
, NOT NULL
, CONSTRAINT PK_Customer_CustomerID PRIMARY KEY
```


)

Following is a DDL script to create the table named

```
CREATE TABLE Product
(
ProductID    INT NOT NULL

, NOT NULL
, CONSTRAINT PK_Product_ProductID PRIMARY KEY
)
```

Following is a DDL script to create the table named

```
CREATE TABLE PurchaseOrder
(
OrderID    INT NOT NULL
, NOT NULL
, NOT NULL
, ProductID    INT NOT NULL
, Quantity    INT NOT NULL
, Rate    NOT NULL
, Amount    NOT NULL
, CONSTRAINT PK_PurchaseOrder_OrderID PRIMARY KEY
, CONSTRAINT FK_PurchaseOrder_CustomerID FOREIGN KEY
REFERENCES Customer
, CONSTRAINT FK_PurchaseOrder_ProductID FOREIGN KEY
REFERENCES Product
)
```

Once all the three tables are created, it can be seen in Object Explorer as shown in the next screenshot:

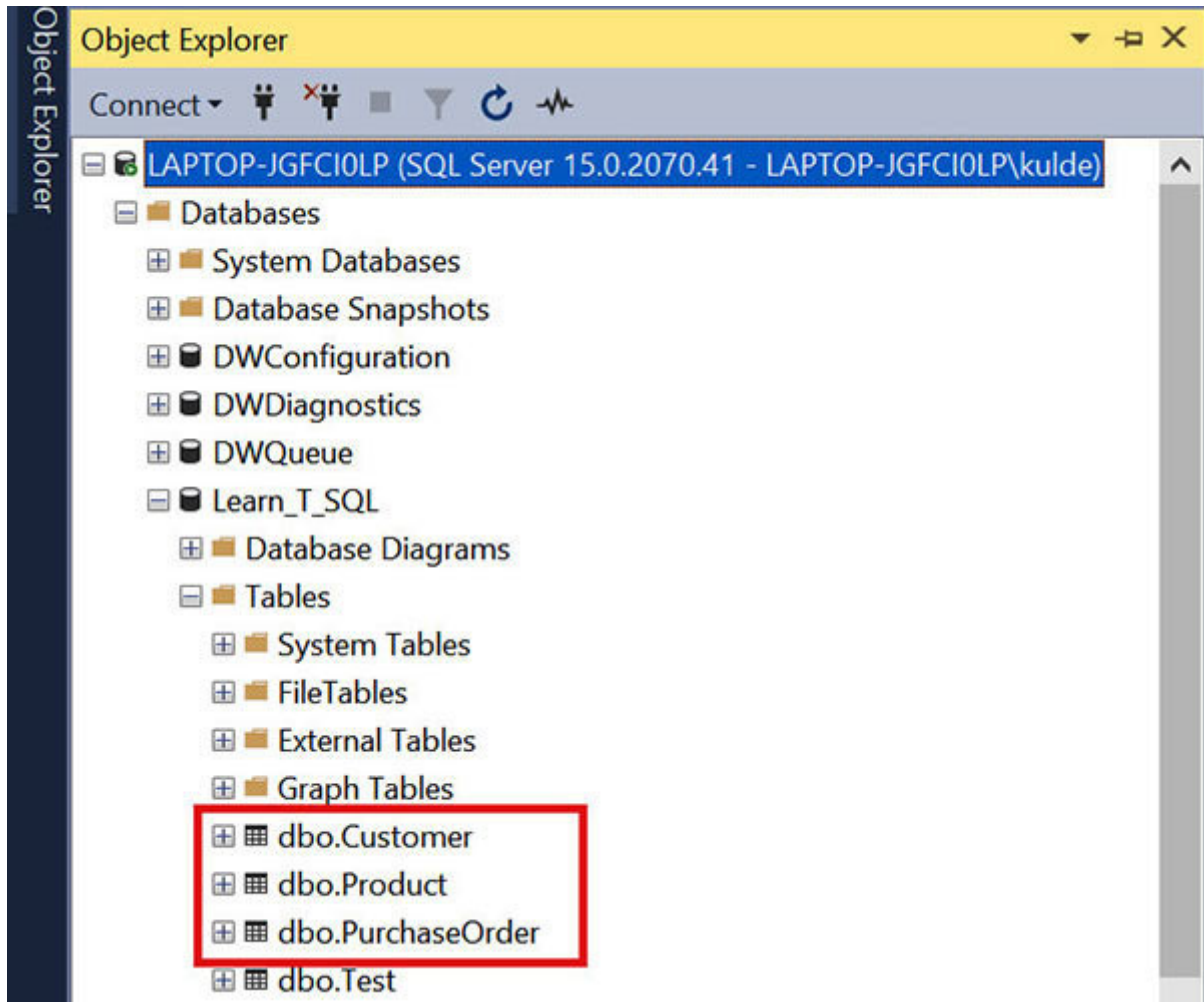


Figure 4.1: Tables reflecting in Object Explorer

Syntax of **INSERT** command is as follows:

INSERT INTO

name> (1 name>, 2 name>, ..., n name>)

VALUES (for column 1>, for column 2>, ..., for column n >)

Explanation of the syntax of **INSERT** command is as follows:

INSERT INTO is the keyword that is an instruction to insert a new record in the table.

name> has to be replaced with the actual table name.

name> has to be replaced with the actual column names.

VALUES is the keyword used to supply the values to each column as specified in the **INSERT**. If you'll observe then **VALUES** has the exactly the same number of values as the number of columns specified in **INSERT**.

Except for Numeric and Bit values, all other values should be supplied within single quotes ("). For example, and so on. You can insert multiple records by supplying multiple sets of values separated by a comma as can be seen in the following

Here is an example of **INSERT** statement. This script will insert three records in the **Customer** table:

```
INSERT INTO Customer
```

```
VALUES
```

```
,
```

```
,
```

SELECT statement

The **SELECT** statement is used to retrieve the records from the table. If you want to view the records stored in a table then you need to write a **SELECT** statement. This is the command used to read the data already stored in the table. **SELECT** statements come in various flavors. We'll discuss each of them individually in the next section.

SELECT *

SELECT * statement represents retrieve all columns.

SELECT * FROM

name>

Explanation of the syntax of **SELECT *** command is as follows:

SELECT is the keyword that is an instruction to retrieve the records.

* represents all columns.

FROM is the keyword that is used to supply the table name from which the records are to be retrieved.

name> has to be replaced with the actual table name.

The following script will return all the rows and columns from the **Customer** table.

```
SELECT * FROM Customer
```

The output of the query will be similar to as shown in the following image:

	CustomerID	CustomerName	CustomerAddress	CustomerMobile
1	1	John	Paris	1111111111
2	2	Kiya	London	2222222222
3	3	Reema	India	3333333333

Figure 4.2: Output of *SELECT ** script on Customer table

SELECT-specific columns

The **SELECT** statement can also be used to retrieve specific columns. Let us understand *how to do it?*

```
SELECT 1 name>  
, 2 name>  
, .  
.   
.   
, n name>  
FROM
```

name>

Explanation of the syntax of **SELECT** specific columns command is as follows:

SELECT is the keyword that is an instruction to retrieve the records.

name> has to be replaced with actual column names.

FROM is the keyword that is used to supply the table name from which the records are to be retrieved.

name> has to be replaced with the actual table name.

Example of **SELECT** specific columns command:

The following script will return all the rows of and **CustomerMobile** columns from the **Customer** table. If you'll notice the output, then it is exactly the same as you can see in [figure](#). This is because the **Customer** table has only four columns - and

SELECT CustomerID

```
, CustomerName  
, CustomerAddress  
, CustomerMobile  
FROM Customer
```

The following screenshot shows the output of the query:

	CustomerID	CustomerName	CustomerAddress	CustomerMobile
1	1	John	Paris	1111111111
2	2	Kiya	London	2222222222
3	3	Reema	India	3333333333

Figure 4.3: Output of *SELECT* specific columns on Customer table

If you've additional rows in your table then the output of the **SELECT** queries may differ in your case. If the tables have exactly the same rows as discussed in this book, only then you'll get the similar results as this book.

Filtering

Data filtering is a process to filter out rows based on certain conditions on the set of columns. It is done by applying filters to the columns. Filters could be an equality, or an inequality condition, or any combination thereof, on the set of columns.

Suppose we've the following table. The table could have N number of rows, whereas we have only five sample rows in the following table. You were asked to fetch only those customers who are not based in **Maharashtra** state and have **CustomerID** greater than In this scenario, there are two filter conditions – **State** should not be equal to and **CustomerID** is greater than Both of these filter conditions are inequality conditions:

conditions:
conditions:
conditions:
conditions: conditions:
conditions:
conditions: conditions:

Table 4.1: *Customer table*

In order to understand the equality conditions, let us assume you are asked to fetch only those customers who belong to **Maharashtra** state. In this scenario, there is only one filter condition, and that too is the equality condition. This is a general explanation of the filtering. However, every tool has its own way to apply the filtering. Let us understand how you can filter the data in T-SQL.

The **WHERE** command is used to filter the records in T-SQL. When it is used with **SELECT** will retrieve only matching records as per the condition(s) specified. When it is used with **UPDATE** will modify the matching records only, as per the condition(s) specified. Similarly, when it is used with **DELETE** will delete the matching records only, as per the condition(s) specified. We'll talk about **UPDATE** and **DELETE** more in detail in the upcoming topics of this chapter.

```
SELECT 1 name>  
, 2 name>  
, .  
.   
.   
, n name>  
FROM
```

```
name>  
WHERE name>
```

/ OR> name>

OR

```
SELECT *  
FROM  
name>  
WHERE name>  
/ OR> name>
```

The explanation is as follows:

SELECT is the keyword which is an instruction to retrieve the records.

name> has to be replaced with actual column names.

FROM is the keyword that is used to supply the table name from which the records are to be retrieved.

name> has to be replaced with the actual table name.

WHERE is the keyword used to filter the data.

are the operators to combine multiple filter conditions.

Following operators can be used with **WHERE** command:

=: Is equality operator, used for checking the equality. It can be used with all the data types. For example, **WHERE CustomerID =**

Both have the same meaning and refers to *not equals* It can be used with all the data types. For example, **WHERE CustomerID <>**

<: Refers to *less* It can be used with all the data types (except string data types). For example, **WHERE CustomerID <**

Refers to *less than or equals* It can be used with all the data types (except string data types). For example, **WHERE CustomerID <=**

>: Refers to *greater* It can be used with all the data types (except string data types). For example, **WHERE CustomerID >**

Refers to *greater than or equals* It can be used with all the data types (except string data types). For example, **WHERE CustomerID >=**

If you want to check a particular value that should match to any of the set of values then make use of this operator. The list of values is to be supplied in parenthesis It can be used with all the data types. For example, **WHERE CustomerID IN (1, 2,** Subqueries too can be used with this operator instead of a static list of values. We'll talk about subqueries more in detail in [Chapter 6, Join, Apply, and](#)

NOT If you want to check a particular value that should not match to any of the set of values then make use of this

operator. The list of values is to be supplied in parenthesis It can be used with all the data types. For example, **WHERE CustomerID NOT IN (2,** Subqueries too can be used with this operator instead of a static list of values. We'll talk about subqueries more in detail in [Chapter 6, Join, Apply, and](#)

It can be used only with String data types. It is used to determine whether a character string matches a specified pattern. A pattern can include regular characters and wildcard characters. Following are few examples with the **LIKE** operator. We'll talk about wildcard characters and wildcard search more in detail in the subsequent *Wildcard search*

WHERE CustomerName LIKE This is an example of an exact search. Here we've used the regular characters in the pattern. 'This means search all the rows where **CustomerName** is This is similar to the equality operator. **WHERE CustomerName LIKE 'ABC'** and **WHERE CustomerName = 'ABC'** will serve the same purpose.

WHERE CustomerName LIKE This is an example of a wildcard search. **ABC** are the regular characters and % is a wildcard character in the pattern. This means search all rows where **CustomerName** starts with

Is used for range search. A range requires boundary values that include lower and upper boundary. It can be used only with numeric and date and/or time data types:

WHERE CustomerID BETWEEN 1 AND 2 This is an example of a range search. Here **1** is the lower boundary and **2** is the upper boundary. This means search all the rows where **CustomerID** is between **1** and **2** and It will fetch all the records where **CustomerID** is greater than or equals to **1** and less than or equals to **2**

WHERE DateOfBirth BETWEEN '2000-01-01' AND '2002-12-31' This is an example of range search. Here, **2000-01-01** is the lower boundary and **2002-12-31** is the upper boundary. This means search all the rows where **DateOfBirth** is between **2000-01-01** and **2002-12-31** and It will fetch all the records where **DateOfBirth** is greater than or equals to **2000-01-01** and less than or equals to **2002-12-31**

WHERE condition can be applied on multiple columns with the help of and **OR** operators.

For example:

WHERE CustomerID = 1 AND CustomerAddress = 'India'

This means search all rows where **CustomerID** is **1** and also **CustomerAddress** is **'India'** If either condition will fail for any row, that row won't be returned.

WHERE CustomerID = 1 OR CustomerAddress = 'India'

This means search all rows where **CustomerID** is **1** or **CustomerAddress** is **'India'** If either condition will succeed for any row, that row will be returned.

row, that row will be returned.

WHERE condition can be made up of multiple **AND** and/or **OR** conditions such as the following. Parenthesis is a must if there are more than one **AND** and/or

```
WHERE ((CustomerID = 1) OR (CustomerAddress = 'India'  
AND CustomerName LIKE 'Ree%'))
```

The data type of both sides of the operator used in the **WHERE** clause should be compatible with each other. For example, if the **CustomerID** column is of type numeric, and the **WHERE** clause is specified as **CustomerID =** The query with such a **WHERE** clause will fail with an error due to incompatible data types. If you'll notice the left part of the equality operator is of type numeric, whereas the right part of the operator is of string type. Both of them are clearly not compatible. This rule applies to all kinds of operators that can be used with the **WHERE** clause.

The following is an example of where clause on a numeric column with equality operator:

```
SELECT *  
FROM Customer  
WHERE CustomerID = 1
```

The following screenshot shows the output of the query:

	CustomerID	CustomerName	CustomerAddress	CustomerMobile
1	1	John	Paris	1111111111

Figure 4.4: Output from Customer table for CustomerID = 1

Following is an example of where clause on a string column with equality operator:

```
SELECT *
FROM Customer
WHERE CustomerAddress = 'London'
```

The following screenshot shows the output of the query:

	CustomerID	CustomerName	CustomerAddress	CustomerMobile
1	2	Kiya	London	2222222222

Figure 4.5: Output from Customer table for CustomerAddress = 'London'

Wildcard search

The wildcard search is a kind of filter condition where instead of filtering a column-based on exact value, you may want to search for specific characters. It could be starting few characters, ending few characters, or in between few characters, or a combination of all these. Wildcard search is primarily meant for string data.

Wildcard search also comes in two variants – and **LIKE** operator is used for wildcard search in T-SQL. **LIKE** is used for equality condition, and **NOT LIKE** is used for inequality condition.

As we discussed earlier **LIKE** operator determines whether a character string matches a specified pattern. A pattern can include regular characters and wildcard characters. There are various kinds of wildcard characters such as _ (underscore), and Each of them has a specific purpose described as follows:

%: represents any string of zero or more characters. It is used to determine if a character string matches the specified character(s):

Pattern **A%** represents string starting with the character

Pattern **%A** represents string ending with the character

Pattern **%A%** represents string having character **A** anywhere in the string.

_(underscore): Represents any single character. It is used to determine if a character string matches the specified regular character(s) at the specified position:

Pattern **_A%** represents string starting with two characters, out of which the second character is A. For example, and so on.

Pattern **%_A_T** represents string ending with four characters, out of which the last character is T and the third last character is For example, and so on.

Pattern **%_A_%CAKE** represents a string having three consecutive characters anywhere in the string, out of which the middle character is character and the ending four characters are For example,

Represents any single character within the specified range (for example, or set (for example, It is used to determine if a character string contains any of the characters as specified in the range or set.

Pattern **[B-D]%MAN** represents string starting with any character in the range **B-D** (possible characters can be **B** or **D** only), and ending with three characters For example,

Pattern **[BSHI]%MAN** represents string starting with any character in the specified set **BSHI** (possible characters can be **B** or **I** only), and ending with three characters For example, and so on.

Represents any single character not within the specified range (for example, or set (for example, It is used to determine if a character string does contain any of the character as specified in the range or set:

Pattern **[^B-D]%MAN** represents string starting with any character which should not be in the range **B-D** (characters other than **B** and **D** and ending with three characters For example, and so on.

Pattern **[^BSHI]%MAN** represents string starting with any character other than ones specified in the set **BSHI** (characters other than **B** and **I** and ending with three characters For example, and so on.

Let us see the comprehensive syntax of the **LIKE** operator:

```
match_expression [NOT] LIKE pattern [ESCAPE  
escape_character]
```

Where:

match_expression can be a column name, subquery, variable name, and so on specified at the left side of the **LIKE** operator in a **WHERE** clause or a **JOIN** condition, and so on. We'll talk about joins and subqueries further in detail in [Chapter 6, Join, Apply, and](#)

NOT is optional and can be used for inequality conditions.

LIKE is the operator used for the wildcard search we have already discussed.

pattern is the pattern to be used for wildcard search with or without wildcard characters.

ESCAPE is the command to use the escape character. It is optional and can be used if any wildcard character is to be treated as a regular character in the pattern specified. For example, if you wish to use the % in the pattern as a search criterion. As we already discussed % is a wildcard character. Suppose we have two strings as follows, and we want to find all the strings that have 50% anywhere in the string:

The amount is 50% of the

Age of Jack is 50 years.

If we'll define the pattern as **%50%%** then we'll not get the desired result, and we may get both of the preceding mentioned strings. Whereas, we wish to have only the first string **The amount is 50% of the** The second string **Age of Jack is 50 years** has **50** but does not have the **%** sign after **The** **LIKE** operator will not give the refined result even though we have specified **50%** in the pattern. This is where the **ESCAPE** clause and escape characters play an important role.

escape_character is a single character put before the wildcard character (part of the search criteria) to indicate that the wildcard character is to be interpreted as a regular character. It is optional. You need to specify the escape character only if you have specified the **ESCAPE** clause. You can use any character which should not clash with the wildcard characters and the characters used in your search pattern. Generally, **!** is used as an escape character. **LIKE '%50!%%'** **ESCAPE** is an example with the **ESCAPE** clause and escape character, based on the strings mentioned in the aforesaid example.

We have talked about the **LIKE** operator, wildcard search, various wildcard characters, and different kinds of search patterns. Let us understand few use-cases of the most-used wildcard character **%** in the context of T-SQL queries. You can practice the remaining wildcards by referring to the various scenarios and examples of the patterns we've discussed.

Following is an example of where clause on a string column with **LIKE** operator for wildcard search (searching starting few characters). The following query will fetch all the columns of

the **Customer** table, for only those rows whose **CustomerAddress** starts with The starting characters in the **CustomerAddress** should be **Lon** in order to satisfy the filter condition.

```
SELECT *  
FROM Customer  
WHERE CustomerAddress LIKE 'Lon%'
```

The following screenshot shows the output of the query:

	CustomerID	CustomerName	CustomerAddress	CustomerMobile
1	2	Kiya	London	2222222222

Figure 4.6: Output from Customer table where CustomerAddress starting with 'Lon'

Following is the example of where clause on a string column with **LIKE** operator for wildcard search (searching ending few characters). The following query will fetch all the columns of the **Customer** table, for only those rows whose **CustomerAddress** ends with The ending characters in the **CustomerAddress** should be **don** in order to satisfy the filter condition.

```
SELECT *  
FROM Customer  
WHERE CustomerAddress LIKE '%don'
```

The following screenshot shows the output of the query:

	CustomerID	CustomerName	CustomerAddress	CustomerMobile
1	2	Kiya	London	2222222222

Figure 4.7: Output from Customer table where CustomerAddress ending with 'don'

Following is the example of where clause on a string column with **LIKE** operator for wildcard search (searching middle few characters). The following query will fetch all the columns of the **Customer** table, for only those rows whose **CustomerAddress** have **ond** characters together somewhere in the string. It could be starting three characters, ending three characters, or could be at any place, but all three characters should be together in the string:

```
SELECT *  
FROM Customer  
WHERE CustomerAddress LIKE '%ond%'
```

The following screenshot shows the output of the query:

	CustomerID	CustomerName	CustomerAddress	CustomerMobile
1	2	Kiya	London	2222222222

Figure 4.8: Output from Customer table where CustomerAddress contains 'ond'

Following is the example of where clause on a string column with **LIKE** operator for comprehensive wildcard search (including multiple criteria such as starting characters, middle characters and ending characters). The following query will fetch all the columns of the **Customer** table, for only those rows whose **CustomerAddress** have the first character as the last character as and three characters **ond** somewhere in the middle of the string, but all these three characters should be together:

```
SELECT *  
FROM Customer  
WHERE CustomerAddress LIKE 'L%ond%n'
```

The following screenshot shows the output of the query:

	CustomerID	CustomerName	CustomerAddress	CustomerMobile
1	2	Kiya	London	2222222222

Figure 4.9: Output from Customer table where CustomerAddress starts with 'L', contains 'ond', and ends with 'n'

You can refer to the following URL of Microsoft official documentation to read more about the LIKE operator:

<https://docs.microsoft.com/en-us/sql/t-sql/language-elements/like-transact-sql?view=sql-server-ver15>

Range search

Range search is a kind of data filtering. Here instead of searching for a specific value, the search is performed on a range. A range consists of the lower and upper boundaries. Range search is applicable only to numeric and date-time values.

For example, you want to search all the customers with **CustomerID** falling between **1** and **5** is the range here. **1** is the lower boundary and **5** is the upper boundary. All the customers will be fetched wherever the **CustomerID** is greater-than or equal to **1** (lower boundary) and less than or equal to **5** (upper boundary).

Similarly, you may want to search all the customers who got onboarded between **Jan'01 20** to **Dec'31 20** is the range here. **Jan'01 20** is the lower boundary and **Dec'31 20** is the upper boundary.

You can perform a range search in T-SQL using **BETWEEN** operator. Following is an example of where clause on a numeric column with **BETWEEN** operator for range search. The following query will fetch all the customers whose **CustomerID** falls between **1** and

SELECT *

FROM Customer

WHERE CustomerID BETWEEN 1 AND 2

The following screenshot is of the output of the query:

	CustomerID	CustomerName	CustomerAddress	CustomerMobile
1	2	Kiya	London	2222222222

Figure 4.10: Output from Customer table where CustomerID is between 1 and 2

Range search can be applied only on Numeric and Date/Time data types.

Sorting

Sorting is the process to sort the data in either ascending or descending order of the column’s values. The sorting can be done on single column or multiple columns. Suppose you have the following table:

table:
table:
table:
table: table:
table:
table: table:

Table 4.2: Customer table with unsorted data

Table 4.2 has unsorted data. If you’ll sort it by **CustomerID** in ascending order. The resulting sorted data will look like as can be seen in the following table:

table:
table:
table:

table: table:
table:
table: table:

Table 4.3: Customer table sorted by CustomerID in ascending order

Similarly, if you'll sort [table 4.2](#) by **CustomerID** in descending order. The resulting sorted data will look like as can be seen in the following table:

table:
table: table:
table:
table: table:
table:
table:

Table 4.4: Customer table sorted by CustomerID in descending order

ORDER BY clause is used in T-SQL to sort the result set. Two sort orders can be applied with the ORDER BY clause. These sort orders are:

Used for ascending order.

Used for descending order.

You can sort the result set based on the multiple columns.
You can also define the sort order on each of these columns.

We'll see an example of each of these sort orders.

```
SELECT 1 name>  
, 2 name>  
, .  
.   
.   
, n name>  
FROM
```

name>

```
ORDER BY 1 name>  
, 2 name>
```

OR

```
SELECT *  
FROM
```

name>

```
ORDER BY 1 name>  
, 2 name>
```

Explanation of the syntax of **ORDER BY** clause:

SELECT is the keyword that is an instruction to retrieve the records.

name> has to be replaced with actual column names.

FROM is the keyword that is used to supply the table name from which the records are to be retrieved.

name> has to be replaced with actual table name.

ORDER BY is the keyword used to apply the sorting. It accepts a column on which sorting to be done. It is to be noted that the column mentioned in **ORDER BY** has to be there in the Any column not included in **SELECT** can't be added in **ORDER**

ASC is the keyword used to define the ascending sorting order on the column. It is the default sort order if not explicitly defined.

DESC is the keyword used to define the descending sorting order on the column.

Any column not included in SELECT can't be added in ORDER
Default sort order is ASC if not explicitly defined.

Following is the example of sorting with ascending sort order:

```
SELECT * FROM Customer
ORDER BY CustomerID ASC
```

The following screenshot shows the output of the query:

	CustomerID	CustomerName	CustomerAddress	CustomerMobile
1	1	John	Paris	1111111111
2	2	Kiya	London	2222222222
3	3	Reema	India	3333333333

Figure 4.11: Output from Customer table sorted by CustomerID in ascending order

Following is the example of sorting with descending sort order:

```
SELECT * FROM Customer
ORDER BY CustomerID DESC
```

The following screenshot shows the output of the query:

	CustomerID	CustomerName	CustomerAddress	CustomerMobile
1	3	Reema	India	3333333333
2	2	Kiya	London	2222222222
3	1	John	Paris	1111111111

Figure 4.12: Output from Customer table sorted by CustomerID in descending order

Each column in the ORDER BY clause can have the different sort order. The default sort order in case not specified is ASC (ascending).

Paging

The **OFFSET** and **FETCH** clauses are the advanced options of the **ORDER BY** clause. They are used to limit the number of rows to be returned by a query.

You can use these commands to implement the paging.

Paging means limiting the number of rows to be displayed in each run.

Paging is generally used in application programming to limit the number of rows being displayed to the user. So, if you get a requirement to implement the paging, you can use this feature.

Have you ever gone through a website such as your own bank website? If you would have noticed then all the records are not shown together unless you export them. The records are generally shown in the form of pages where you've the choice to choose how many records you would like to see in a page such as 50, 100, and so on. You've the flexibility to traverse through pages using and

We'll understand these commands in the context of paging itself. Paging has the following important components. I'll not explain these as they are self-explanatory.

Page size.

Page count.

Current page number.

If there are 55 records and if the page size is defined as 10 rows. Then there will be total of 6 pages. Each page will have 10 rows excluding the 6th page which will have 5 rows.

OFFSET can be understood as *'(Page number to be retrieved – 1) * Page*

If you want to read the first page then the offset should be set as

If you want to read the second page then the offset should be set as

If you want to read the third page then the offset should be set as **20** and so on.

FETCH command is used to fetch the specified number of top or next rows:

FETCH FIRST is an alternative of **TOP** command.

FETCH NEXT is used in conjunction with **OFFSET** to skip a specified number of rows and select the next specified number of rows.

```
SELECT 1 name>  
, 2 name>  
, .  
.   
.   
, n name>  
FROM
```

```
name>  
ORDER BY 1 |  
, 2 |  
OFFSET row |  
FETCH | row | ONLY
```

Explanation of the syntax of **OFFSET ... FETCH** command:

SELECT is the keyword which is an instruction to retrieve the records.

name> has to be replaced with actual column names.

FROM is the keyword that is used to supply the table name from which the records are to be retrieved.

name> has to be replaced with an actual table name.

ORDER BY is the keyword used to apply the sorting. It accepts a column on which sorting to be done. It is to be noted that the column mentioned in **ORDER BY** has to be there in the Any column not included in **SELECT** can't be added in **ORDER**

ASC is the keyword used to define the ascending sorting order on the column. It is the default sort order if not explicitly defined.

DESC is the keyword used to define the descending sorting order on the column.

OFFSET is the keyword used to skip the specified number of rows. In other words, it is used to set the pointer to select the rows after the specified number of rows. For example, if you want to read the record from 21st row onwards then **OFFSET** is to be set as

FETCH is the keyword to select the specified number of rows after

are the supporting keywords used with the **FETCH** command. Both of these have exactly the same functioning. If the purpose is to select the top specified number of rows, then use If the purpose is to skip a specified number of rows and select the next specified number of rows then use It would be confusing to use **FETCH FIRST** to read the next records. However, technically both serve the same purpose.

are the supporting keyword used with **OFFSET** and **FETCH** commands while specifying the number of rows. Both of these have exactly the same functioning and they serve the same purpose. You can use either of them and there should not be an issue. You can use **ROW** with **0** and **1** (singular) and for others, you can use **ROWS** (plurals).

Following is the simple example of **OFFSET** with **FETCH**

```
SELECT *  
FROM Customer  
ORDER BY CustomerAddress ASC  
OFFSET 0 ROW  
FETCH FIRST 1 ROW ONLY
```

The following screenshot shows the output of the query:

	CustomerID	CustomerName	CustomerAddress	CustomerMobile
1	3	Reema	India	3333333333

Figure 4.13: Fetching first row from the Customer table

Following is the simple example of **OFFSET** with **FETCH**

```
SELECT *  
FROM Customer  
ORDER BY CustomerAddress ASC  
OFFSET 1 ROW  
FETCH NEXT 1 ROW ONLY
```

The following screenshot shows the output of the query:

	CustomerID	CustomerName	CustomerAddress	CustomerMobile
1	2	Kiya	London	2222222222

Figure 4.14: *Fetching second row from the Customer table*

TOP

The **SELECT** statement can also be used with the **TOP** keyword to fetch the top specified number of records. You can specify the number of rows you would like to retrieve. The only catch here is that it will retrieve top specified number of records as it's stored in the table. However, you can also sort the result set with the help of the **ORDER BY** command. We'll understand each of these in detail.

TOP is mostly used in conjunction with the **ORDER BY** clause.

Before we deep dive on **SELECT** with it's recommended to go through the output of **SELECT** query as mentioned in aforesaid [*figure*](#)

SELECT TOP

```
1 name>  
, 2 name>  
, .  
.   
.   
, n name>
```

FROM

name>

Explanation of the syntax of **TOP** command:

SELECT is the keyword that is an instruction to retrieve the records.

TOP is the keyword that is used to specify the top number of rows to be retrieved.

has to be replaced with the actual number of rows to be retrieved.

name> has to be replaced with actual column names.

FROM is the keyword that is used to supply the table name from which the records are to be retrieved.

name> has to be replaced with the actual table name.

Following is the simple example which will fetch the top 1 record as it's stored in the table.

```
SELECT TOP 1 * FROM Customer
```

The following screenshot shows the output of the query:

	CustomerID	CustomerName	CustomerAddress	CustomerMobile
1	1	John	Paris	1111111111

Figure 4.15: Fetching top 1 row from the Customer table

Following is the evolved example which will fetch the top 1 record from the sorted data set based on the **CustomerAddress** column in ascending sort order:

```
SELECT TOP 1 * FROM Customer  
ORDER BY CustomerAddress ASC
```

The following screenshot shows the output of the query:

	CustomerID	CustomerName	CustomerAddress	CustomerMobile
1	3	Reema	India	3333333333

Figure 4.16: Fetching top 1 row from the list of customers sorted by CustomerID in ascending order

Following is the evolved example which will fetch the top 1 record from the sorted data set based on the **CustomerAddress** column in descending sort order:

```
SELECT TOP 1 * FROM Customer  
ORDER BY CustomerAddress DESC
```

The following screenshot shows the output of the query:

	CustomerID	CustomerName	CustomerAddress	CustomerMobile
1	1	John	Paris	1111111111

Figure 4.17: *Fetching top 1 row from the list of customers sorted by CustomerID in descending order*

UPDATE statement

The **UPDATE** statement is used to modify the existing records. If you've existing data in your table and wish to modify the data within the table then you'll have to use the

Let's understand the syntax and a simple example of **UPDATE** statement.

UPDATE

```
name>  
SET 1 name> = value for column 1>  
, 2 name> = value column 2>  
, ...  
, n name>) = value column n >  
WHERE name>  
/ OR> name>
```

Explanation of the syntax of **UPDATE** statement:

UPDATE is the keyword that is an instruction to modify the existing records.

name> has to be replaced with the actual table name.

SET is the keyword that is used to supply the new values to the columns.

name> has to be replaced with actual column names.

value> has to be replaced with new values for the respective columns.

WHERE is the keyword used to filter the data.

are the operators to combine multiple filter conditions.

Example of **UPDATE** statement:

The following query will modify the **CustomerAddress** column value to **India** for **CustomerID** = Originally, the **CustomerAddress** was **India** for **CustomerID** =

UPDATE Customer

SET CustomerAddress = 'Mumbai, India'

WHERE CustomerID = 3

Execute a **SELECT** statement on the **Customer** table to see the changes we made to **CustomerAddress** column of **CustomerID** = 3 using **UPDATE** statement. If you'll see the following screenshot, then you'll find the **CustomerAddress** modified with

The screenshot shows the SQL Server Enterprise Manager interface. The top toolbar includes a dropdown menu set to 'Learn_T_SQL', an 'Execute' button, and other standard icons. The main window displays two tabs: 'SQLQuery3.sql - LA...FCI0LP\kulde (52))*' and 'SQLQuery2.sql - LA...FCI0LP\kulde (52))*'. The active tab shows the query 'SELECT * FROM Customer'. Below the query editor, the 'Results' tab is selected, displaying a table with 5 columns: 'CustomerID', 'CustomerName', 'CustomerAddress', and 'CustomerMobile'. The table contains 3 rows of data. The first row is highlighted, showing '1' for CustomerID, 'John' for CustomerName, 'Paris' for CustomerAddress, and '1111111111' for CustomerMobile. The second row shows '2' for CustomerID, 'Kiya' for CustomerName, 'London' for CustomerAddress, and '2222222222' for CustomerMobile. The third row shows '3' for CustomerID, 'Reema' for CustomerName, 'Mumbai, India' for CustomerAddress, and '3333333333' for CustomerMobile. The 'Messages' tab is also visible but empty.

	CustomerID	CustomerName	CustomerAddress	CustomerMobile
1	1	John	Paris	1111111111
2	2	Kiya	London	2222222222
3	3	Reema	Mumbai, India	3333333333

Figure 4.18: Customer details post modification

DELETE statement

DELETE statement is used to delete/remove the existing records. If you've existing data in your table and wish to delete/remove the data within the table then you'll have to use the

Let's understand the syntax and a simple example of the **DELETE** statement.

DELETE FROM

```
name>  
WHERE name>  
/ OR> name>
```

Explanation of the syntax of **DELETE** statement:

DELETE is the keyword that is an instruction to delete/remove the existing records.

FROM is the keyword that is used to supply the table name from which the records are to be deleted/removed.

name> has to be replaced with the actual table name.

WHERE is the keyword used to filter the data.

are the operators to combine multiple filter conditions.

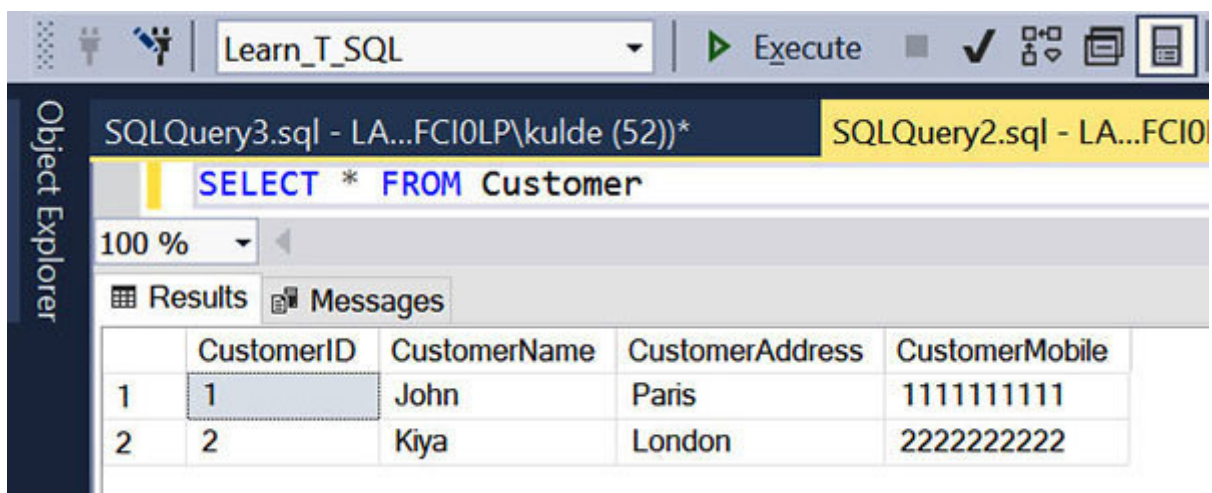
DELETE statement can be used with **TOP** command to limit the number of rows to be deleted. For example, **DELETE TOP (10) FROM**

Example of **DELETE** statement: The following query will delete the row belonging to the **CustomerID =**

DELETE FROM Customer

WHERE CustomerID = 3

Execute a **SELECT** statement on the **Customer** table, and you will find the row with **CustomerID = 3** no more exists in the table. The following screenshot shows the **SELECT** statement and it's resulting result. You can see the **CustomerID = 3** is not populating, because it has been deleted by the **DELETE** statement we ran:



The screenshot displays the SQL Server Enterprise Manager interface. The top toolbar includes a dropdown menu set to 'Learn_T_SQL', an 'Execute' button, and other standard icons. Below the toolbar, two query windows are visible: 'SQLQuery3.sql - LA...FCI0LP\kulde (52))*' and 'SQLQuery2.sql - LA...FCI0...'. The active window shows the query 'SELECT * FROM Customer'. Below the query, the 'Results' tab is selected, displaying a table with the following data:

	CustomerID	CustomerName	CustomerAddress	CustomerMobile
1	1	John	Paris	1111111111
2	2	Kiya	London	2222222222

Figure 4.19: *Customer details post deletion*

TRUNCATE statement

TRUNCATE statement removes all the rows from a table. It deletes rows by deallocating the pages allocated to the table. It makes an entry for the de-allocation of pages in the transaction log. It does not log each row deletion in the transaction log.

It is like **DELETE** statement, but it is not a DML statement. It is a DDL statement. There is one similarity between delete and truncate statement and that is - both of them remove the rows from tables. However, there are various differences between them. Let us understand such a differentiating factor:

TRUNCATE statement removes all the rows, whereas **DELETE** statement can remove specific or all the rows.

TRUNCATE statement resets the identity value, whereas **DELETE** statement does not reset the identity value. We'll talk about identity property further in detail in [Chapter 13](#).

A transaction with **TRUNCATE** statement cannot be rolled-back, whereas rollback is possible with **DELETE** statement. We'll talk about transactions further in detail in [Chapter 11](#), [Error Handling and Transaction](#)

TRUNCATE statement does not log the deletion of each row in the transaction log, whereas **DELETE** statement does.

TRUNCATE statement does not support the where clause, whereas **DELETE** statement does.

TRUNCATE statement is much faster as compared to **DELETE** statement. **DELETE** statement is slower because it logs the deletion of each row in the transaction log.

TRUNCATE statement is not supported on a table referenced by another table, whereas there is no such restriction with the **DELETE** statement.

TRUNCATE statement does not activate a DML trigger, whereas **DELETE** statement does. We'll talk about triggers further in detail in [Chapter 13](#).

TRUNCATE statement requires the alter permission on the table, whereas **DELETE** statement requires the delete permission on the table.

TRUNCATE statement consumes less transaction log space, whereas **DELETE** statement consumes comparatively more transaction log space.

TRUNCATE statement can be used on physical and temporary tables but it cannot be used on the table variables. Whereas, **DELETE** statement can be used on physical and temporary

tables, as well as on the table variables. We'll talk about temporary tables further in detail in [Chapter 10, Temporary Tables, CTE, and MERGE](#) Table-variables will be discussed further in detail in [Chapter 9, Variables and Control Flow](#)

Syntax of **TRUNCATE** statement:

TRUNCATE TABLE

name>

Example of **TRUNCATE** statement:

TRUNCATE TABLE Test

Magic tables (inserted and deleted tables)

SQL Server automatically creates and manages the *inserted* and *deleted* tables. They are also called the **magic**. These tables reside in memory and never be stored on a disk. The inserted table is created at the time of execution of **INSERT** statement. The deleted table is created at the time of execution of **DELETE** statement.

Both deleted and inserted tables are created at the time of execution of **UPDATE** statement. You must be wondering *why?*

UPDATE statement technically involves two different operations – **DELETE** and **INSERT**. When you run an **UPDATE** command, first the respective rows are deleted, and then the modified data is inserted in the table. That is why both deleted and inserted tables are created during the update.

The DML triggers are purely based on these magic tables (inserted and deleted tables). Additionally, these tables can be also used at the time of **DELETE** with **OUTPUT** clause to track the rows that have been inserted, updated, or deleted, along with the respective identity value (if the table has an identity column).

The inserted table is created for each **INSERT** and **UPDATE** statement at the time of execution (run-time). Similarly, the deleted table is created for each **DELETE** and **UPDATE**

statement at the time of execution (run-time). If the same DML statement is being executed at the same time in different sessions, each session will have a different inserted or deleted (depending upon the type of DML statement).

You cannot directly query (or access) these magic tables the way you can do with the normal tables. You can access these tables only with the help of the **OUTPUT** clause used with DML statements, or inside a DML trigger.

Let us see how to use the **OUTPUT** clause in DML statements.

OUTPUT clause with INSERT statement

Following is an example of **OUTPUT** clause with an **INSERT** statement. This is the same example as we have discussed earlier in this chapter in *INSERT* topic. We've just added the **OUTPUT** clause in the original query. If you've already executed the earlier query then you may get an error of primary key violation while running the following query. In that case, first, you'll have to delete the existing rows from the **Customer** table and then you can try running the following query:

```
INSERT INTO Customer  
OUTPUT  
VALUES
```

```
,  
,
```

The result of the preceding query will be similar to as can be seen in the following screenshot. You can see the entire row data of all the rows being inserted is returned with the help of **OUTPUT**. Here inserted is the magic table we've already talked about:

	CustomerID	CustomerName	CustomerAddress	CustomerMobile
1	1	John	Paris	1111111111
2	2	Kiya	London	2222222222
3	3	Reema	India	3333333333

Figure 4.20: Output of inserted table

If you do not want all the columns of the inserted table, you can also choose to fetch only the specific columns. The query will look like as can be seen as follows:

INSERT INTO Customer

OUTPUT
VALUES

,

The behavior of the inserted and deleted tables with other DML statements such as **UPDATE** and **DELETE** is similar to as what we've seen here in the case of However, let us learn the implementation of the **OUTPUT** clause with **UPDATE** and **DELETE** statements too.

OUTPUT clause with DELETE statement

Following is an example of the **OUTPUT** clause with the **DELETE** statement. It will also return the entire row data for all the rows being deleted:

```
DELETE FROM Customer  
OUTPUT  
WHERE CustomerID =
```

The preceding **DELETE** query can be also written as follow:

```
DELETE Customer  
OUTPUT  
WHERE CustomerID =
```


OUTPUT clause with UPDATE statement

Following is an example of **OUTPUT** clause with **UPDATE** Statement. If you'll notice we've used both deleted and inserted tables in the **OUTPUT** clause. You have the liberty to either use deleted, inserted, or both. You can also choose to select only the specific columns from inserted and deleted tables. Both of these deleted and inserted tables will return the same number of columns, but the data may vary (if the data is modified). The deleted table will return the old (earlier version of the data), whereas the inserted table will return the new (modified version of the data):

```
UPDATE Customer
SET CustomerName = 'Jim'
OUTPUT
WHERE CustomerID = 1
```

You can read more on the **OUTPUT** clause at the following URL of Microsoft official documentation:

<https://docs.microsoft.com/en-us/sql/t-sql/queries/output-clause-transact-sql?view=sql-server-ver15>

You can also insert the output of **OUTPUT** clause in a table-variable or temporary table using **INTO** clause.

You can read more on inserted and deleted tables at the following URL of Microsoft official documentation:

<https://docs.microsoft.com/en-us/sql/relational-databases/triggers/use-the-inserted-and-deleted-tables?view=sql-server-ver15>

Conclusion

The table is used to store the records or data. We push new records, modify existing records, delete existing records, and query or retrieve the data that the table holds.

In this chapter, we've learned various DML statements such as and **INSERT** statement is used to create new records in the table. The **UPDATE** statement is used to modify existing records in the table. **DELETE** statement is used to delete or remove existing records from the table.

The **SELECT** statement is used to query or retrieve records from the table. We understood various options that we can use with **SELECT** such as filtering using sorting using **ORDER**. We also understood various options associated with **WHERE** such as range search, wildcard search, and so on. We also talked about **TOP** and **OFFSET ...**

This chapter was an important milestone in learning T-SQL. Almost all the T-SQL queries include either of the DML statements we've discussed in this chapter.

In the next chapter, we'll explore more into T-SQL and learn about built-in functions of SQL Server. These functions are frequently used with/without DML statements to write complex T-SQL queries.

Points to remember

Values except numeric and bit has to be supplied in single quotes (').

Any column not included in **SELECT** can't be added in **ORDER**

Default sort order is **ASC** if not explicitly defined.

Range search can be applied only on Numeric and Date/Time data types.

DELETE without **WHERE** clause will delete the entire rows from the table.

UPDATE without **WHERE** clause will modify the column(s) with the new value(s) of the entire rows of the table.

Although **TOP** command can be used with **OFFSET ...**
However, it doesn't make sense to use **TOP** command with **OFFSET ...** **OFFSET ... FETCH** can be used to achieve the purpose of

Multiple choice questions

You want to fetch the top 20 records from a table. Which of the following options will you choose?

WHERE

TOP 20

BETWEEN

FETCH TOP 20

You got a requirement to get all the rows wherever BankName contains 'Of'. Which of the following options will you choose?

LIKE ' Of '

LIKE '% Of '

LIKE ' Of %'

LIKE '% Of %'

Answers

B

D

Questions

Which command is used to add new records in the table?

Which command is used to modify existing records in the table?

Which command is used to delete existing records in the table?

Can you delete columns using the **DELETE** command?

Can you apply range search on **varchar** columns?

Key terms

INSERT is used to create/add new records in the table.

UPDATE is used to modify the existing records of the table.

DELETE is used to delete/remove the existing records of the table.

SELECT is used to query/retrieve the records/data from the table.

WHERE is used to filter the records and can be used with and **DELETE** commands.

ORDER BY is used to sort the records.

CHAPTER 5

Built-In Functions - Part 1

We understood various DML statements in the previous chapter. You must now be comfortable with DML statements. In this chapter, we'll talk about various built-in functions and learn to implement them. Built-in functions are shipped by default with SQL Server. It means you don't have to do any additional activity to use them. Once you've installed the SQL Server, they are available for use.

Built-in functions are very important when it comes to querying and analyzing data. These functions play very vital roles in querying the meaningful data out of the database. If you need count, sum, average, minimum, and maximum values of rows, and so on, then you'll have to leverage these functions. There are many more such built-in functions with purposes varying from just the aggregation.

When we deal with data, it's not just dumping the data and retrieving the stored data. Data is playing a major role in the world of technology today. That is why Big Data, Data Science, and so on are booming. World today is focusing on leveraging the data available today. Similarly, when we talk about the relational database, it's more than just a placeholder. We can store the data in various forms including

flat files, Excel, and so on, but when we go with a relational database, there is a lot expected out of it.

T-SQL is a strong tool. But just like other tools, it has no meaning unless the person using it knows how to use it. Built-in functions are one of such useful features of SQL Server which gives more out of just a plain raw data. You can play with your data and generate meaningful analytics out of it with the help of these functions. There are many such features that we'll talk about gradually as we'll move ahead.

Structure

In this chapter, we will cover the following topics:

Aggregate functions

String functions

Numeric functions

Date functions

Objective

After studying this chapter, you'll be able to work with the various built-in functions ranging from aggregate functions used for aggregation, string functions used with the string's values, the numeric function used with the numeric values, and date functions used with the date values. These string, numeric, and date values could be a static value or a constant, a variable, or a column. By the end of this chapter, you'll be able to implement these built-in functions.

Aggregate functions

Before we understand the aggregate functions, let's first understand what a function is:

A function is a block of organized and reusable code used to perform predefined actions and return a value.

For example, if you want the addition of two values then you can create a function to accept two input parameters *value 1* and *value 2* and the function will perform *value 1 + value 2* and return the result. Suppose you supplied *value 1* as 100 and *value 2* as 200 then the function will return the result as $100 + 200 =$

Please note, a function always returns a value.

Let us extend the same example and suppose we've thousands of rows, and rows are continuously growing. *Can you write a similar customized function to perform the addition of all the values across the rows of a column?* Answer is

We need special functions to perform such tasks of grouping the rows and producing the desired meaningful single result such as summation, count, average, minimum, and maximum.

When a function performs such kinds of grouping (or aggregation), it's called **aggregate**

We also need to understand a few special T-SQL keywords/commands those are by product of aggregate functions. They are **GROUP BY** and **HAVING** clauses.

GROUP BY

As we already understood that aggregation is the process of grouping multiple rows. So, **GROUP BY** is the instruction in T-SQL to instruct to perform grouping on the set of columns.

Please note, you can use **GROUP BY** only with the **SELECT** command. If your **SELECT** statement has five columns and out of which aggregate function is applied on two columns and then you'll have to add the rest of the three columns and in the **GROUP BY** command, to instruct SQL Server to perform the aggregation of those two columns and by grouping the three columns and

GROUP BY command is a must in every query using aggregate functions on columns with other columns too used without aggregate functions. If only column(s) with aggregate function exists in the **SELECT** statement then **GROUP BY** is not needed.

We'll understand its syntax and practical implementation in further topics.

HAVING

HAVING is another special keyword/command in T-SQL that is used with an aggregate function. This command is used to filter the aggregate result. Its function is similar to **WHERE** command we learnt in [Chapter 4,](#) but it can be used only with aggregate functions. You can't filter the aggregate result using **WHERE** clause. You need the **HAVING** clause to filter it.

HAVING is a complimentary command that you can decide to use/not to use. If you want to filter the aggregate result, go for it and use, otherwise do not use it.

As a thumb rule, **GROUP BY** comes before the **HAVING** clause in the T-SQL query. It's logical too. You first need to group the dataset to get the aggregated result, and then you can filter it.

We'll understand its syntax and practical implementation in further topics.

Clauses and commands are used interchangeably and have the same meaning.

Populating sample data

We created three tables - Customer, Product, and PurchaseOrder in [Chapter 4](#). Since we need data for exploring various T-SQL queries in this chapter and future chapters, hence better we load these tables with some data to use it across.

The **Customer** table was populated with few sample records in [Chapter 4](#). We'll first clean the existing data from the **Customer** table using the following script. Alternatively, the **TRUNCATE** statement can also be used in such cases where the entire table needs to be deleted. **TRUNCATE** statement would be faster than **DELETE** statement. Although, there are certain limitations of using the **TRUNCATE** statement. You can read more about it in [Chapter 4](#).

DELETE FROM Customer

DELETE statement will delete all the rows from the **Customer** table. The **SELECT** statement returns no rows as shown in the following screenshot:

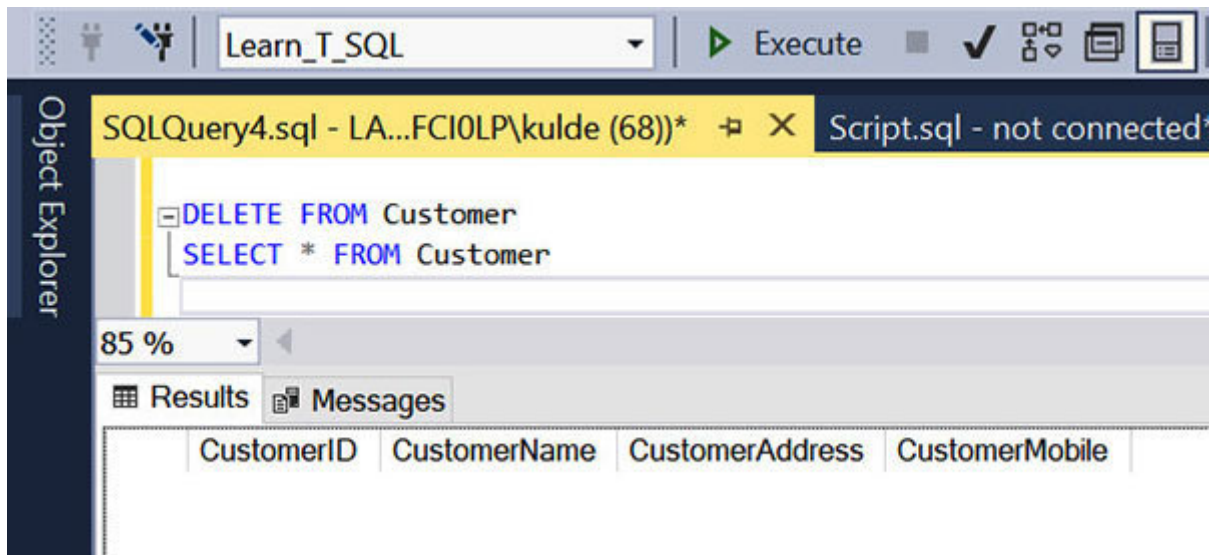


Figure 5.1: Deleting all the rows from Customer table

We'll now populate all the three tables with sample data using the following scripts:

```
INSERT INTO Customer
VALUES 'Customer 'Customer 1
, 'Customer 'Customer 2
, 'Customer 'Customer 3
, 'Customer 'Customer 4
, 'Customer 'Customer 5
, 'Customer 'Customer 6
, 'Customer 'Customer 7
, 'Customer 'Customer 8
, 'Customer 'Customer 9
, 'Customer 'Customer 9

, 'Customer 'Customer 11
, 'Customer 'Customer 12
, 'Customer 'Customer 13
, 'Customer 'Customer 14
```

, 'Customer 'Customer 15

INSERT INTO Product

VALUES 'Product

, 'Product

, 'Product

, 'Product

, 'Product

INSERT INTO PurchaseOrder

VALUES

,

,

,

,

,

,

,

,

,

,

,

,

,

,

,

,

,

,

,

,

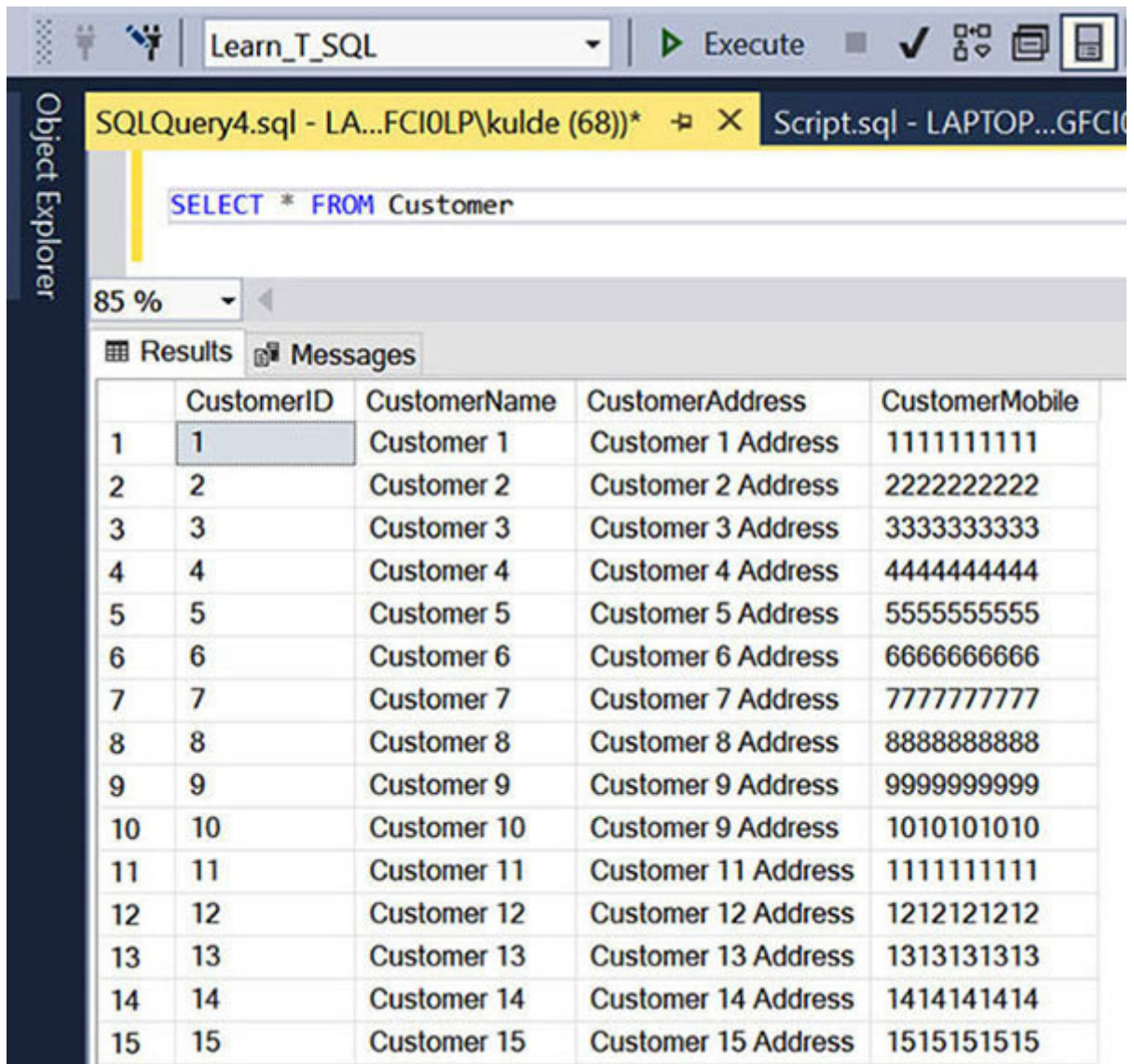
,

INSERT INTO PurchaseOrder

VALUES

```
,  
,  
,  
,  
,  
,  
,  
,  
,  
,  
,
```

Running the **SELECT** statement on the **Customer** table will return *15 rows* as we populated with the help of the **INSERT** script. It can be seen in the following screenshot:



SQLQuery4.sql - LA...FCI0LP\kulde (68))* X Script.sql - LAPTOP...GFCI0

SELECT * FROM Customer

85 %

Results Messages

	CustomerID	CustomerName	CustomerAddress	CustomerMobile
1	1	Customer 1	Customer 1 Address	1111111111
2	2	Customer 2	Customer 2 Address	2222222222
3	3	Customer 3	Customer 3 Address	3333333333
4	4	Customer 4	Customer 4 Address	4444444444
5	5	Customer 5	Customer 5 Address	5555555555
6	6	Customer 6	Customer 6 Address	6666666666
7	7	Customer 7	Customer 7 Address	7777777777
8	8	Customer 8	Customer 8 Address	8888888888
9	9	Customer 9	Customer 9 Address	9999999999
10	10	Customer 10	Customer 9 Address	1010101010
11	11	Customer 11	Customer 11 Address	1111111111
12	12	Customer 12	Customer 12 Address	1212121212
13	13	Customer 13	Customer 13 Address	1313131313
14	14	Customer 14	Customer 14 Address	1414141414
15	15	Customer 15	Customer 15 Address	1515151515

Figure 5.2: Output of Customer table after populating data

Running the **SELECT** statement on the **Product** table will return 5 rows as we populated with the help of the **INSERT** script. It can be seen in the following screenshot:

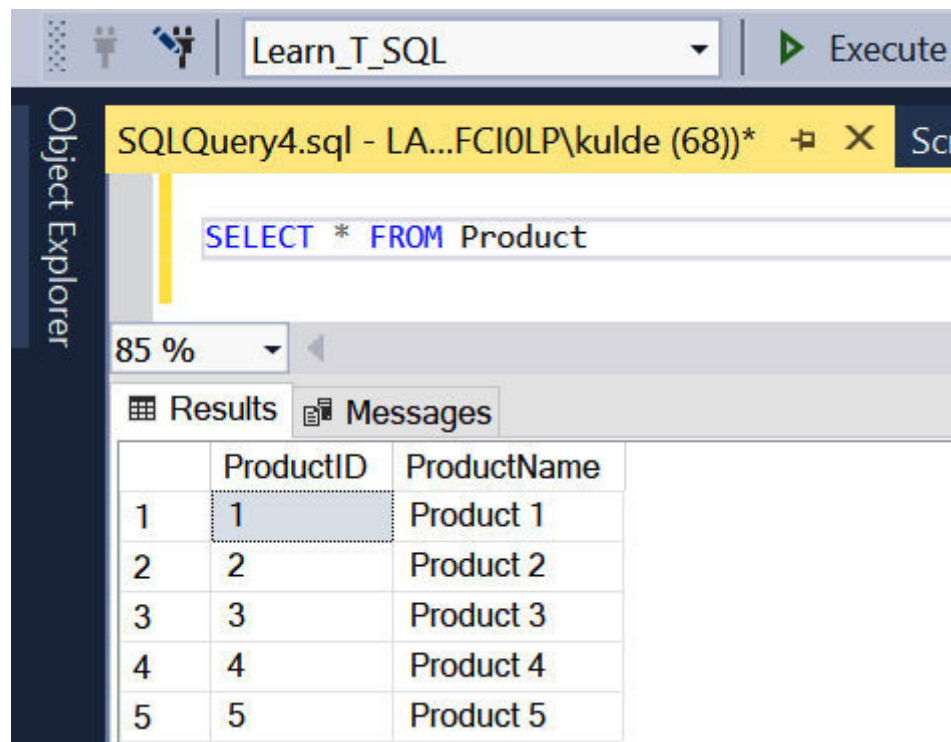


Figure 5.3: Output of Product table after populating data

Running the **SELECT** statement on the **Customer** table will return 33 rows as we populated with the help of the **INSERT** script. Since all the rows can't be shown in the single screenshot, hence it has been broken into two separate screenshots.

Following screenshot shows the first 20 rows as being returned by the **SELECT** statement:

SQLQuery4.sql - LA...FCI0LP\kulde (68))* X Script.sql - LAPTOP...GFCI0LP\kulde (64))*

SELECT * FROM PurchaseOrder

100 %

Results Messages

	OrderID	TransactionDate	CustomerID	ProductID	Quantity	Rate	Amount
1	1	2019-01-01	1	1	1	150000.00	150000.00
2	2	2019-05-15	1	2	1	250000.00	250000.00
3	3	2019-08-20	1	3	2	150000.00	300000.00
4	4	2019-10-20	1	4	1	350000.00	350000.00
5	5	2019-12-20	1	5	3	100000.00	300000.00
6	6	2019-01-01	2	1	1	150000.00	150000.00
7	7	2019-08-15	2	2	1	250000.00	250000.00
8	8	2019-12-20	3	3	2	150000.00	300000.00
9	9	2020-02-20	3	4	1	350000.00	350000.00
10	10	2020-03-20	3	5	3	100000.00	300000.00
11	11	2019-01-20	4	5	3	100000.00	300000.00
12	12	2019-05-01	5	1	1	150000.00	150000.00
13	13	2019-10-15	5	2	1	250000.00	250000.00
14	14	2019-04-20	6	3	2	150000.00	300000.00
15	15	2020-01-20	7	4	1	350000.00	350000.00
16	16	2020-02-20	7	5	3	100000.00	300000.00
17	17	2020-03-20	8	1	3	100000.00	300000.00
18	18	2019-01-20	8	3	3	100000.00	300000.00
19	19	2019-05-01	8	5	1	150000.00	150000.00
20	20	2019-10-15	9	2	1	250000.00	250000.00

Figure 5.4: Output of PurchaseOrder table after populating data

Following screenshot shows the next 13 rows after the 20th row as being returned by the **SELECT** statement:

21	21	2019-04-20	10	2	2	150000.00	300000.00
22	22	2020-01-20	11	4	1	350000.00	350000.00
23	23	2020-02-20	12	5	3	100000.00	300000.00
24	24	2019-01-01	13	1	1	150000.00	150000.00
25	25	2019-05-15	13	2	1	250000.00	250000.00
26	26	2019-08-20	13	3	2	150000.00	300000.00
27	27	2019-10-20	14	4	1	350000.00	350000.00
28	28	2019-12-20	14	5	3	100000.00	300000.00
29	29	2019-01-10	15	1	1	160000.00	160000.00
30	30	2019-05-25	15	2	1	200000.00	200000.00
31	31	2019-08-22	15	3	2	150000.00	300000.00
32	32	2020-01-13	15	4	1	350000.00	350000.00
33	33	2020-03-25	15	5	3	100000.00	300000.00

Figure 5.5: Output of Customer table after populating data continued

We've sufficient data and we can perform various queries on this dataset.

Let us start understanding each aggregate function in T-SQL one by one.

Syntax for using the aggregate functions

The following is the syntax for using aggregate functions in T-SQL:

```
SELECT SUM / COUNT / AVG / MIN / MAX (1 name>
, 2 name>
, 3 name>
, 4 name>
, 5 name>
```

FROM

name>

WHERE name>

/ OR> name>

GROUP BY 2 name>

, 3 name>

, 4 name>

, 5 name>

HAVING SUM / COUNT / AVG / MIN / MAX (1 name>)

ORDER BY name> /

All the operators you can use with the WHERE clause can also be used with the HAVING clause.

Here is the explanation of the syntax.

SELECT is the keyword that is an instruction to retrieve the records.

SUM / COUNT / MIN / MAX are aggregate functions.
Column name has to be specified in an aggregate function.

FROM is the keyword that is used to supply the table name from which the records are to be retrieved.

name> has to be replaced with the actual table name.

WHERE is the keyword used to filter the data. The list of operators that can be used with **WHERE** clause is discussed in

Chapter 4.

are the operators to combine multiple filter conditions.

GROUP BY is the keyword used to specify the grouping conditions on the set of columns.

HAVING is the keyword used to filter the aggregated result. All the operators that can be used with **WHERE** clause can also be used with the **HAVING** clause.

ORDER BY is the keyword used to sort the result set.

are the sort orders. **ASC** means *ascending* and **DESC** mean

SUM()

SUM is an aggregate function, used to perform the summation of rows of a column. Multiple columns can also be specified but only if there is an operator between them. Such as *column 1 + column* and so on.

For example, if we have data as shown in the following table and if we need to perform summation then we've only two options – perform addition of every single row or make use of aggregate function SUM. In the following table, we've only 10 Assume if you've thousands or millions of rows then *is it possible to add those individually?* As we already discussed earlier, the answer is

Table 5.1: Table with Amount column

We can use the **SUM** function and easily perform summation of all the rows of **Amount** column and get the single meaningful value as

The syntax is as follows:

SUM (name>)

The example is as follows:

The following is the query to get the summation of all the purchase orders for the period *April 2019 to 30th June*

```
SELECT  
FROM PurchaseOrder  
WHERE TransactionDate BETWEEN '2019-04-01' AND '2019-06-30'
```

Running the query will return the result as shown in the following screenshot:

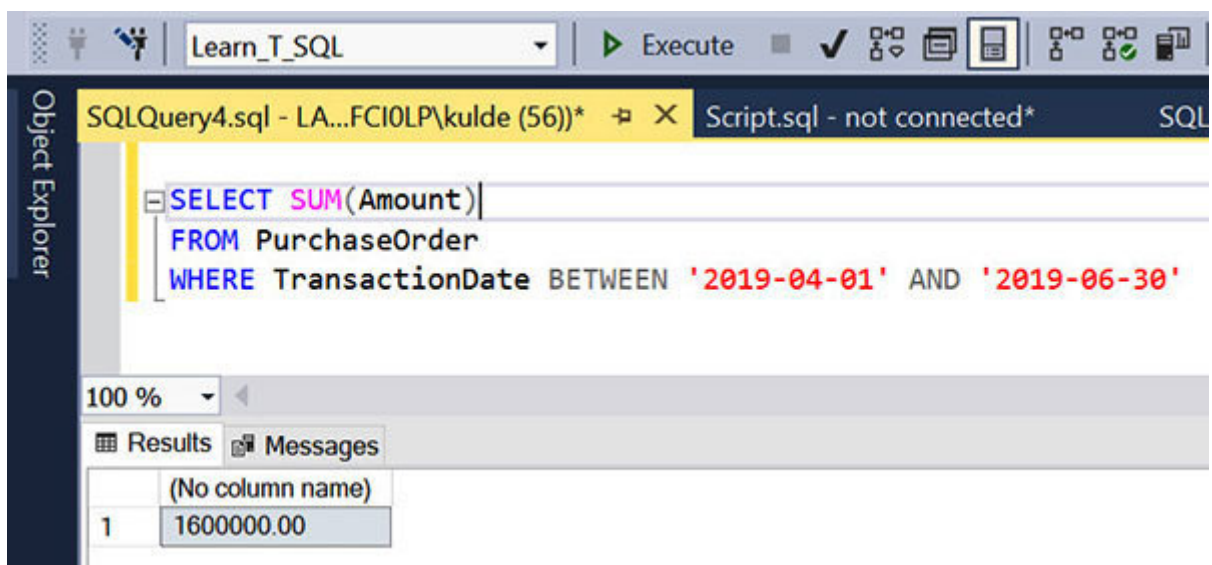


Figure 5.6: *Output of SUM, no column name is displayed in the result*

If you'll notice the column name in the result pane is coming as **(No column name)** as shown in the preceding screenshot. *Do you wonder why?* Because we've applied the aggregate function on the column and we've not specified the column name for the aggregated column.

We can provide the custom name to such columns using **AS** keyword as shown in the following example.

If you'll also notice the column name **TotalAmount** has been specified within the square bracket. Even if you'll not specify it within the bracket, the query will work. But if you would like to name it as **Total Amount** it won't due to space in the name:

```
SELECT AS [TotalAmount]
FROM PurchaseOrder
WHERE TransactionDate BETWEEN '2019-04-01' AND '2019-06-30'
```

AS keyword is used to provide the custom name to the columns. You can even use **AS** keyword with all the columns to specify the customer name to it, irrespective of whether the aggregate function is specified on the column or not. **AS** is optional, you can also specify the custom name by simply having a space between the custom name and the column

name. For example, SUM(Amount) However, it is advised to specify the AS keyword for the custom column name.

An object name such as table, column, procedure, function, and view names cannot have spaces. But if it is the demand to use the space in the name (including in the custom column name) then such name should be specified within the [] (square bracket). For example, SUM(Amount) AS [Total If you'll notice there is a space in the name.

Running the query will return the result as shown in the following screenshot. If you'll notice the column name in the result pane is now being shown as desired, that is,

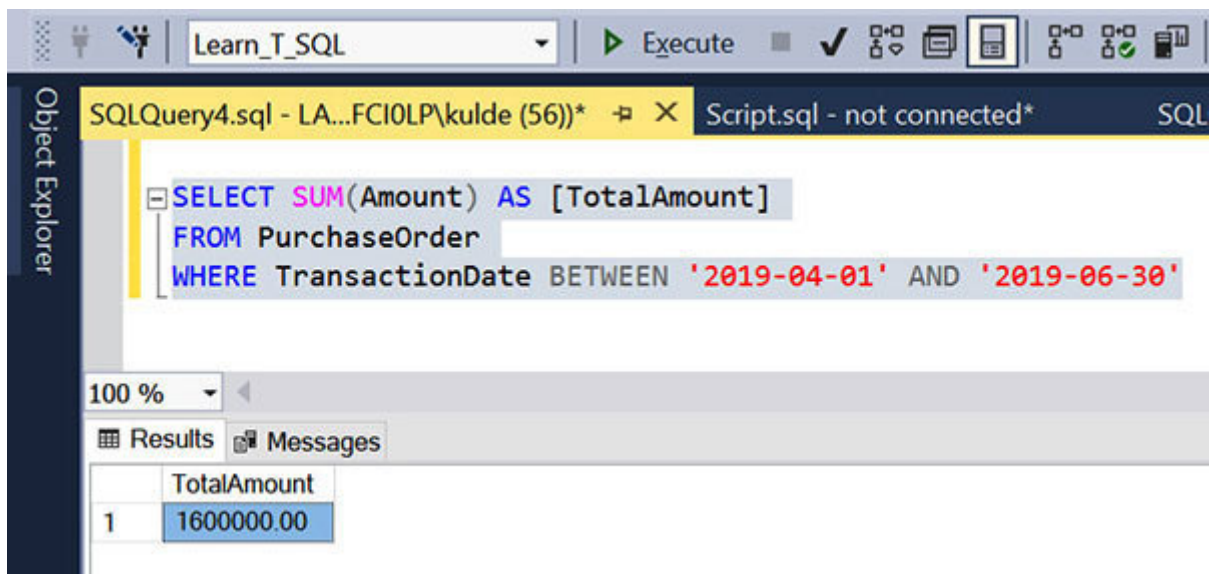


Figure 5.7: Output of SUM, column name is displayed in the result

The following is the query to get the summation of all the purchase orders for the period 1st April 2019 to 30 June 2019

for each customer:

```
SELECT CustomerID  
, AS [TotalAmount]  
FROM PurchaseOrder  
WHERE TransactionDate BETWEEN '2019-04-01' AND '2019-  
06-30'  
GROUP BY CustomerID
```

All the columns other than columns with aggregate function should be specified in the GROUP BY clause. If you'll not specify the query will fail with an error. GROUP BY is not needed if the aggregate function is not used in SELECT statement. GROUP BY is also not needed if only aggregate functions are used, and there is no other column specified in the SELECT statement.

Running the query will return the result as shown in the following screenshot:

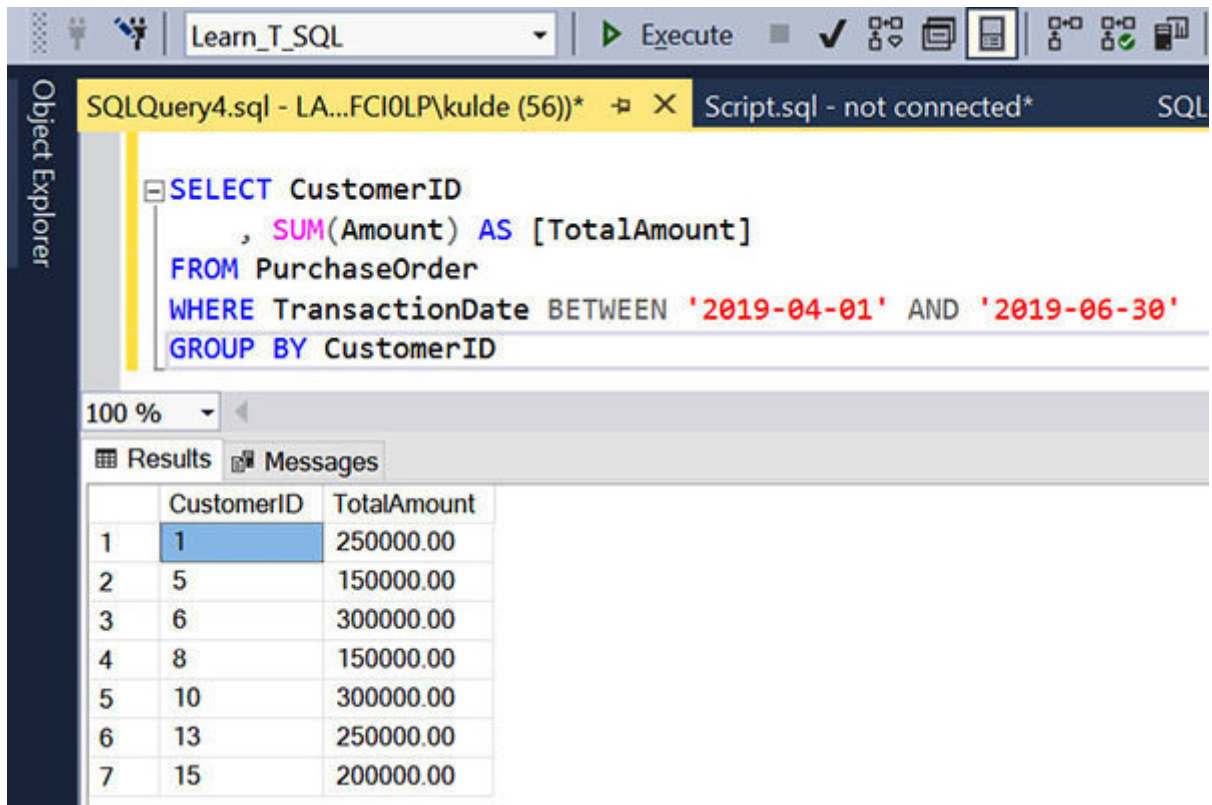


Figure 5.8: Output of SUM by CustomerID

The following is the query to get the summation of all the purchase orders made in the year for each customer and for each product, where the total amount is greater than and sort the result set by total amount in the descending order, and show only *top 5* results:

```
SELECT TOP 5 CustomerID
      , ProductID
      , AS [TotalAmount]
FROM PurchaseOrder
WHERE TransactionDate BETWEEN '2019-01-01' AND '2019-12-31'
GROUP BY CustomerID
      , ProductID
```

HAVING > 100000
ORDER BY [TotalAmount] DESC

Running the query will return the result as shown in the following screenshot:

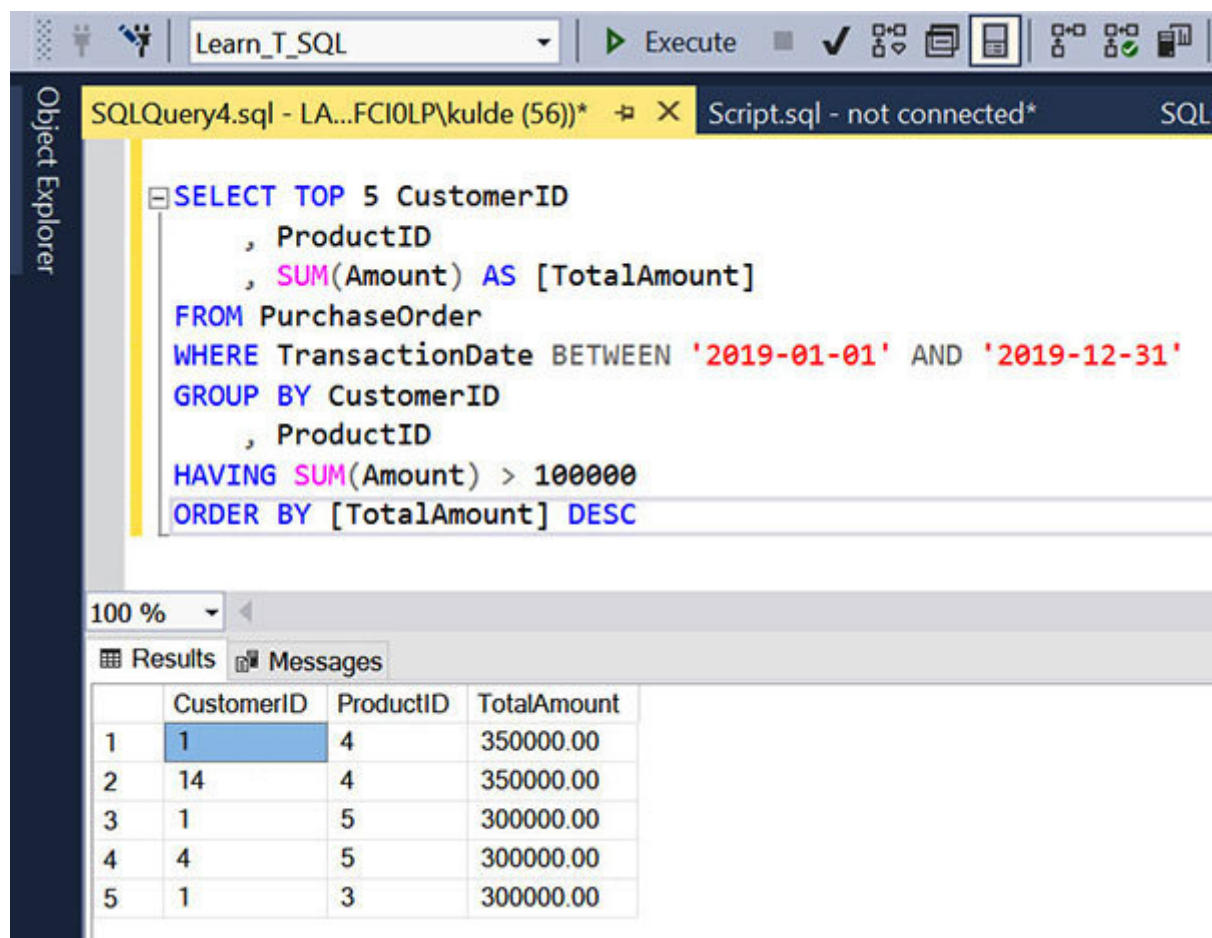


Figure 5.9: Output of SUM by CustomerID and ProductID, returning only top 5 rows sorted by TotalAmount in descending order

The same query can be written as following to sort the result set based on the column sequence number instead of the column name. Column sequence starts from

In the following query, the column sequence for **CustomerID** is 1, for **ProductID** is 2, and for TotalAmount is 3, that's why 3 is used instead of TotalAmount in the **ORDER BY** clause of the following query. Sorting can be done in either way interchangeably. Both will perform the same action and return the same result:

```
SELECT TOP 5 CustomerID
, ProductID
, AS [TotalAmount]
FROM PurchaseOrder

WHERE TransactionDate BETWEEN '2019-01-01' AND '2019-12-
31'
GROUP BY CustomerID
, ProductID
HAVING > 100000
ORDER BY 3 DESC
```

Running the query will return the result as shown in the following screenshot:

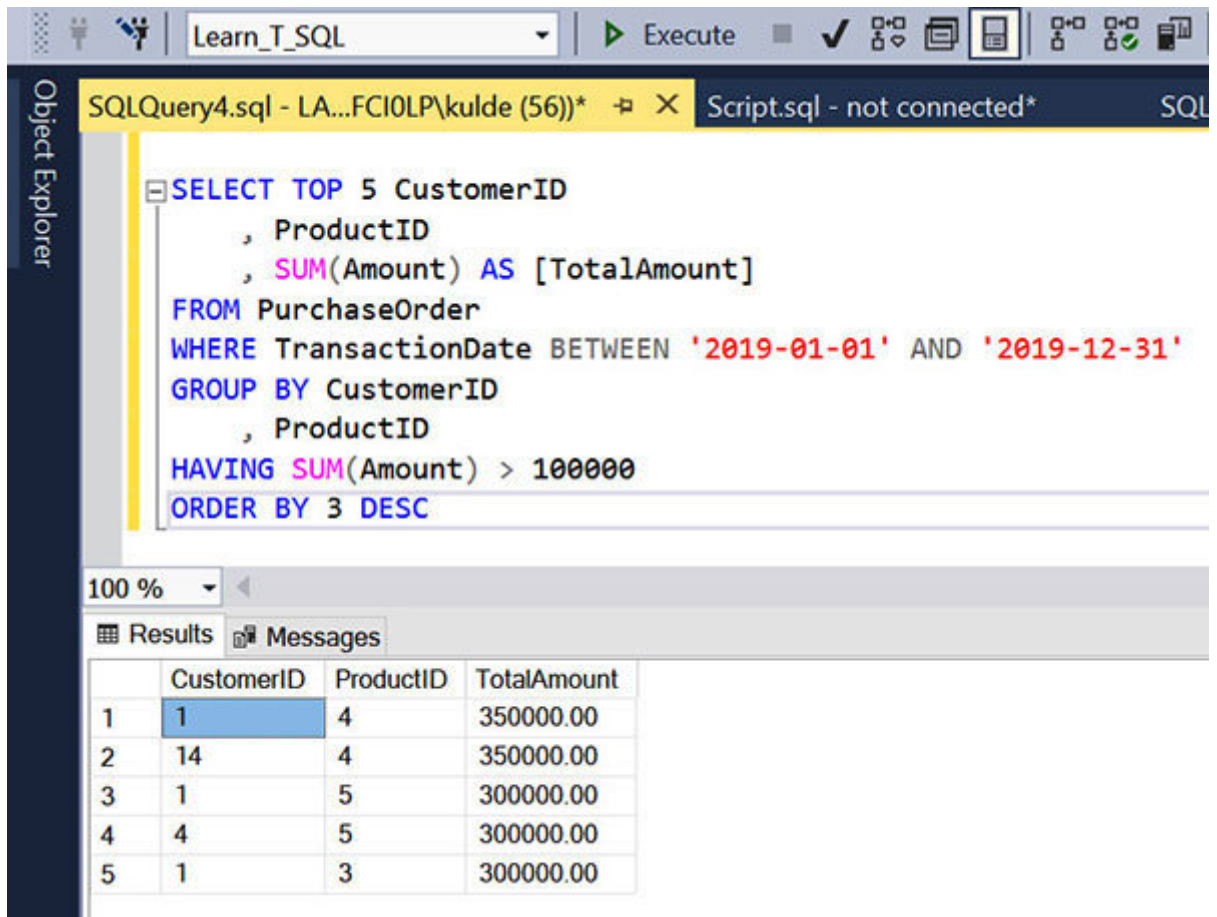


Figure 5.10: Output of SUM by CustomerID and ProductID, returning only top 5 rows sorted by (in descending order) column at the position (TotalAmount)

COUNT()

COUNT is the aggregate function used to count the number of rows of a particular column or all columns.

COUNT (name>)

It is used to count the number of rows in the column specified within the count function. However, it will only count the non-null values. The rows having the null values for the column shall not be considered. For example, if the values are – it will return

The syntax is as follows:

COUNT (name>)

COUNT(*)

It is used to count the number of rows being returned from the query. If there is just a single table in the query on which **COUNT(*)** is used then it will return the count of number of the rows in the table, if no **WHERE** clause is specified.

For example, **SELECT COUNT(*) FROM Customer** will return the count of all the rows of the **Customer** tables.

But if there is **WHERE** clause or any other joins to filter the rows then it will return the count of the number of rows returned by the query.

For example, **SELECT COUNT(*) FROM Customer WHERE City = 'Mumbai';** will return the count of all the rows of the **Customer** tables where the **City** is **Mumbai**. If there are total 100 customers out of which 10 belongs to **Mumbai**, the query will return 10.

Similarly, if the **COUNT(*)** is used with a query having multiple joins then it will return the count of the number of rows being returned by the query.

The syntax is as follows:

COUNT (*)

COUNT (DISTINCT name>)

It is another flavor of **COUNT** function. It is used to count the distinct number of rows of a particular column. If there are duplicate values, only one of them will be considered.

Suppose a column with values mentioned in comma separated format as following:

COUNT (*) and **COUNT (name>)** will return whereas **COUNT (DISTINCT name>)** will return

The syntax is as follows:

COUNT name>)

Example is as follows:

Following is the query to get the number of purchase orders and number of unique customers, for the year

```
SELECT AS [NumberOfOrders]
, AS [NumberOfCustomers]
FROM PurchaseOrder
WHERE TransactionDate BETWEEN '2019-01-01' AND '2019-12-31'
```

Running the query will return the result as shown in the following screenshot:

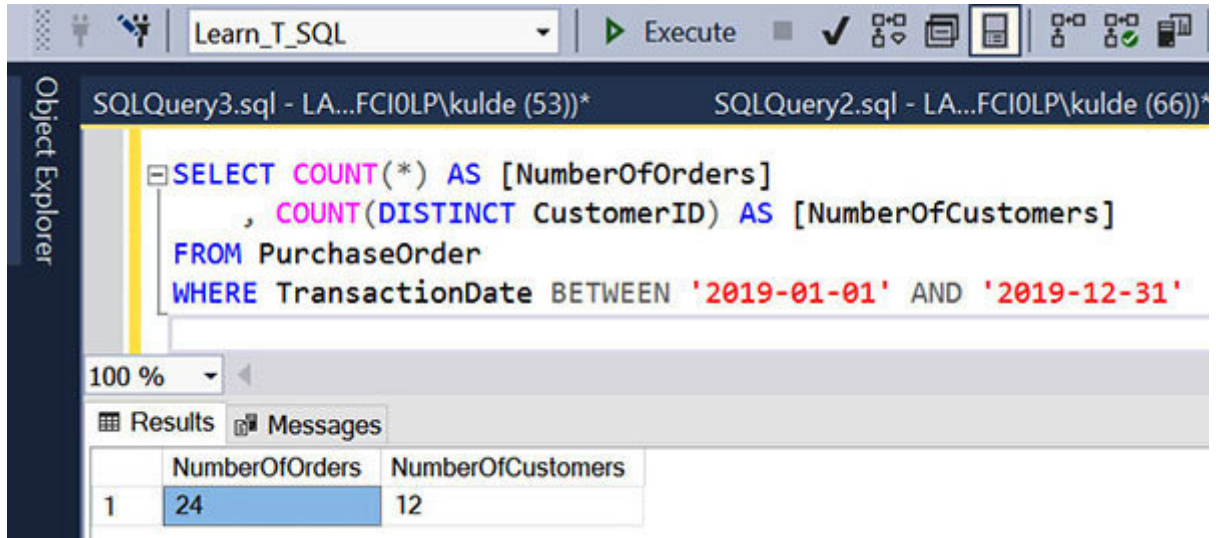


Figure 5.11: Output of COUNT functions

The following is the query to get the number of purchase orders, by each customer, for the entire period:

```
SELECT CustomerID
      , AS [NumberOfOrders]
FROM PurchaseOrder
GROUP BY CustomerID
```

Running the query will return the result as shown in the following screenshot:

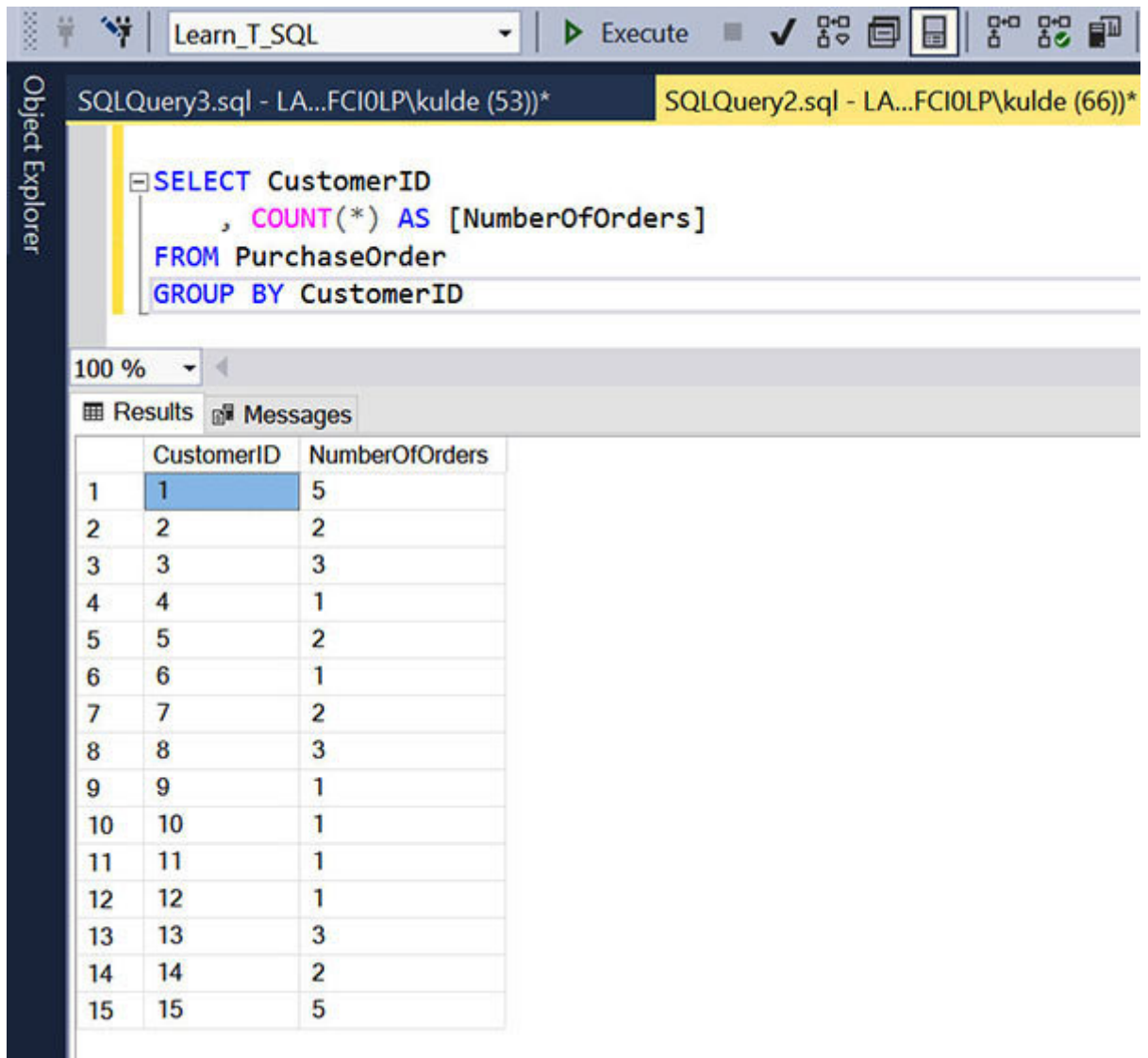


Figure 5.12: Output of COUNT by CustomerID

AVG()

AVG is an aggregate function used to calculate the average of rows of a column. Multiple columns can also be specified but only if there is an operator between them. Such as *column 1* + *column* and so on.

The syntax is as follows:

AVG (name>)

The example is as follows:

The following is the query to get the average of the purchase order amount, by each customer, for the year

```
SELECT CustomerID
, AS [AvgAmount]
FROM PurchaseOrder
WHERE TransactionDate BETWEEN '2019-01-01' AND '2019-12-
31'
GROUP BY CustomerID
```

Running the query will return the result as shown in the following screenshot:

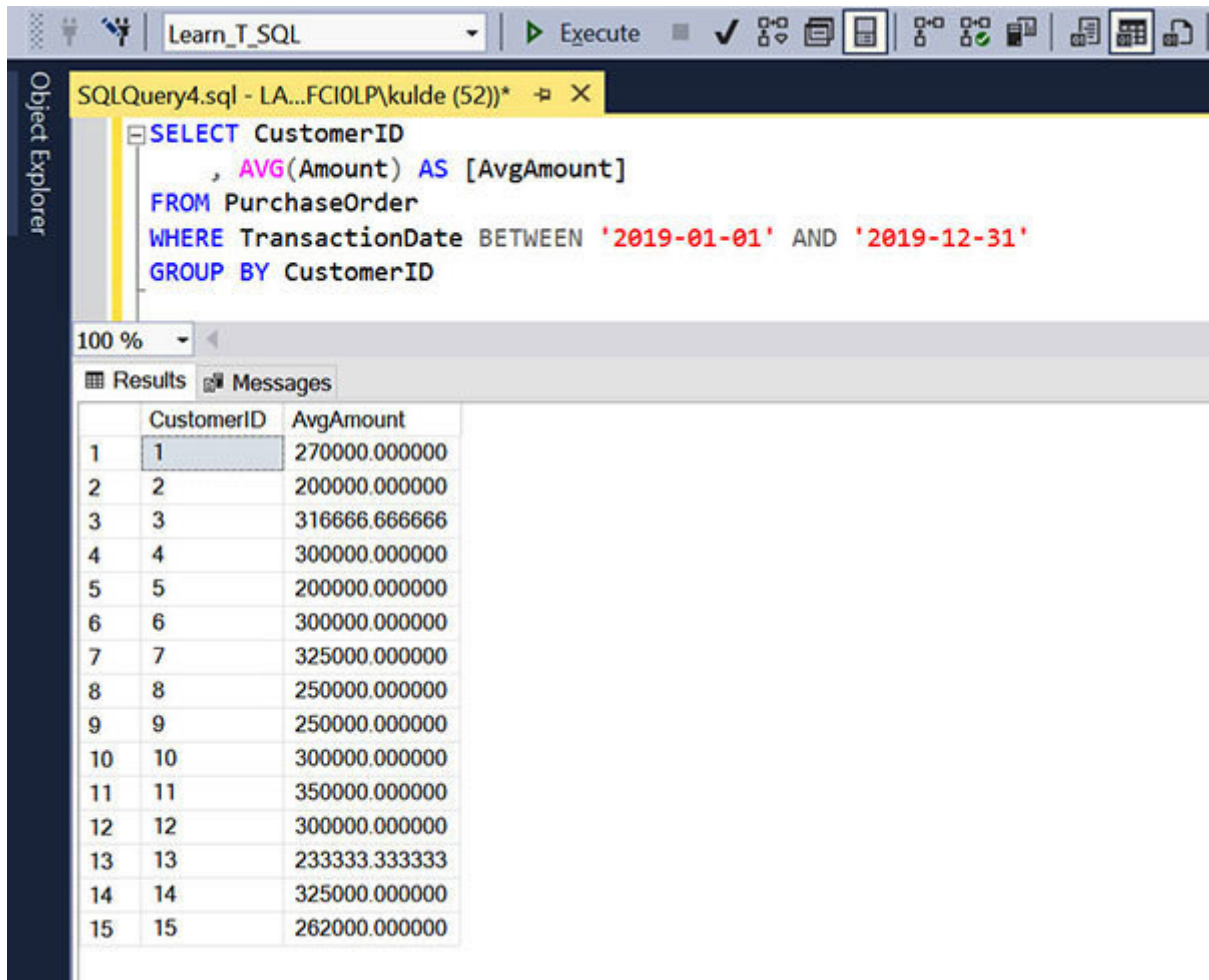


Figure 5.13: Output of AVG by CustomerID

MIN()

MIN is an aggregate function used to calculate the minimum of rows of a column. Multiple columns can also be specified but only if there is an operator between them. Such as *column 1 + column* and so on.

Syntax is as follows:

MIN (name>)

The example is as follows:

The following is the query to get the minimum of the purchase order amount, by each customer, for the year

```
SELECT CustomerID
, AS [MinAmount]
FROM PurchaseOrder
WHERE TransactionDate BETWEEN '2019-01-01' AND '2019-12-
31'
GROUP BY CustomerID
```

Running the query will return the result as shown in the following screenshot:

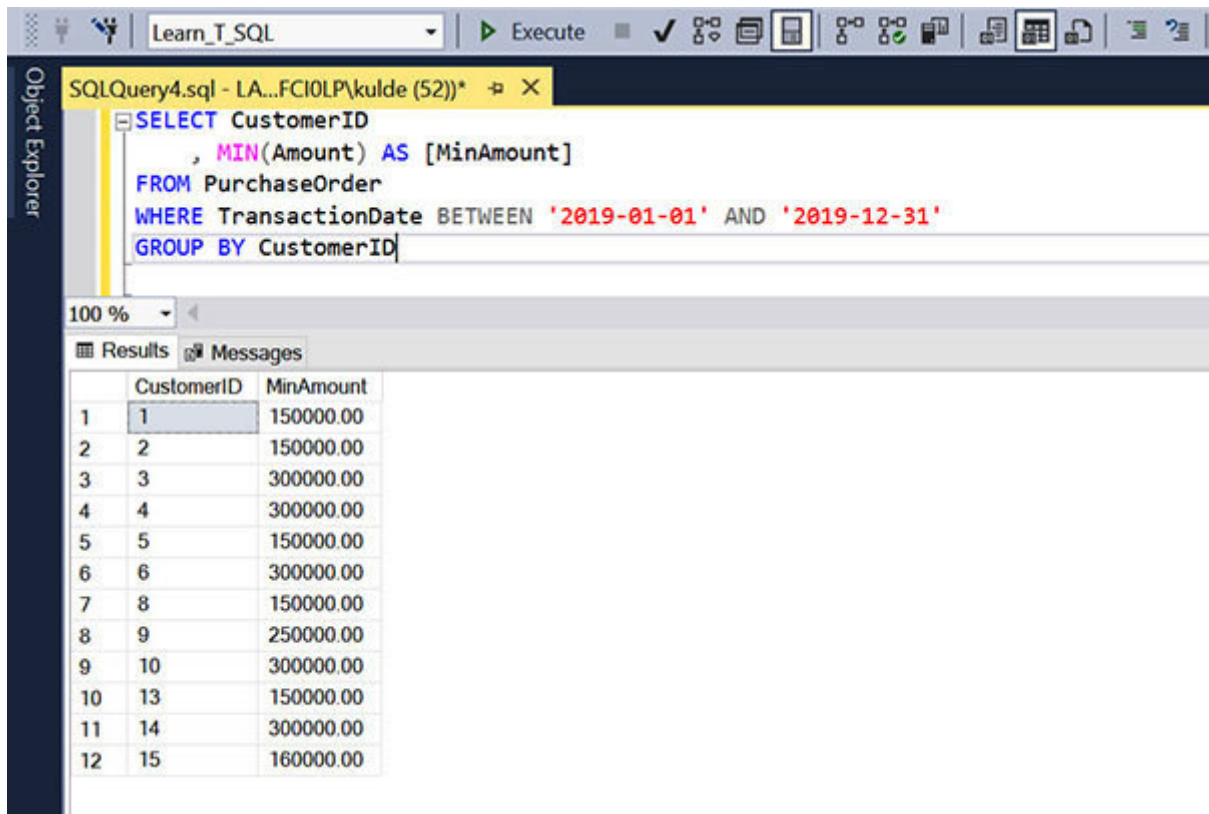


Figure 5.14: Output of MIN by CustomerID

MAX ()

MAX is an aggregate function used to calculate the maximum of rows of a column. Multiple columns can also be specified but only if there is an operator between them. Such as *column 1 + column* and so on.

The syntax is as follows:

MAX (name>)

The example is as follows:

The following is the query to get the maximum of the purchase order amount, by each customer, for the year

```
SELECT CustomerID
, AS [MaxAmount]
FROM PurchaseOrder
WHERE TransactionDate BETWEEN '2019-01-01' AND '2019-12-
31'
GROUP BY CustomerID
```

Running the query will return the result as shown in the following screenshot:

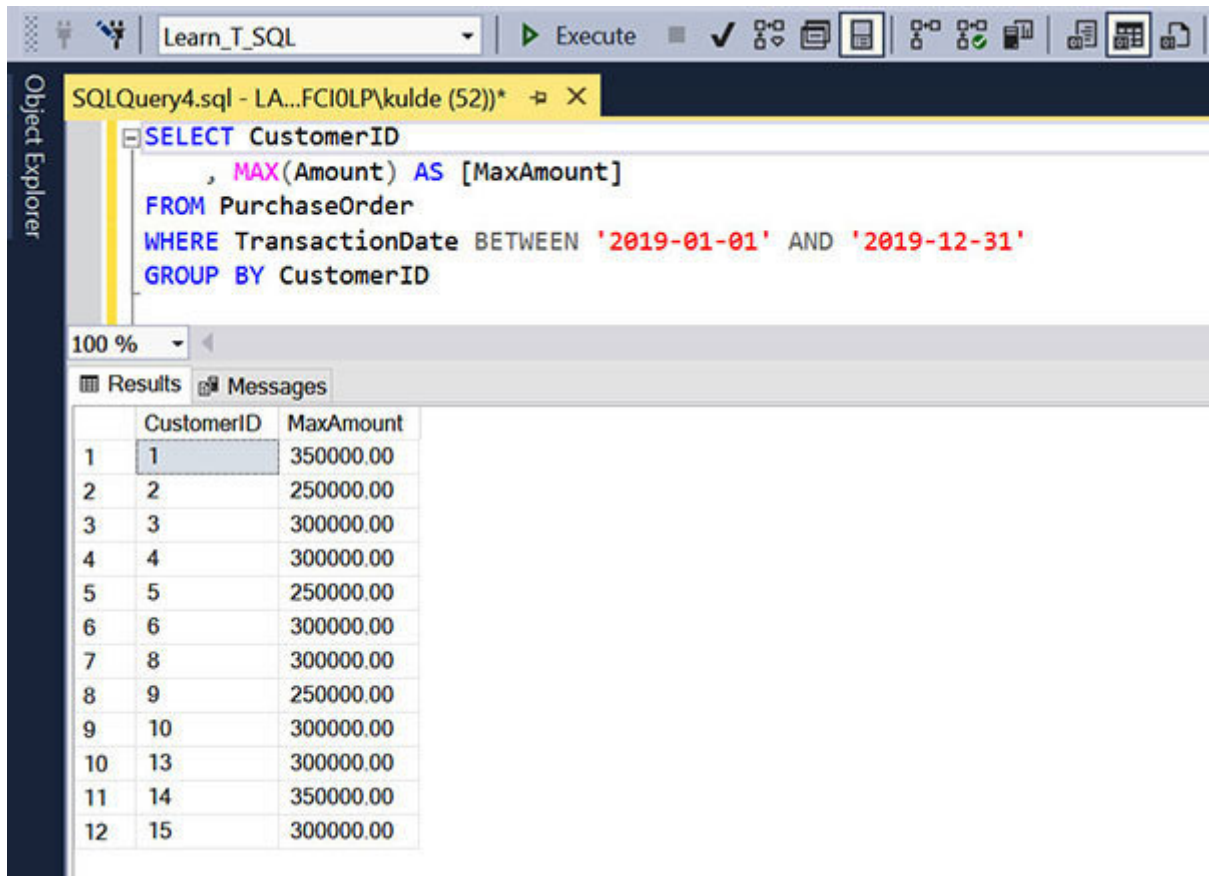


Figure 5.15: Output of MAX by CustomerID

Aggregate functions with INSERT

So far, we've seen the application of aggregate functions with the **SELECT** statement. We'll now discuss how to use these functions with an **INSERT** statement.

We've already talked about the **INSERT** command in [Chapter 4](#). We learnt about **INSERT ... VALUES** to add new records in a table. **INSERT** can also be used with **SELECT** for the batch insert. If we know the values to be inserted, then we use the **INSERT ... VALUES** way. But if we want to insert the output of a **SELECT** statement then we use **INSERT ...**

Let us have a look into how it works.

Assume we've a table that stores the yearly total of all the purchase orders made in the year for each customer. The table structure of the said table will look like as follows:

```
CREATE TABLE PurchaseOrderYearlySummary
(
  , CustomerID    INT
  , Quantity      INT
  ,
  , Amount
)
```

We need to average out the rate, sum up the quantity, and amount for each year and for each customer from purchase order and insert the output in the newly created table. All such new rows are to be posted against the last date of each year as the transaction date.

If we've to achieve it through **INSERT ... VALUES** method then we need to first get the result in excel and then generate the **VALUES** script for each row. This would become extremely difficult or we can also say impossible if there are billions of rows.

What is the solution then?

Batch insert also called **INSERT ... SELECT** comes to the rescue. The following script can be used to achieve it:

```
INSERT INTO PurchaseOrderYearlySummary
(
  TransactionDate
, CustomerID
, Quantity
, Rate
, Amount
)
SELECT AS + '-12-31' AS TransactionDate
, CustomerID
, AS Quantity
```

```
, AS Rate
, AS Amount
FROM PurchaseOrder
GROUP BY CustomerID
ORDER BY CustomerID ASC
```

Running the query will return the result as shown in the following screenshot. Total 18 rows are inserted:

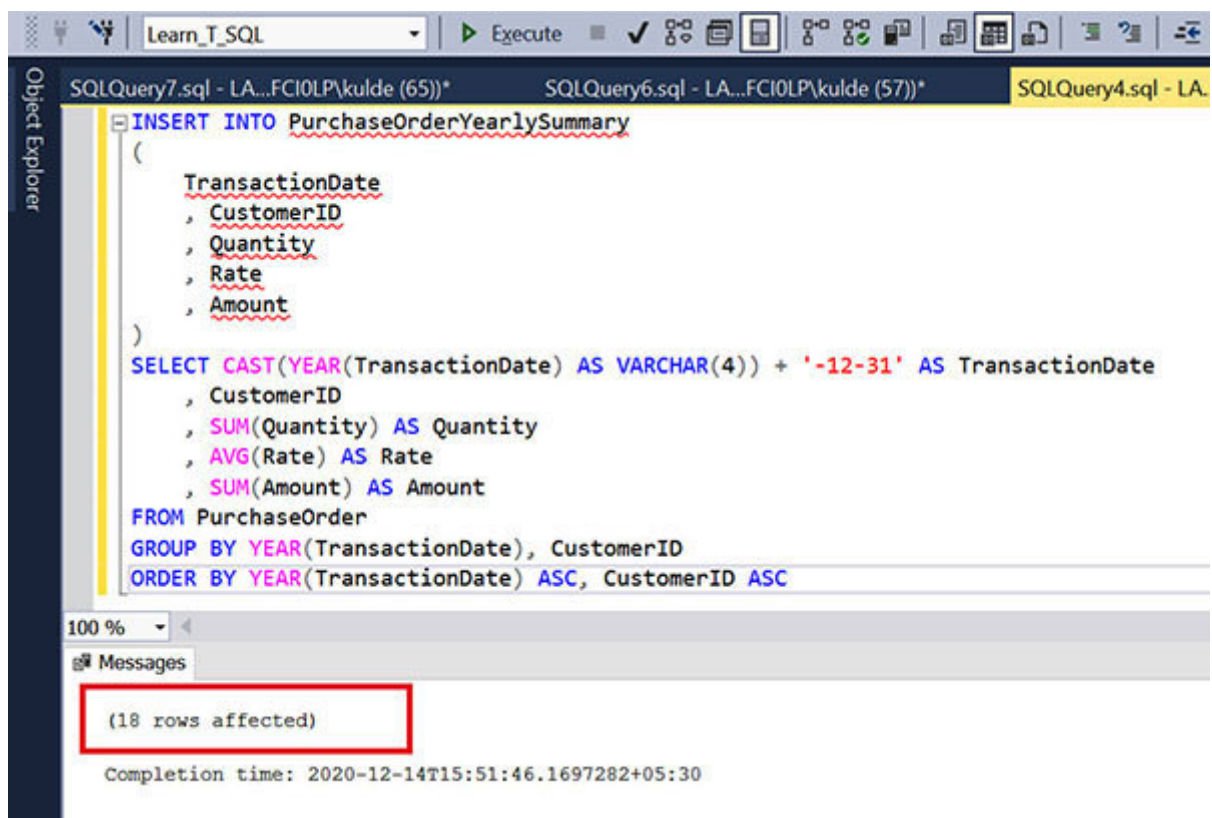


Figure 5.16: Aggregate functions with INSERT

Let us see what data has been inserted. We'll run a simple select statement on the **PurchaseOrderYearlySummary** table as shown in the following screenshot:

Object Explorer

Learn_T_SQL

Execute

SQLQuery7.sql - LA...FCI0LP\kulde (65))*

SQLQuery6.sql - LA...FCI0LP\kulde (57))*

```
SELECT * FROM PurchaseOrderYearlySummary
ORDER BY TransactionDate ASC, CustomerID ASC
```

100 %

Results Messages

	TransactionDate	CustomerID	Quantity	Rate	Amount
1	2019-12-31	1	8	200000.00	1350000.00
2	2019-12-31	2	2	200000.00	400000.00
3	2019-12-31	3	2	150000.00	300000.00
4	2019-12-31	4	3	100000.00	300000.00
5	2019-12-31	5	2	200000.00	400000.00
6	2019-12-31	6	2	150000.00	300000.00
7	2019-12-31	8	4	125000.00	450000.00
8	2019-12-31	9	1	250000.00	250000.00
9	2019-12-31	10	2	150000.00	300000.00
10	2019-12-31	13	4	183333.33	700000.00
11	2019-12-31	14	4	225000.00	650000.00
12	2019-12-31	15	4	170000.00	660000.00
13	2020-12-31	3	4	225000.00	650000.00
14	2020-12-31	7	4	225000.00	650000.00
15	2020-12-31	8	3	100000.00	300000.00
16	2020-12-31	11	1	350000.00	350000.00
17	2020-12-31	12	3	100000.00	300000.00
18	2020-12-31	15	4	225000.00	650000.00

Figure 5.17: Output of PurchaseOrderYearlySummary table

String functions

String functions are the type of functions used with string data. There are various string functions. Although, we'll talk about most commonly used ones.

LEFT()

LEFT function is used to select the specified number of characters from the left side of a string.

The syntax of **LEFT** function is as follows:

LEFT (, number of characters)

The explanation of **LEFT** function is as follows:

Suppose we've a string **abcdef** and we want two characters from left, then using the **LEFT** function will return

Example of **LEFT** function is as follows:

SELECT

Upon running the preceding query will return the result **ab** as shown in the following screenshot:

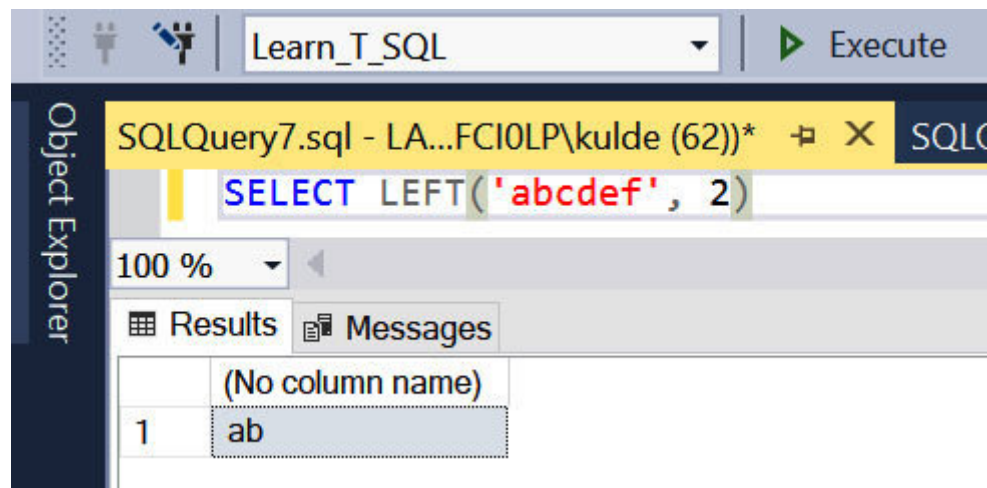


Figure 5.18: *LEFT function*

RIGHT()

RIGHT function is used to select the specified number of characters from right side of a string.

Syntax is as follows:

RIGHT (, number of characters)

The explanation is as follows:

Suppose we've a string **abcdef** and we want two characters from right, then using the **RIGHT** function will return

Example of **RIGHT** function is as follows:

SELECT

Upon running the preceding query will return the result **ef** as shown in the following screenshot:

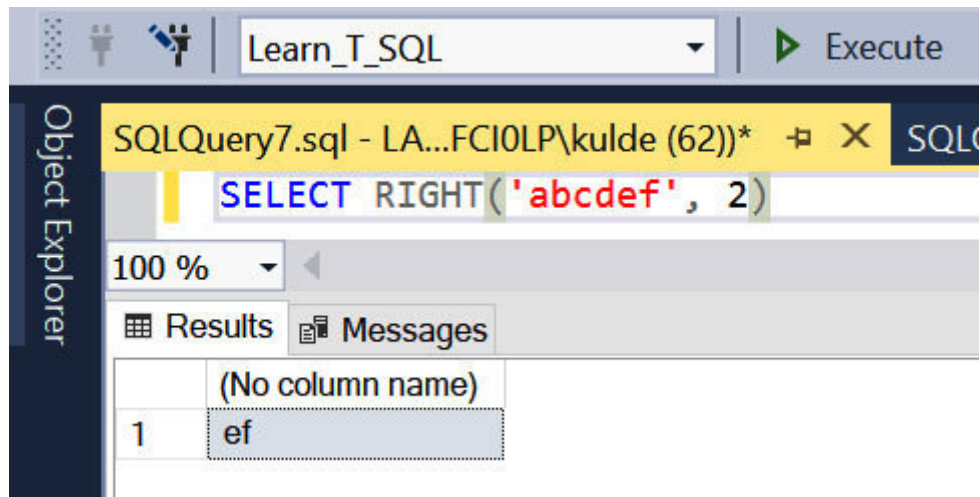


Figure 5.19: *RIGHT function*

SUBSTRING ()

SUBSTRING function is used to select the specified number of characters from the specified character position. Character position starts from Suppose if there is a string **abcdef** then **a** has character position as **b** as **c** as and so on.

Syntax is as follows:

SUBSTRING (, character position, number of characters)

The explanation is as follows:

Suppose we've a string **abcdef** and we want two characters from the third character onwards, then using the **SUBSTRING** function will return

Example of **SUBSTRING** function is as follows:

SELECT

Upon running this query will return the resulting **cd** as shown in the following screenshot:

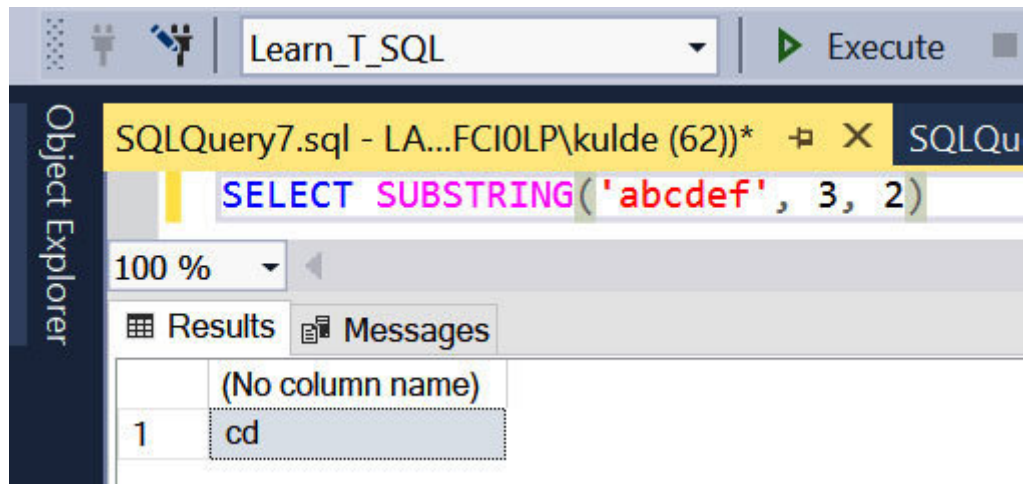


Figure 5.20: *SUBSTRING function*

REPLACE()

REPLACE function is used to replace the existing character(s) in a string with the new character(s).

The syntax is as follows:

REPLACE (, character(s) to be replaced, new character(s))

The explanation is as follows:

Suppose we've a string **adleebe** and we want to replace **lee** with **o** then using the **REPLACE** function will return

Similarly, if we've a string **adleede** and we want to replace **e** with **o** then using the **REPLACE** function will return

It is to be noted that if you want to replace multiple characters individually then you'll have to run **REPLACE** function multiple times. For example, if we've a string **adleede** and we want to replace **e** with **o** and **l** with **z** then we'll run **REPLACE** function twice, first for **e** and then for **l**. We'll see this in action in the following examples:

Examples are as follows:

SELECT
SELECT
SELECT

Upon running the previous queries will return the result as shown in the following screenshot:

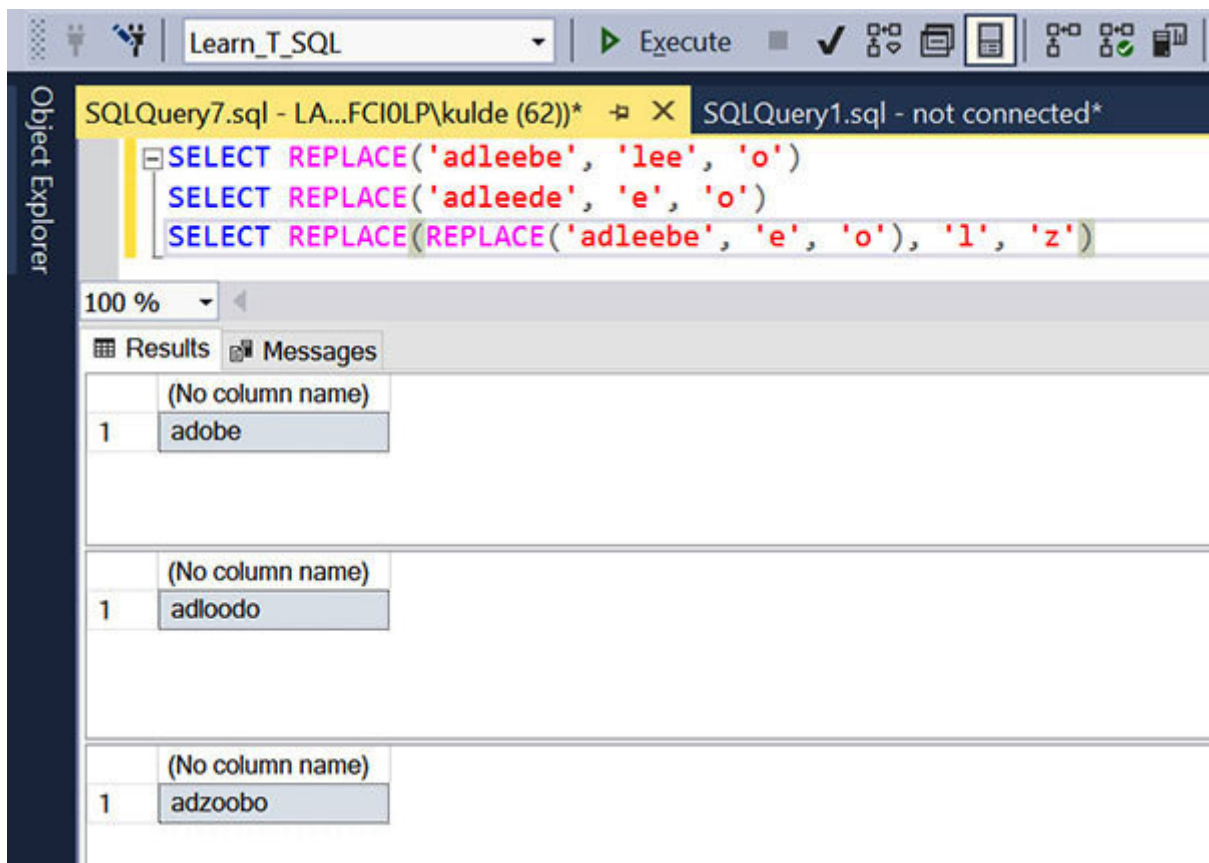


Figure 5.21: REPLACE function

LEN ()

LEN function is used to get the count of characters in a string.

Syntax is as follows:

LEN ()

The explanation is as follows:

Suppose we have a string **adleebe** and we want to count the number of characters in the string. Using the **LEN** function will return We'll see the working of **LEN** function using the following example:

The example is as follow:

SELECT

Upon running the query will return the result as shown in the following screenshot:

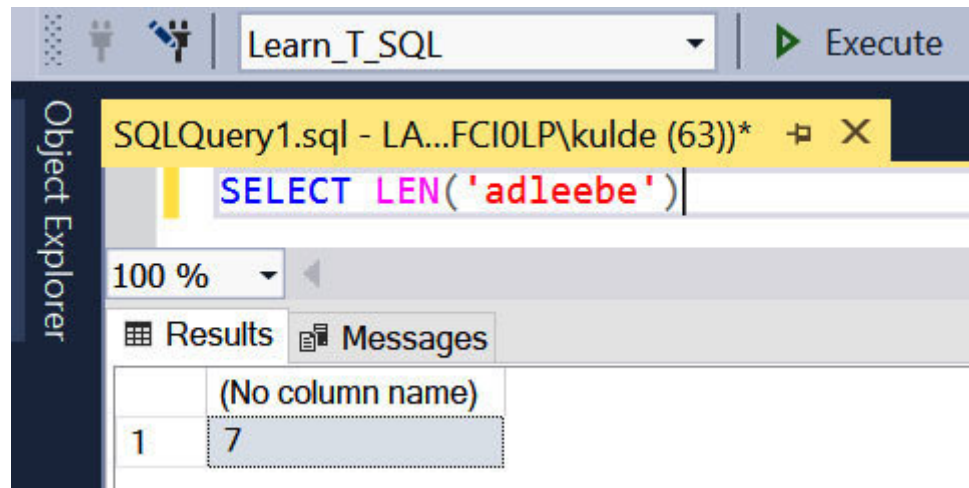


Figure 5.22: *LEN function*

LTRIM()

Sometimes a string contains the leading spaces. For example,
' You can see there are two space characters before

The **LTRIM** function is used to get the string without leading space characters.

The syntax is as follows:

LTRIM ()

Explanation is as follows:

Suppose we've a string ' and we want the string without the leading space characters. Using the **LTRIM** function will return We'll see the working of the **LTRIM** function using the following example.

Example is as follows:

SELECT

Upon running the query will return the result as shown in the following screenshot:

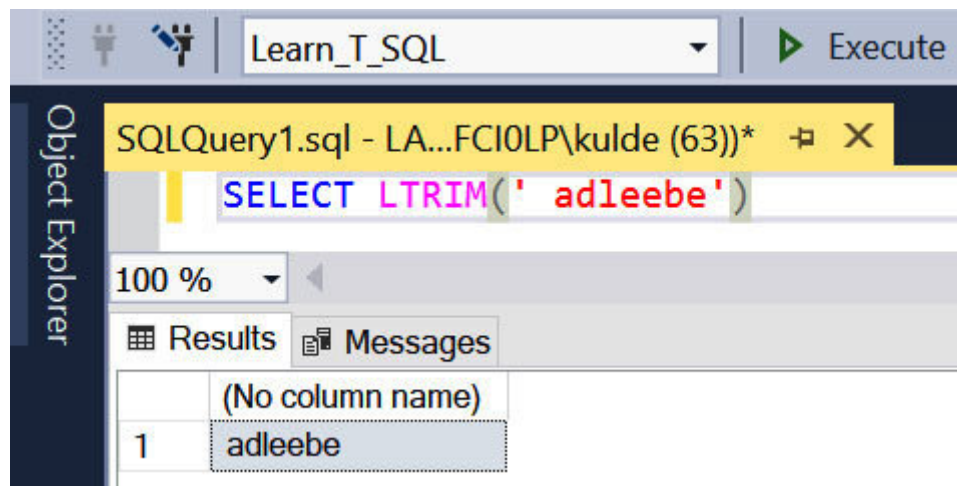


Figure 5.23: *LTRIM function*

RTRIM()

Sometimes a string contains the trailing spaces. For example, You can see there are two space characters after

RTRIM function is used to get the string without trailing space characters.

The syntax is as follows:

RTRIM ()

The explanation is as follows:

Suppose we've a string ' ' and we want the string without the trailing space characters. Using the **RTRIM** function will return We'll see the working of **RTRIM** function using the following example:

These example is as follows:

SELECT

Upon running the preceding query will return the result as shown in the following screenshot:

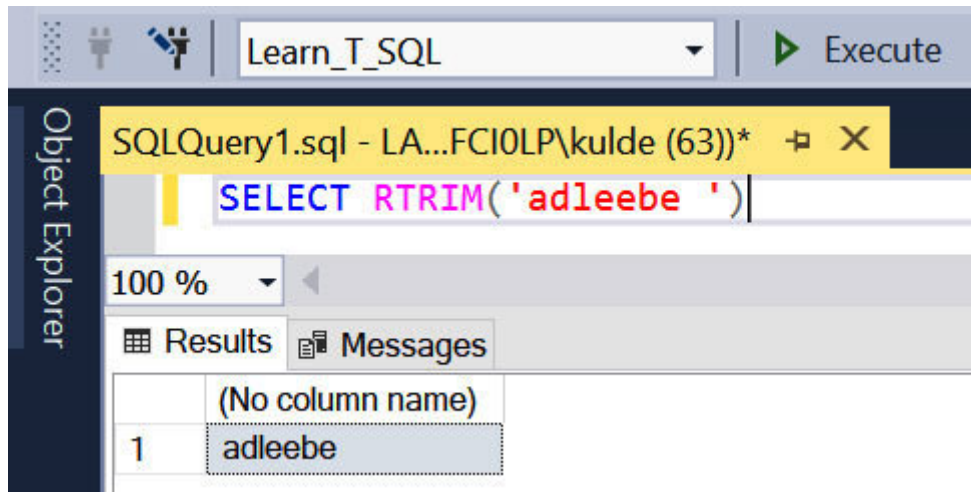


Figure 5.24: *RTRIM function*

LOWER ()

LOWER function is used to get the string in lower case.

The syntax is as follows:

LOWER ()

The explanation is as follows:

Suppose, we've a string **ABCDEF** and we want the string in lower case. Using the **LOWER** function will return We'll see the working of **LOWER** function using the following example.

The example is as follows:

SELECT

Upon running the preceding query will return the result as shown in the following screenshot:

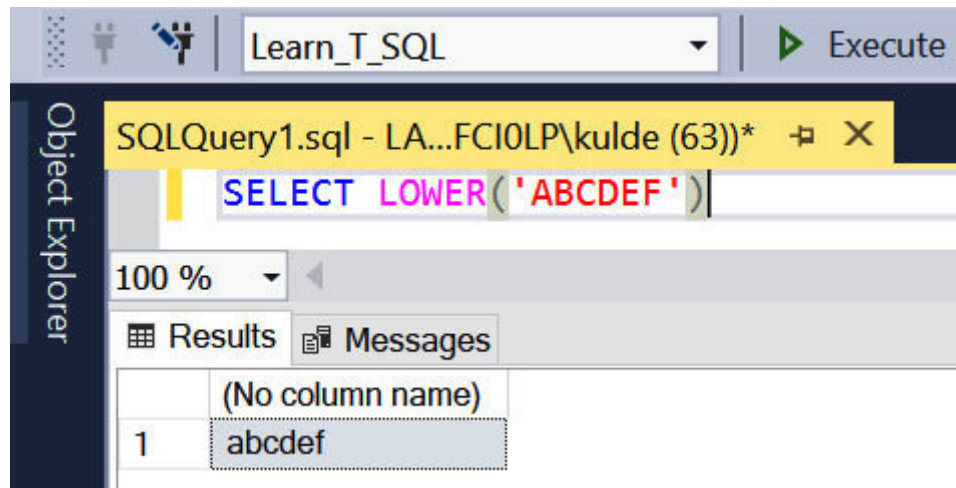


Figure 5.25: *LOWER function*

UPPER ()

UPPER function is used to get the string in upper case.

The syntax is as follows:

UPPER ()

The explanation is as follows:

Suppose we've a string **abcdef** and we want the string in upper case. Using the **UPPER** function will return We'll see the working of **UPPER** function using the following example:

The example is as follows:

SELECT

Upon running the preceding query will return the result as shown in the following screenshot:

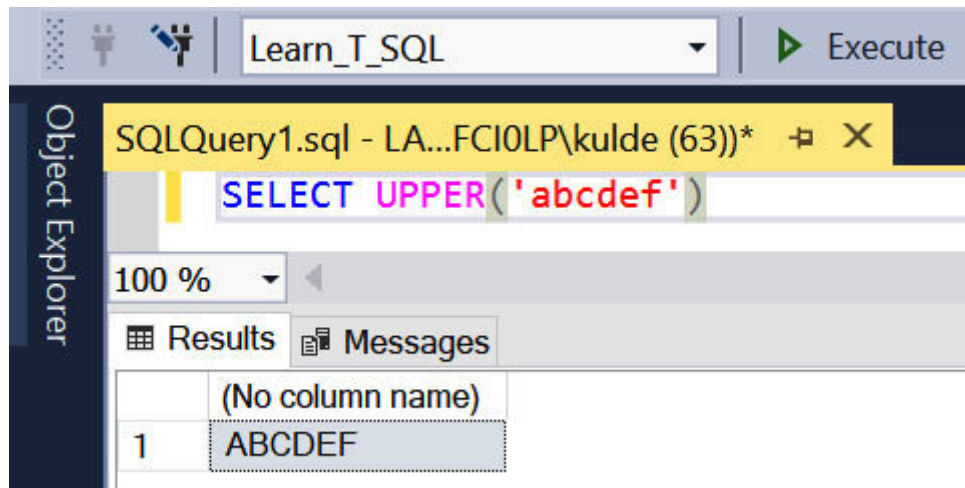


Figure 5.26: *UPPER function*

SPACE ()

SPACE function is used to generate the spaces without pressing the spacebar key. It accepts the number which represents the number of space characters to be generated.

The syntax is as follows:

SPACE (of space characters>)

The explanation is as follows:

Suppose we want to generate five space characters. So we need to supply **5** as a parameter to **SPACE** function and it will generate the string as ' '. We'll see the working of **SPACE** function using the following example.

The example is as follows:

SELECT

Upon running the preceding query will return the result as shown in the following screenshot:

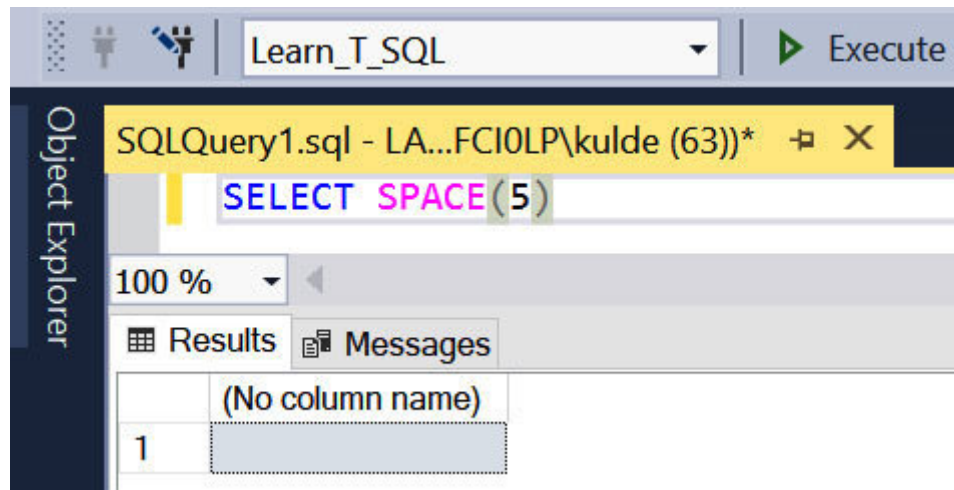


Figure 5.27: *SPACE function*

REPLICATE ()

REPLICATE function is used to replicate the specified character(s) for the specified number of times.

The syntax is as follows:

REPLICATE (, of times to be replicated>)

The explanation is as follows:

Suppose we want to replicate **abc 10** time then using the **REPLICATE** function will return We'll see the working of **REPLICATE** function using the following example.

The example is as follows:

SELECT

Upon running the preceding query will return the result as shown in the following screenshot:

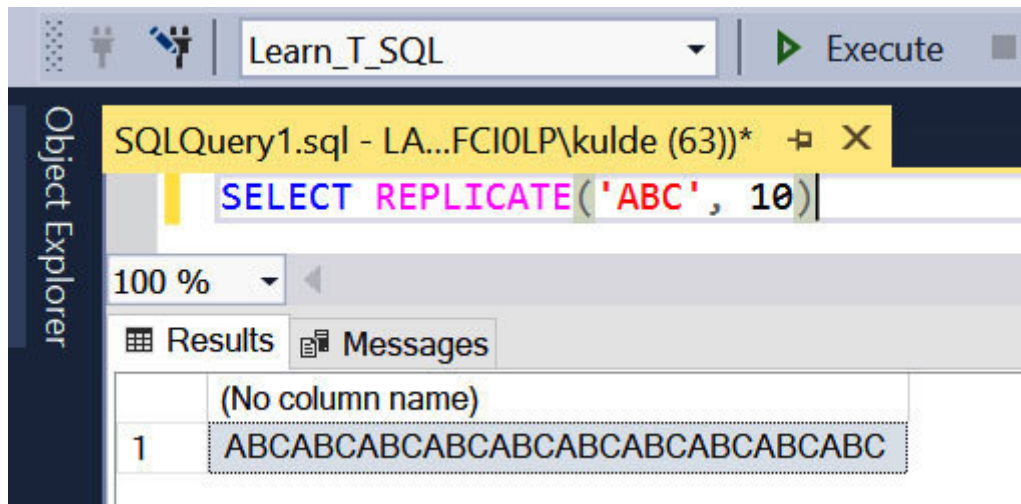


Figure 5.28: *REPLICATE function*

REVERSE ()

REVERSE function is used to reverse the specified string.

The syntax is as follows:

REVERSE ()

The explanation is as follows:

For example, if the string is **EDCBA** then using **REVERSE** function will return We'll see the working of **REVERSE** function using the following example.

The example is as follows:

SELECT

Upon running the query will return the result as shown in the following screenshot:

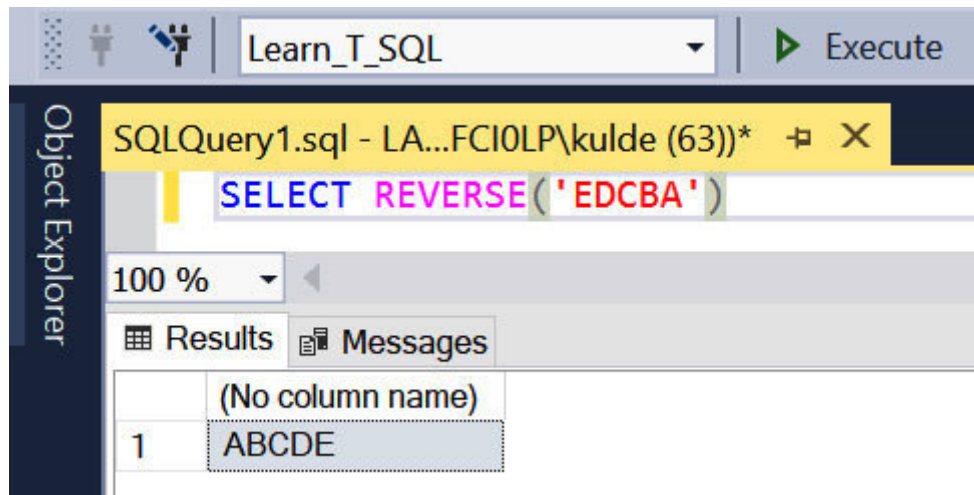


Figure 5.29: *REVERSE function*

CHARINDEX()

CHARINDEX function is used to find the index (position) of a character in a string. The function even has the flexibility to define the position (location) from where you would want to search the character.

The syntax is as follows:

CHARINDEX (to search>, , location>)

Explanation is as follows:

Let's say we've a string **EDCBAD** and we would like to find the index of character **D**. If you see we've two **D** characters, one at the second position and another at the sixth position. By default, **CHARINDEX** function will return 2. However, if we'll change the start location to some value below 6 and greater than the same function will return 6. We'll see the working of **CHARINDEX** function using the following examples.

Examples is as follows:

SELECT

SELECT

SELECT

SELECT

Upon running the preceding queries will return the result as shown in the following screenshot:

The screenshot shows the SQL Server Enterprise Manager interface. The top toolbar includes a dropdown menu set to 'Learn_T_SQL', an 'Execute' button, and other standard icons. The main query window displays four SQL queries using the CHARINDEX function to find the position of the character 'D' in the string 'EDCBAD'.

```
SELECT CHARINDEX('D', 'EDCBAD')
SELECT CHARINDEX('D', 'EDCBAD', 1)
SELECT CHARINDEX('D', 'EDCBAD', 3)
SELECT CHARINDEX('D', 'EDCBAD', 7)
```

Below the queries, the 'Results' tab is active, showing four result sets. Each result set has a single column labeled '(No column name)' and one row of data.

	(No column name)
1	2

	(No column name)
1	2

	(No column name)
1	6

	(No column name)
1	0

Figure 5.30: CHARINDEX function

QUOTENAME ()

QUOTENAME function is used to surround the supplied string with a square bracket. In SQL Server, generally, the object names such as table, column, stored procedure, function, view name, and so on are enclosed within a square bracket to avoid the name clash with reserved keywords.

Reserved keywords are the keywords used internally by SQL Server. You'll find these keywords in grey, pink, and blue colors when you'll type them in SSMS.

The syntax is as follows:

QUOTENAME ()

The explanation is as follows:

For example, if the string is **Customer** then using **QUOTENAME** function will return **[Customer]**. We'll see the working of **QUOTENAME** function using the following example.

The example is as follows:

SELECT

Upon running the query will return the result as shown in the following screenshot:

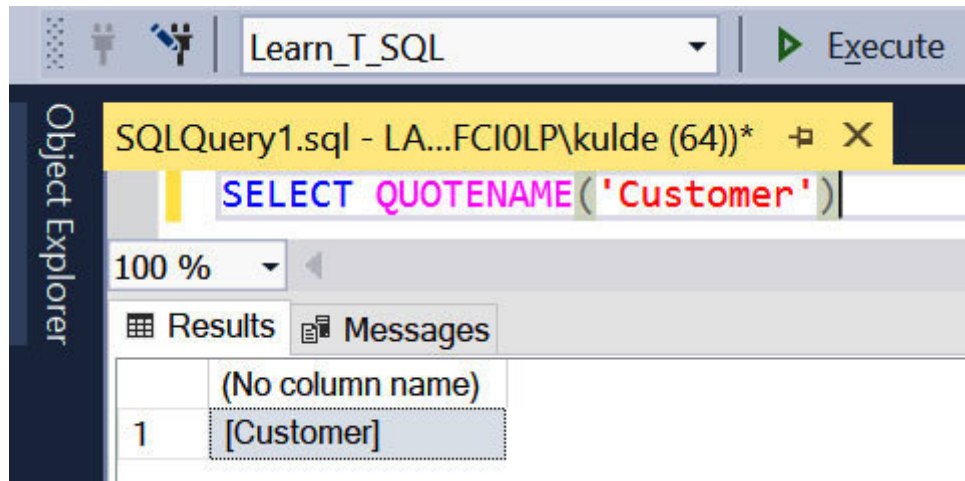


Figure 5.31: *QUOTENAME function*

STRING_SPLIT ()

STRING_SPLIT is a table valued function that returns a table. This function accepts a string containing the values separated by some separator (such as comma, vertical bar, and so on) as the first parameter and the separator itself as the second parameter. It then splits the values and returns the result in the form of a table. Each value is returned as a row.

Syntax is as follows:

STRING_SPLIT (,)

Explanation is as follows:

For example, if the string is **Apple|Ball|Cat** then using **STRING_SPLIT** function will return the table as shown here:

here:
here:
here:

Table 5.2: Table returned by *STRING_SPLIT* function

We'll see the working of **STRING_SPLIT** function using the following example.

Example is as follows:

```
SELECT * FROM
```

Upon running this query will return the result as shown in the following screenshot:

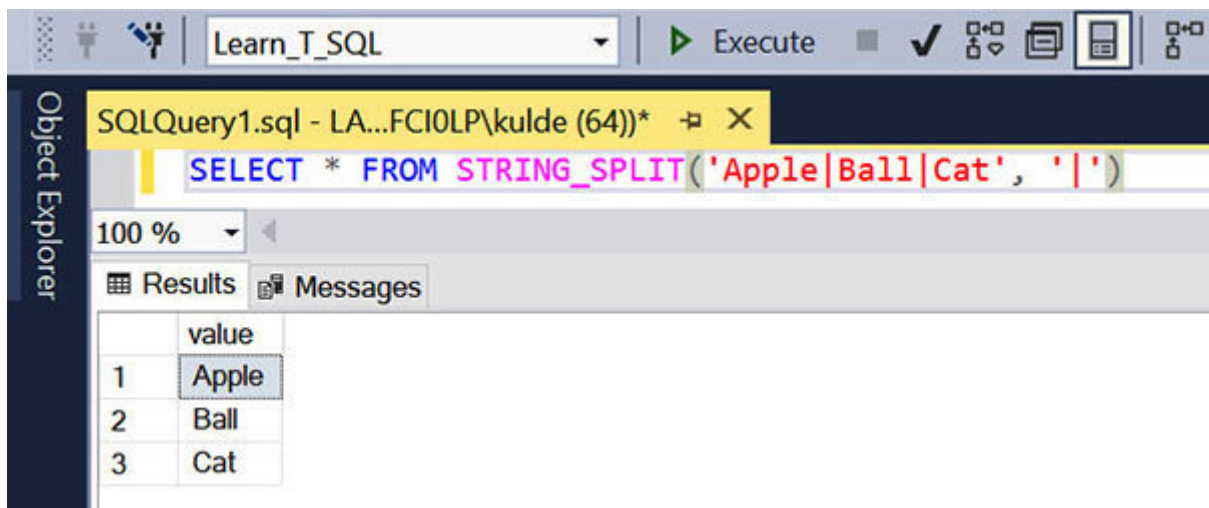


Figure 5.32: *STRING_SPLIT function*

Numeric functions

Numeric functions are the type of functions used with numeric data. There are various numeric functions. Although, we'll talk about the most used ones.

ABS ()

ABS function is used to get the absolute value of the supplied number.

The syntax of ABS function is as follows:

ABS ()

The explanation of **ABS** function is as follows:

For example, the absolute value of **10** is **10** and absolute value of **-10** is also

The example of **ABS** function is as follows:

SELECT

SELECT

Upon running these queries will return the result as shown in the following screenshot:

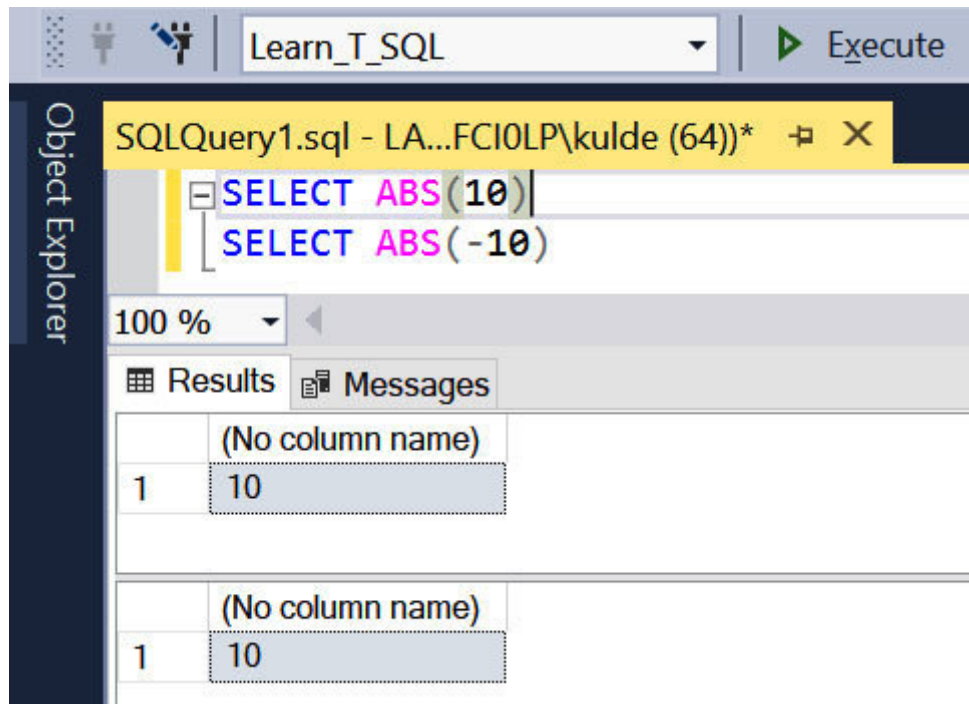


Figure 5.33: *ABS function*

ISNUMERIC ()

ISNUMERIC function returns 1 if the supplied value is a numeric value. Otherwise returns 0 in case of non-numeric value.

Syntax is as follows:

ISNUMERIC ()

Explanation is as follows:

For example, if the value is **99999** or any other numeric value (including negative ones), then **ISNUMERIC** function will return 1. If the value is non-numeric such as **2020-11-20** or **Being** the **ISNUMERIC** function will return 0.

Example is as follows:

SELECT

SELECT

SELECT

SELECT

Upon running these queries will return the result as shown in the following screenshot:

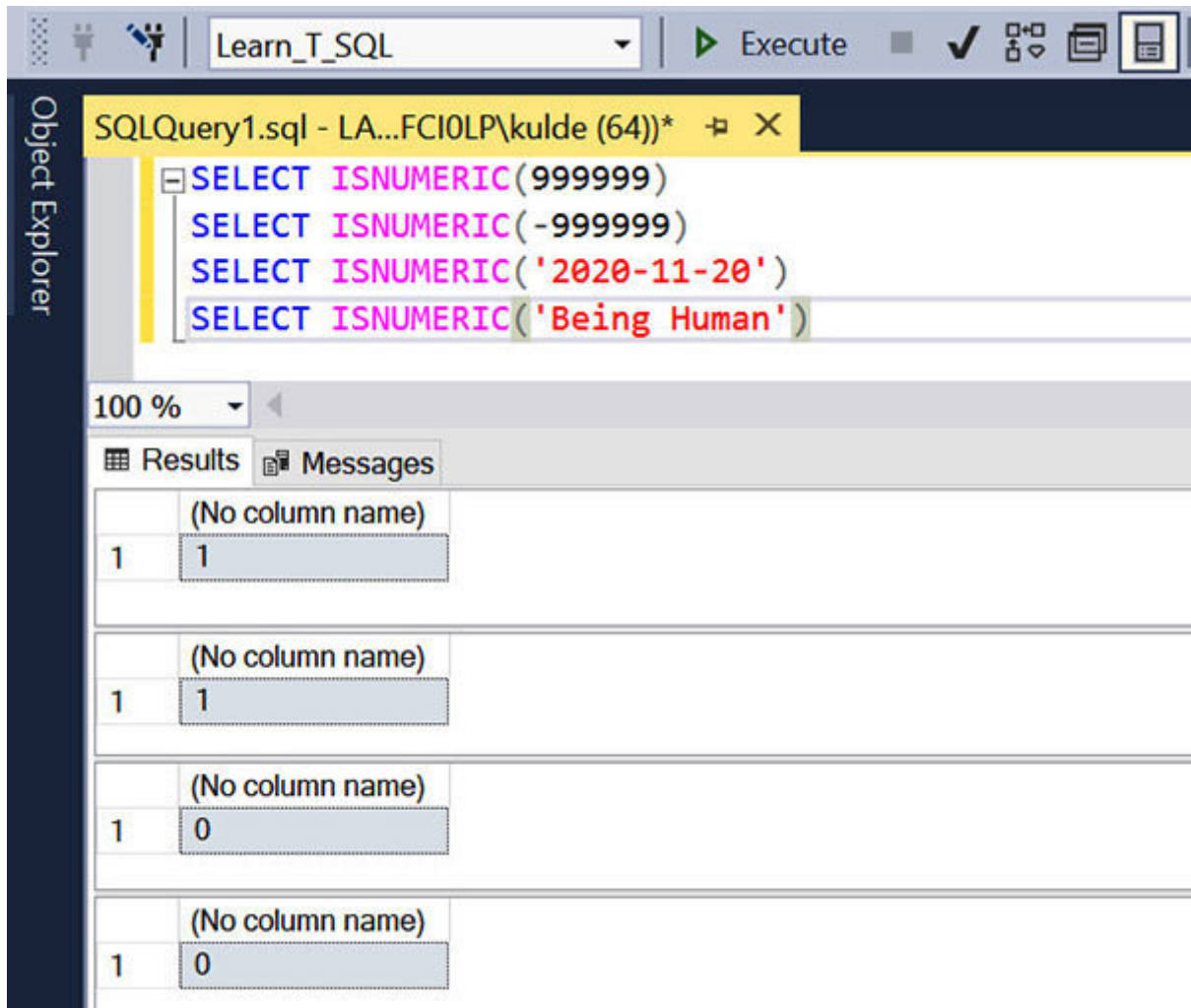


Figure 5.34: *ISNUMERIC function*

ROUND ()

ROUND function is used to round the supplied value to the supplied number of decimal places.

Syntax is as follows:

ROUND (, of decimal places>)

Explanation is as follows:

For example, if the number is **10.8756** and we wish to round it to two decimal places then the **ROUND** function will return **10.88**. Similarly, if the number is **10.4978** and we wish to round it to two decimal places then **ROUND** function will return **10.50**. We'll see the working of the **ROUND** function using the following examples:

The example is as follows:

```
SELECT ROUND(10.8756, 2);  
SELECT ROUND(10.4978, 2);
```

Upon running these queries will return the result as shown in the following screenshot:

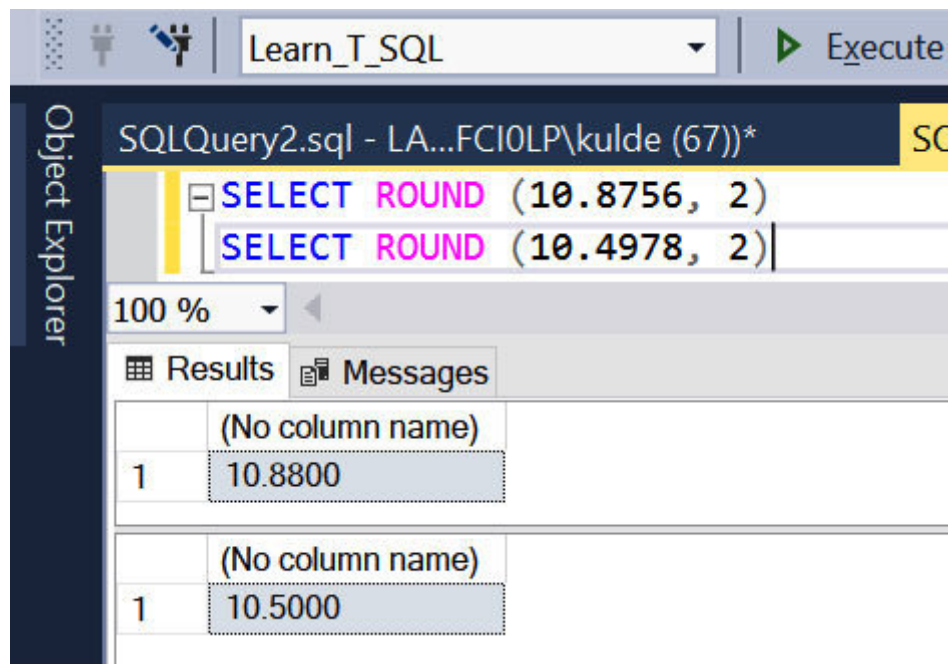


Figure 5.35: *ROUND function*

FLOOR () and CEILING ()

FLOOR function returns the largest integer value which is less than or equal to the supplied number. Integer is also a number without decimal places.

Whereas, **CEILING** function returns the smallest integer value which is greater than or equal to the supplied number.

The syntax is as follows:

FLOOR ()
CEILING ()

The explanation is as follows:

For example, if the number is **10.8756** then **FLOOR** function will return **10**. Whereas, **CEILING** function will return **11**.

The example is as follows:

SELECT FLOOR
SELECT CEILING

Upon running the queries will return the result as shown in the following screenshot:

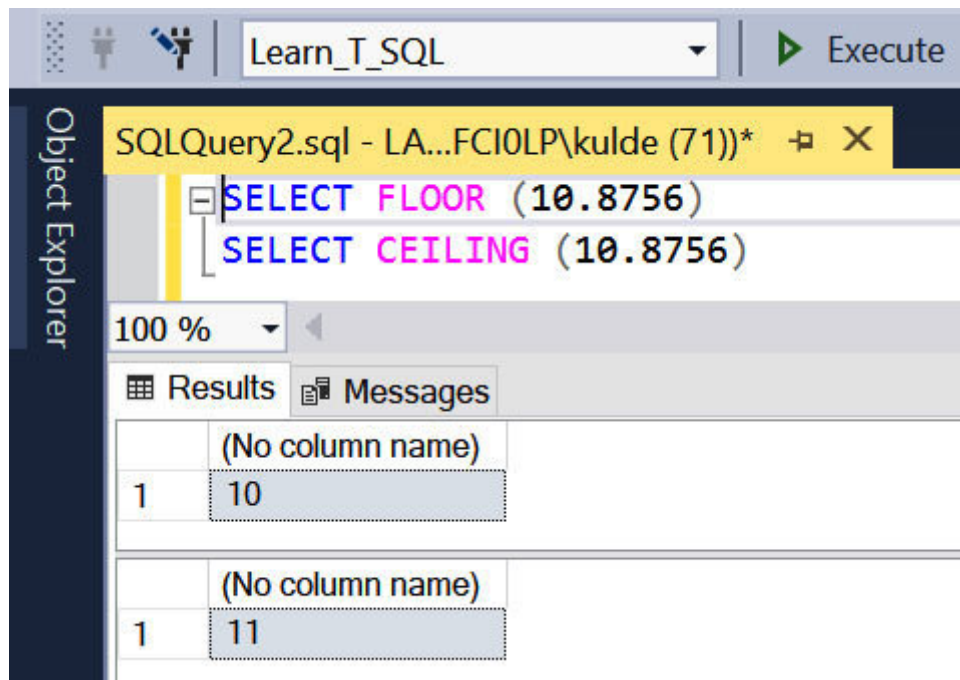


Figure 5.36: *FLOOR and CEILING functions*

POWER ()

POWER function is used to calculate the power of the supplied number to the supplied power value.

Syntax is as follows:

POWER (,)

Explanation is as follows:

Power of 2 is 4 ($2 * 2$) Power of 3 is 8 ($2 * 2 * 2$) Power of 10 is 1024 ($2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2$) When we calculate the power, there are two components involved – the number of which power to be calculate and the power value. If we take the example of then 2 is the number of which power to be calculated and 10 is the power value. The following is the T-SQL example for the **POWER** function.

Example is as follows:

SELECT POWER

Upon running this query will return the result as shown in the following screenshot:

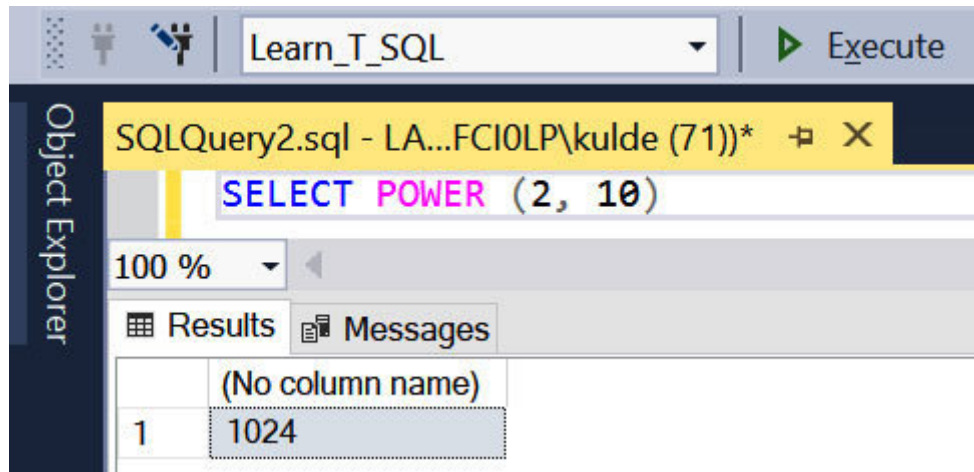


Figure 5.37: *POWER function*

RAND ()

RAND function is used to generate the random number. The random number generated through this function is a float value greater than 0 but less than 1. **RAND** function also accepts an optional parameter called the seed. If seed is not supplied then SQL Server assigns a random seed value. For a specified seed value, the result returned by the **RAND** function is always the same.

The syntax is as follows:

RAND ()

RAND (value>)

The example is as follows:

```
SELECT RAND ()
```

```
SELECT RAND
```

Upon running these queries will return the result as shown in the following screenshot:

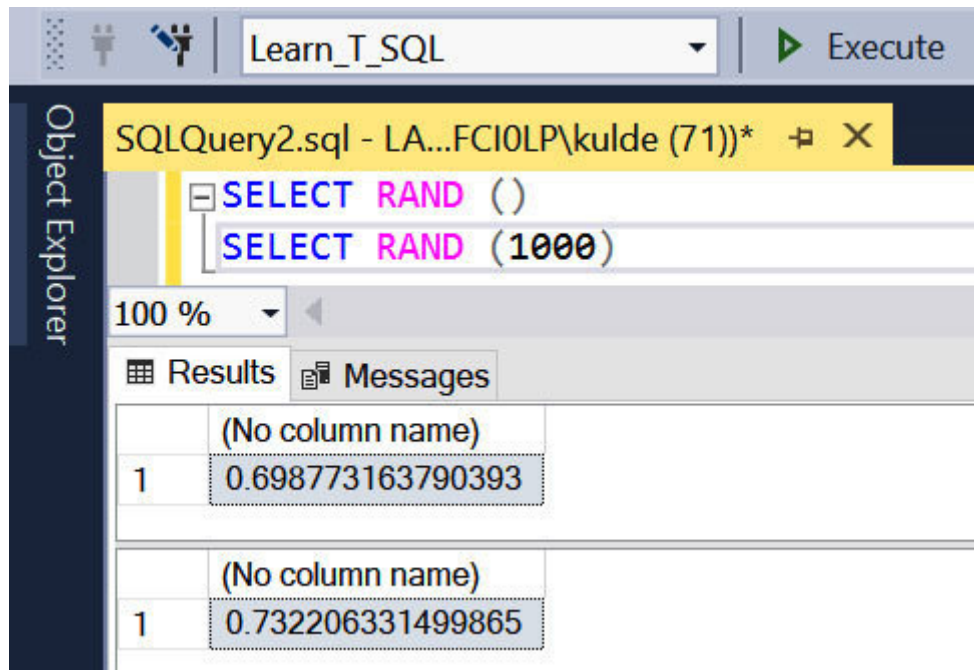


Figure 5.38: RAND function run 1

We'll run the same queries again to see what random value is generated. The following screenshot shows the random values generated. However, when you'll run it, you may get different values:

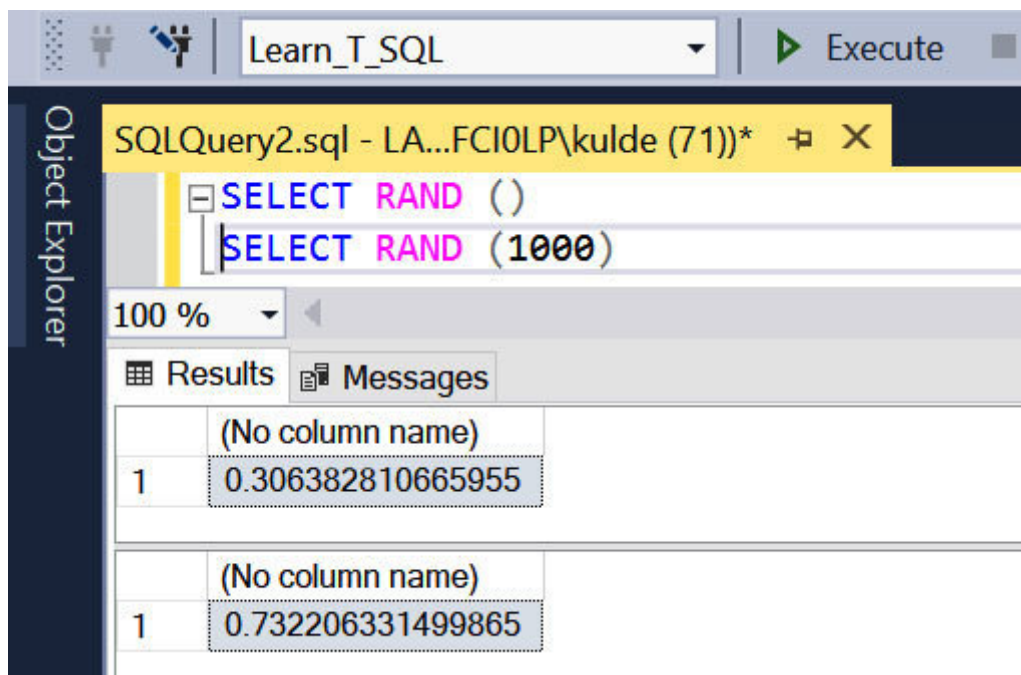


Figure 5.39: *RAND function run 2*

Follow the approach mentioned here to generate the random integer values:

Multiply the random float value generated by the **RAND** function with 10 and apply the **FLOOR** or **CEILING** function to get up to 1 digit random integer values. For example, **FLOOR(RAND() * 10)**. If you'll use the **CEILING** function you may also get 10 which is 2 digits integer value.

Multiply the random float value generated by the **RAND** function with 100 and apply the **FLOOR** or **CEILING** function to get up to 2 digits random integer values. For example, **FLOOR(RAND() * 100)**. If you'll use the **CEILING** function you may also get 100 which is 3 digits integer value.

Multiply the random float value generated by the **RAND** function with 1000 and apply the **FLOOR** or **CEILING** function to get up to 3 digits random integer values. For example, **FLOOR(RAND() * 1000)**. If you'll use the **CEILING** function you may also get 1000 which is 4 digits integer value.

Similarly, you can generate,

4 digits random integer values by multiplying with 10000.

5 digits random integer values by multiplying with 100000.

6 digits random integer values by multiplying with 1000000, and so on.

You can also use **CAST** and **CONVERT** functions (to convert the float value to integer) instead of **FLOOR** and **CEILING** functions. We'll talk about **CAST** and **CONVERT** functions in [Chapter 12, Data Conversion, Cross-Database, and Cross-Server Data](#)

Date functions

Date functions are the type of functions used with date and datetime data.

GETDATE ()

GETDATE function is used to get the current date and time. This function will return the current date and time of the server on which the SQL Server is running. Current date and time is returned in **yyyy-mm-dd hh:mm:ss:nnn** format.

yyyy is the year, **mm** is month, **dd** is the day, **hh** is hours, **mm** is minutes, **ss** is seconds, and **nnnn** is milliseconds.

The syntax of **GETDATE** function is as follows:

GETDATE ()

The example of **GETDATE** function is as follows:

SELECT

Upon running the query will return the result as shown in the following screenshot:

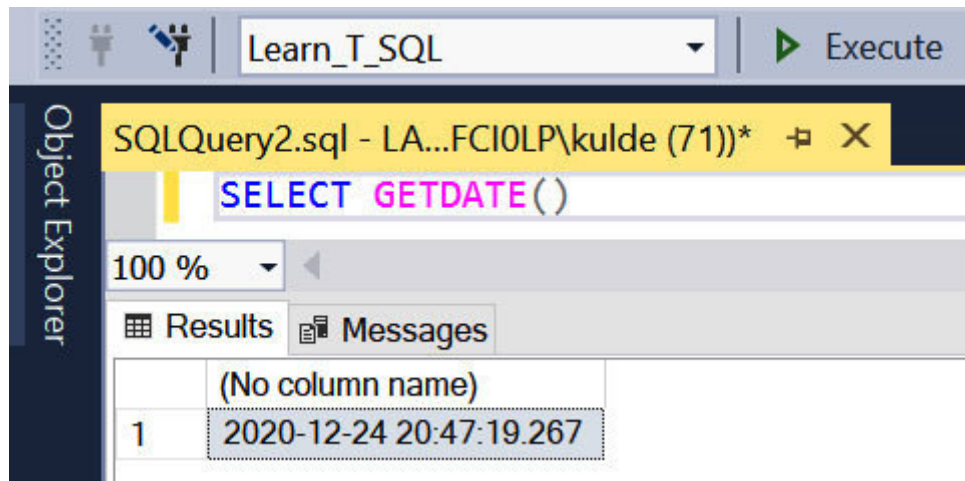


Figure 5.40: *GETDATE function*

ISDATE ()

ISDATE function returns **1** if the supplied value is a date or datetime value. Otherwise returns **0** in case of non-date or datetime value.

Syntax is as follows:

ISDATE ()

The explanation is as follows:

For example, if the value is **2020-12-25** or **2020-12-25 20:02:16.223** or any other date or datetime value then **ISDATE** function will return **1**. If the value is non-date or non-datetime such as **999999** or **Being** the **ISDATE** function will return **0**.

Example is as follows:

SELECT

SELECT

SELECT

SELECT

Upon running the preceding queries will return the result as shown in the following screenshot:

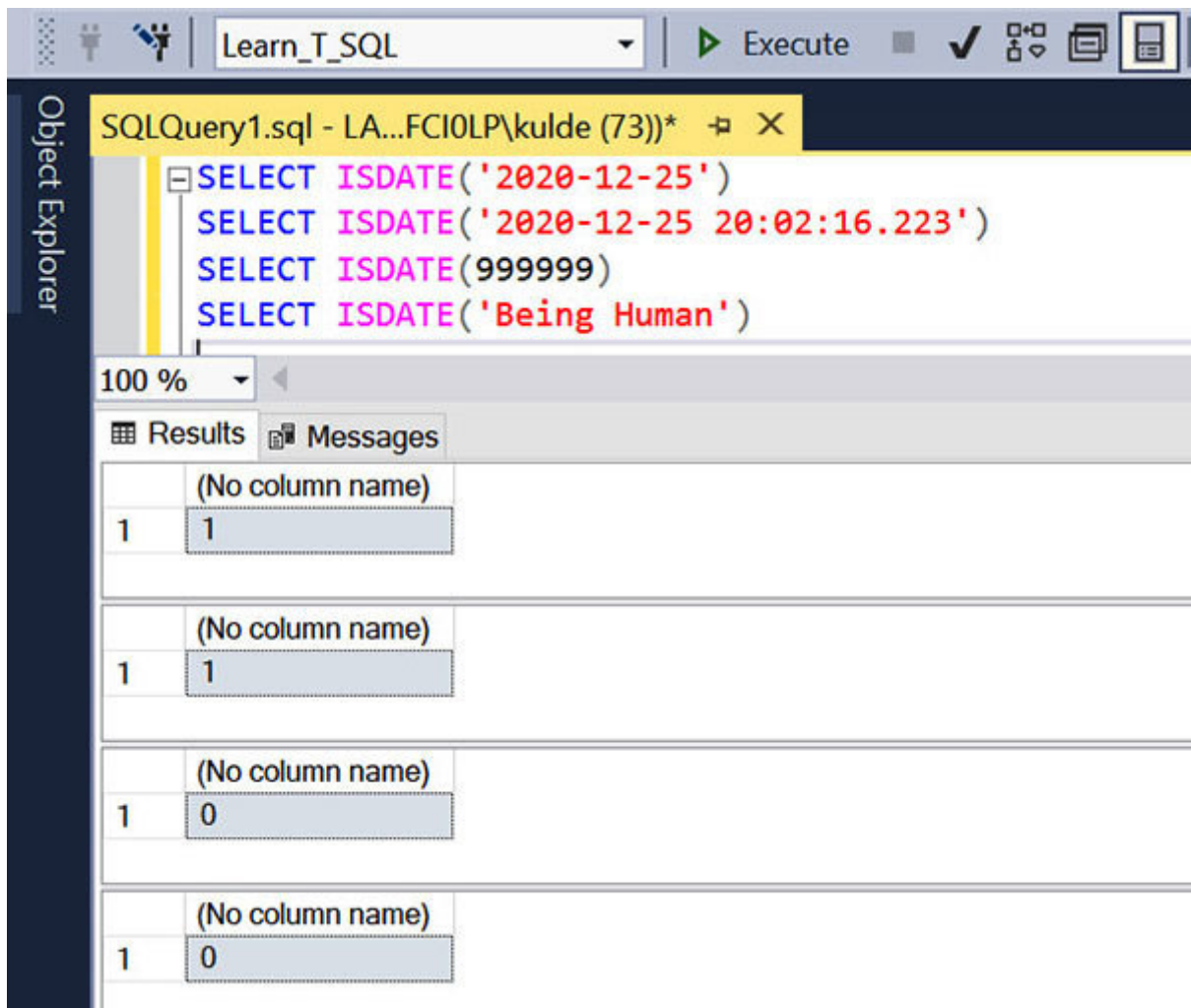


Figure 5.41: ISDATE function

DATEPART ()

DATEPART function is used to get the part of the supplied datetime value. As we all know the following forms the datetime. Each of them is considered as part, also known as the We can fetch either of these parts of datetime using the **DATEPART** function, but one at a time:

Day

Month

Year

Hour

Minute

Second

Millisecond

The syntax is as follows:

DATEPART (,)

DATEPART (, >)

We'll see examples both with datetime and date values individually. Following are the examples of **DATEPART** function with date time values.

Examples with date time values:

```
SELECT '2020-12-25'
```

```
SELECT '2020-12-25'
```

```
SELECT '2020-12-25'
```

```
SELECT '2020-12-25'
```

```
SELECT '2020-12-25'
```

```
SELECT '2020-12-25'
```

```
SELECT '2020-12-25'
```

Upon running the queries will return the result as shown in the following screenshot:

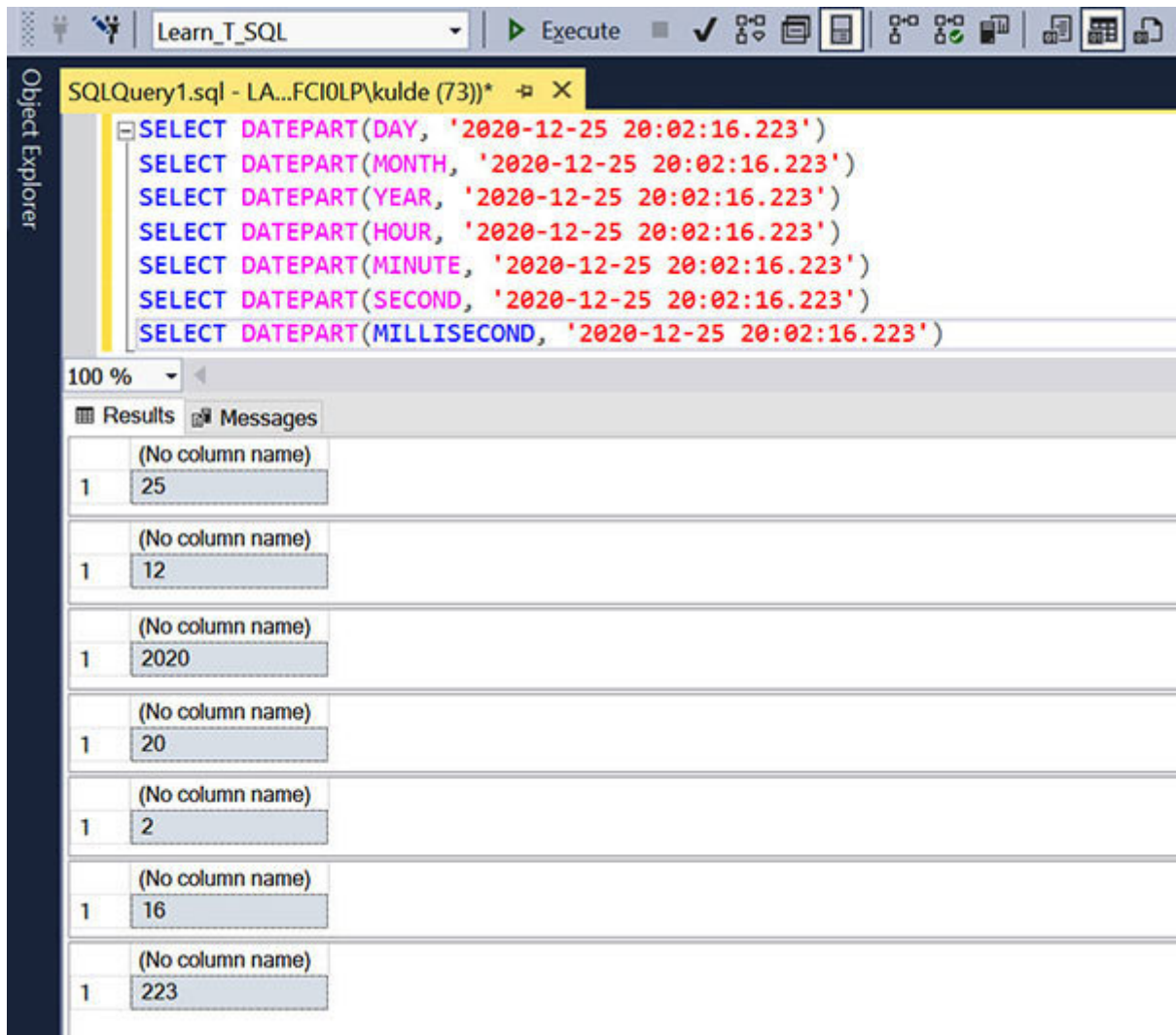


Figure 5.42: DATEPART function with date and time values

The following are the examples of **DATEPART** function with date values:

Examples with date value is as follows:

```

SELECT
SELECT
SELECT
SELECT
SELECT

```

SELECT

SELECT

Upon running these queries will return the result as shown in the following screenshot:

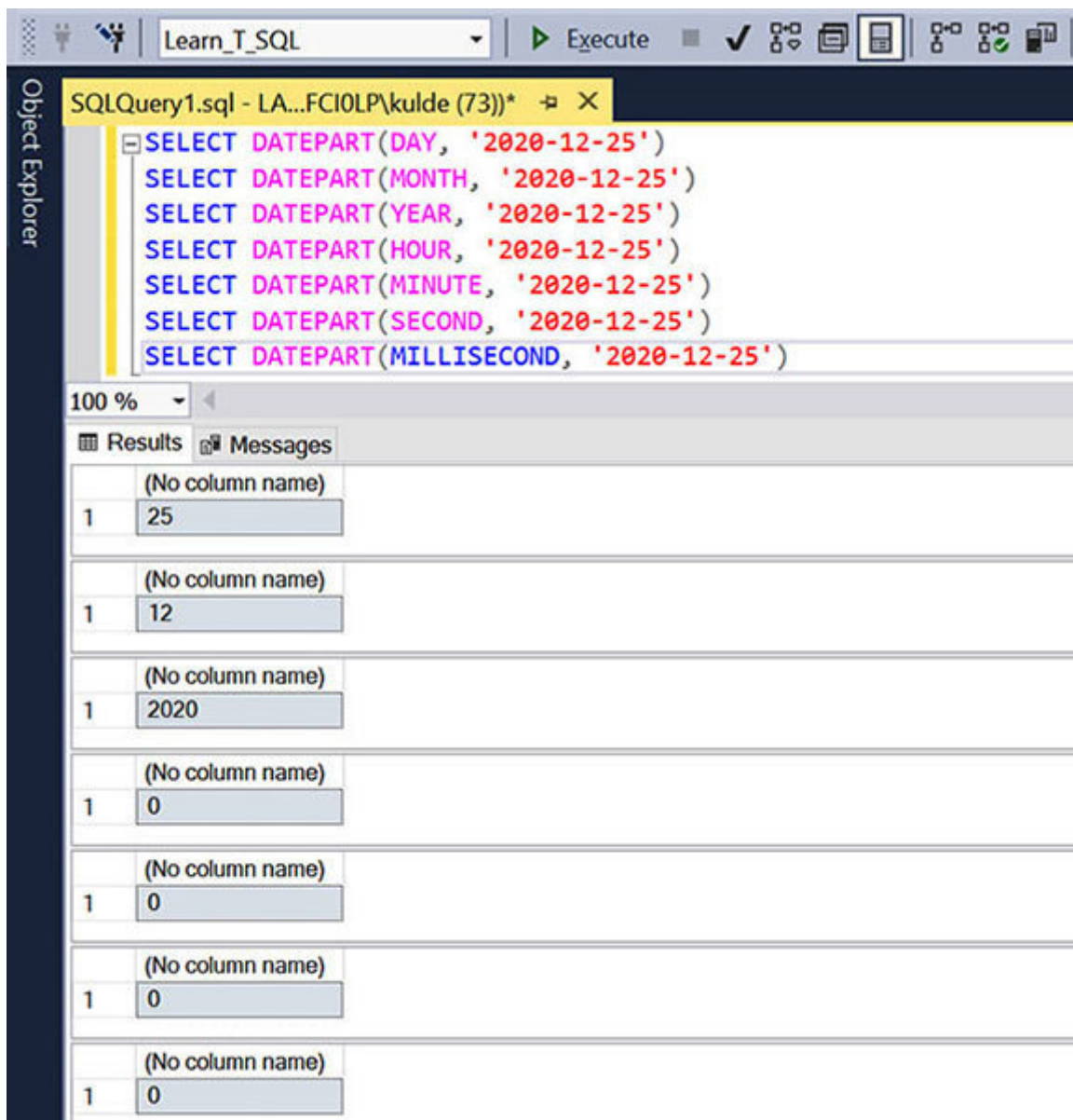


Figure 5.43: DATEPART function with date values

If you've noticed the difference between the results with datetime and date values, **DATEPART** function with date value returned 0 for time-related intervals such as hour, minute, second, and millisecond. You must be wondering *why*? The reason is simple because date value supplied to the **DATEPART** function does not contain the time.

DATEADD ()

DATEADD function is another useful function. It is used to add a specific intervals to the supplied datetime value. As we discussed in **DATEPART** function, the following forms the datetime. We can add either of these intervals using the **DATEADD** function, but one at a time:

Day

Month

Year

Hour

Minute

Second

Millisecond

The syntax is as follows:

DATEADD (, ,)

DATEADD (, ,)

You must be wondering *why I've mentioned two syntaxes, one with increment and another with decrement?* It is another beauty of **DATEADD** function. We can do both increment and decrement of interval in the supplied datetime value. The increment can be done by supplying a positive integer as increment. Decrement can be done by supplying negative integer as decrement.

For example, if you want to increment the day by two of the date **2020-12-23** then we'll use the following syntax and it will return

```
DATEADD (DAY, 2, '2020-12-23')
```

Similarly, if you want to decrement the day by two of the date **2020-12-23** then we'll use the following syntax and it will return

```
DATEADD (DAY, -2, '2020-12-21')
```

We'll see examples both with increment and decrement individually. The following are the examples of **DATEADD** function with increment.

Examples with increment:

SELECT
SELECT
SELECT
SELECT
SELECT
SELECT
SELECT

Upon running these queries will return the result as shown in the following screenshot:

The screenshot displays the SQL Server Enterprise Manager interface. The top toolbar includes the 'Execute' button. The query window shows the following SQL statements:

```
SELECT DATEADD(DAY, 1, '2020-12-25')
SELECT DATEADD(MONTH, 1, '2020-12-25')
SELECT DATEADD(YEAR, 1, '2020-12-25')
SELECT DATEADD(HOUR, 1, '2020-12-25')
SELECT DATEADD(MINUTE, 10, '2020-12-25')
SELECT DATEADD(SECOND, 50, '2020-12-25')
SELECT DATEADD(MILLISECOND, 100, '2020-12-25')
```

The 'Results' tab is active, showing a grid with seven rows of data. Each row has a single column with the value '1' and a corresponding date and time string.

	(No column name)
1	2020-12-26 00:00:00.000
1	2021-01-25 00:00:00.000
1	2021-12-25 00:00:00.000
1	2020-12-25 01:00:00.000
1	2020-12-25 00:10:00.000
1	2020-12-25 00:00:50.000
1	2020-12-25 00:00:00.100

Figure 5.44: *DATEADD function with increment*

The following are the examples of **DATEADD** function with decrement.

Examples with decrement:

```
SELECT  
SELECT  
SELECT  
SELECT  
SELECT  
SELECT  
SELECT
```

Upon running the preceding queries will return the result as shown in the following screenshot:

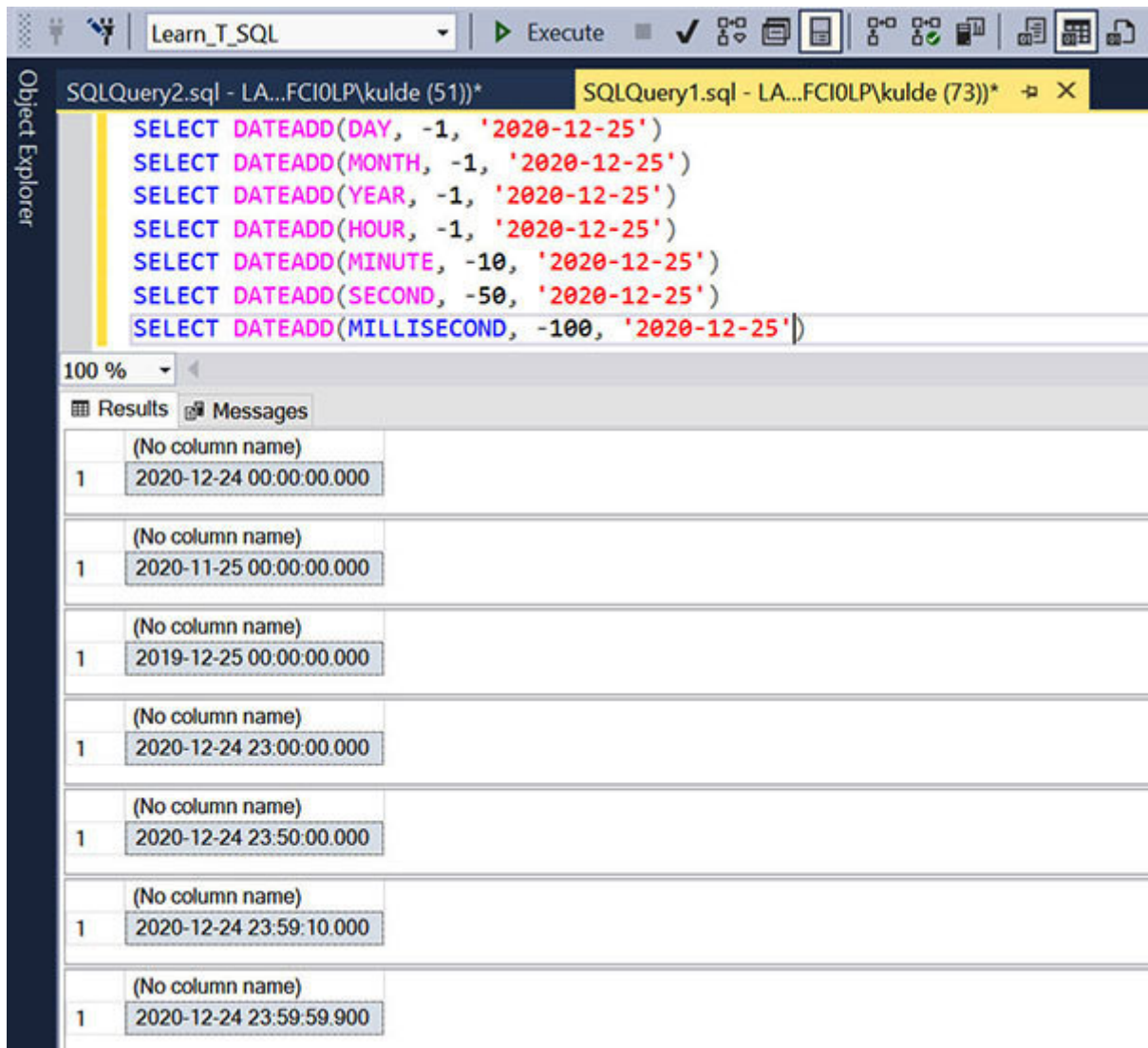


Figure 5.45: DATEADD function with decrement

DATEDIFF ()

DATEDIFF function is another useful function, used to get the difference of the supplied interval, between two supplied datetime values. The interval difference of either of the following intervals can be calculated with the help of **DATEDIFF** function, but one at a time:

Day

Month

Year

Hour

Minute

Second

Millisecond

The syntax is as follows:

DATEDIFF (, datetime>, datetime>)

DATEDIFF (, date >, date >)

As we've already seen, the working of **DATEADD** and **DATEPART** functions are the same with both date and datetime values. The same goes with **DATEDIFF** function too. The behavior of **DATEDIFF** function is the same with both date and datetime values. We'll see the example only with date value and is mentioned as follows.

The example is as follows:

SELECT

SELECT

SELECT

Upon running these queries will return the result as shown in the following screenshot:

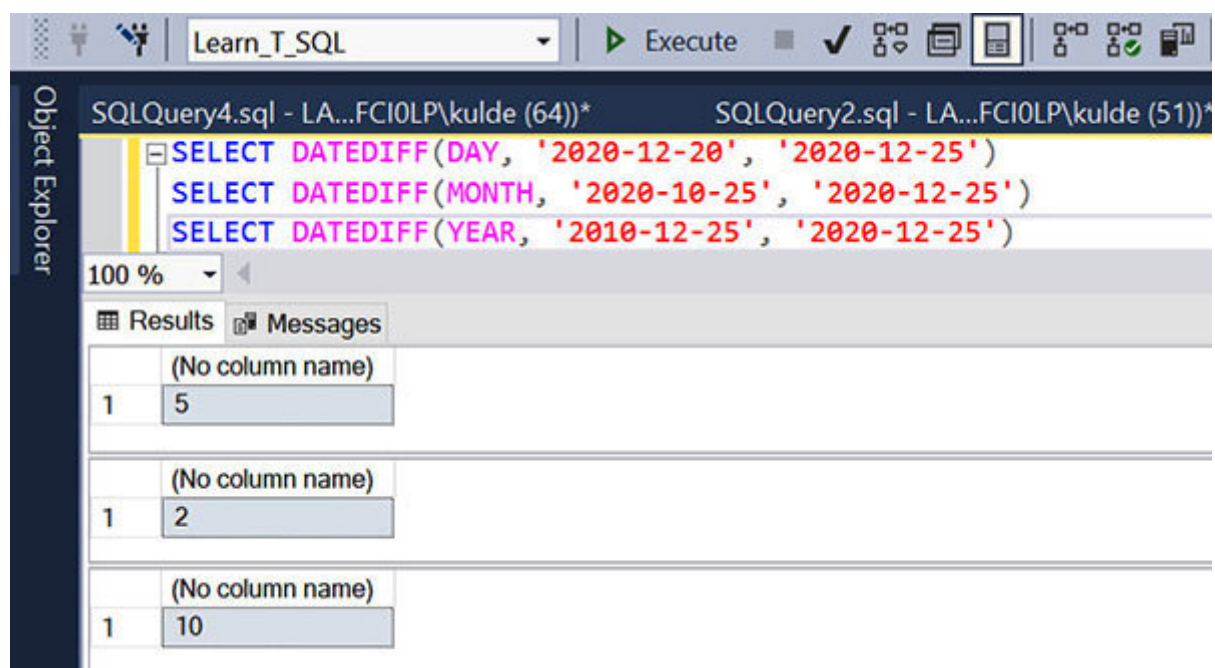


Figure 5.46: *DATEDIFF function*

Similarly, the difference of other intervals such as hour, minute, second, and millisecond can be derived using the **DATEDIFF** function.

Conclusion

We learnt various important built-in functions offered by SQL Server. They come in handy and are very useful when you are working with T-SQL queries. We learnt about various aggregate functions, numeric functions, date functions, and string functions. All of them play very vital roles while working with data and performing the analytics using T-SQL.

All of these functions discussed in this chapter can be used with both the individual values as well as with the columns. You can even use these functions in **SELECT** and **UPDATE** statements. For example, you can use **DATEDIFF** function to calculate the date difference between two date columns of a table. You can use **ISNUMERIC** function to check if the column value is numeric or not. Similarly, other functions can also be used with the columns as well.

So far, we have learnt various things including in previous chapters, like creating the table, DML to insert, update, delete, and retrieve the plain data. In this chapter, we learned about various built-in functions and moved one ladder up in the journey of learning T-SQL, but so far, we have learnt to query only one table.

In the next chapter, you'll learn to combine the multiple tables and to query them using join, apply, and subquery to

get the unified result of multiple tables.

Points to remember

Function always returns a value and it should always be followed by parenthesis. The parameters to the functions have to be supplied within parenthesis.

String functions are used with the string data.

Numeric functions are used with the numeric data.

Date functions are used with the date and datetime data.

Aggregate functions are used to perform aggregation such as summation, average, count, minimum, and maximum.

Aggregate functions, string functions, numeric functions and date functions discussed in this chapter can be also used in **SELECT** and **UPDATE** statements.

Built-in functions used in this chapter can be used with the individual values as well as with columns too. Column(s) can be supplied as the parameter to the function, if it accepts the parameter, and the data type of both column(s) as well as the parameter is the same.

The terminologies *clause* and *command* are used interchangeably and have the same meaning.

All the operators you can use with the **WHERE** clause can also be used with the **HAVING** clause.

Multiple choice questions

There is a string Bring me You want two characters from the character onward, which of the following functions would you use?

QUOTENAME()

RIGHT()

SUBSTRING()

RIGHT()

You got a requirement to return the first three characters of a column, which of the following functions would you use?

STRING_SPLIT()

LEFT()

LTRIM()

REPLACE()

There are two columns in a table, both are of datetime data type. You need to get all the rows wherever the day difference between these two columns is greater or less than Which of the following function shall you use in WHERE clause?

DATEDIFF()

DATEADD()

ISDATE()

GETDATE()

Answers

C

B

A

Questions

Which function is used to add interval to datetime value or column?

Can we supply a parameter to **GETDATE()** function?

How to get the minutes from a datetime value or column?

Which function is used to check if a value is numeric or not?

Which function can be used to return the comma-separated values as a table where each value will be returned as a row?

How to get the number of characters in a string?

How can we get the position of a character in a column?

What is the difference between **LTRIM** and

There is string We want output as Which function to use?

Can we supply **GETDATE()** to **DATEADD()** function?

Key terms

GROUP BY is the instruction in T-SQL to instruct to perform grouping on set of columns. You can use **GROUP BY** only with the **SELECT** command. **GROUP BY** command is must in every query using aggregate functions on columns with other columns too used without aggregate functions. If only column(s) with aggregate function exists in the **SELECT** statement then **GROUP BY** is not needed.

HAVING is used to filter the aggregate result. Its purpose is similar to **WHERE** command, but it can be used only with aggregate functions. You can't filter the aggregate result using **WHERE** clause. You need **HAVING** clause to filter it.

CHAPTER 6

Join, Apply, and Subquery

We learnt to create tables and to query them in the previous chapters. However, so far, we have queried only a single table. When we talk about a database, it's more than just querying a table. A database has many tables, and there is a need to combine the data from multiple tables and present a single unified result.

Join is a feature of SQL Server that allows us to join multiple tables, and return the relevant rows and columns from multiple tables, with the help of various types of joins available. There are few other features as well such as **APPLY** clause and subquery that also plays crucial roles when it comes to querying multiple tables. We'll understand all these features in this chapter. At the end of the chapter, you'll be able to query multiple tables and you'll reach one ladder up in the process of learning T-SQL.

Structure

In this chapter, we will cover the following topics:

JOIN

VLOOKUP in Excel versus JOIN

INNER JOIN

OUTER JOIN

LEFT OUTER JOIN

RIGHT OUTER JOIN

FULL OUTER JOIN

APPLY

CROSS APPLY

OUTER APPLY

Subquery

Objective

We'll discuss the various joins and other features of SQL Servers such as **APPLY** clause and subqueries to query multiple tables using a single **SELECT** query. We'll also learn to and **DELETE** records based on some matching conditions between various tables.

In short, at the end of this chapter, you'll be comfortable enough to deep dive into T-SQL and write more complex queries to get the combined result from multiple tables. In the next [Chapter 7, Built-In Functions - Part](#) we'll explore more advanced built-in functions to gain further knowledge to perform complex analytics.

JOIN

JOIN clause is used to combine rows from two or more tables based on related columns of both tables.

Let’s look at the following example. This is the same example we’ve discussed in [Chapter 4](#), If you remember, all the following tables are available in our database

Table 6.1: Customer table

table
table
table
table

Table 6.2: Product table

table
table

table
table
table
table

Table 6.3: *PurchaseOrder table*

If you closely look at the **PurchaseOrder** table then it has **CustomerID** instead of customer attributes such as name, address, contact, and so on. However, the **Customer** table has all these attributes. Similarly, there is a separate table Product holding the product details. **PurchaseOrder** table is referencing the **ProductID** of Product table.

Now, suppose you've been asked to write a query to get the following columns together. This is where **JOIN** comes into play:

OrderID

TransactionDate

CustomerName

ProductName

Quantity

Rate

Amount

As we already discussed, the **JOIN** clause is used to combine rows from two or more tables based on the related column(s) of both tables, that is, **JOIN** clause demands to mention the condition based on the related column(s) of tables. In the case of aforesaid example, **PurchaseOrder** table will be joined with the **Customer** table using **CustomerID** as the related column.

Similarly, there will be another join between the **Product** and **PurchaseOrder** table based on **ProductID** column, which is related column in both tables. So, technically, there will be two joins – one between **PurchaseOrder** with **Customer** and another between **PurchaseOrder** with

We'll discuss the joins in details as we'll dwell further in the chapter. However, let's understand it with a real-world example, even in non-relational database world. If you are familiar with Microsoft Excel then you would have heard of Let's have a look at how **VLOOKUP** and **JOIN** are related.

VLOOKUP in Excel vs JOIN

We'll be focused on our goal to learn the T-SQL. We'll not deviate from this. The purpose behind talking about the **VLOOKUP** is to teach you T-SQL joins quickly by leveraging your knowledge of Excel.

If you are not aware of **VLOOKUP** then you can learn about it at the following URL:

<https://support.microsoft.com/en-us/office/vlookup-function-0bbc8083-26fe-4963-8ab8-93a18ad188a1>

We'll replicate the same example of and **PurchaseOrder** in Excel to see how it works with In the Excel file, we'll place all these three tables in three different sheets. The following screenshot shows the first sheet, which contains the **Customer** table:

A2				
	A	B	C	D
1	CustomerID	CustomerName	CustomerAddress	CustomerMobile
2	1	John	Paris	1111111111
3	2	Kiya	London	2222222222
4	3	Reema	India	3333333333
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				
15				
16				
17				
18				
19				
20				
21				
22				
23				
	Customer	Product	PurchaseOrder	

Figure 6.1: Customer sheet

The following screenshot shows the second sheet which contains the **Product** table:

B1	:	✕	✓	fx	ProductNam
	A	B	C	D	
1	ProductID	ProductName			
2	1	Electra			
3	2	Thunderbird			
4	3	Classic			
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					
16					
17					
18					
19					
20					
21					
22					
23					
		Customer	Product	PurchaseOrder	

Figure 6.2: Product sheet

The following screenshot shows the third sheet which contains the **PurchaseOrder** table. If you see the *yellow* highlighted columns in the following screenshot, then you can see **CustomerName** and **ProductName** columns. It is to be noted that these columns do not contain the typed data. These columns are derived with the help of **VLOOKUP** formula and here is the formula used. We'll discuss the **VLOOKUP** formula for each of the columns separately.

Formula for `VLOOKUP(C2, Customer!$A$1:$D$4, 2, FALSE – Exact match)`.

Formula for `Product!$A$1:$B$4, 2, FALSE – Exact match)`.

The explanation of **VLOOKUP** formula is as follows:

We've used the **VLOOKUP** function in two columns. We'll discuss the formula for The explanation would remain the same even in the case of with only difference in the parameter values:

C2 is the first parameter of the **VLOOKUP** function. It is the **CustomerID** column from the **PurchaseOrder** sheet as can be seen in the following screenshot. The first parameter of **VLOOKUP** is called as As the name suggests, it's the value to be searched.

Customer!\$A\$1:\$D\$4 is the second parameter of the **VLOOKUP** function. The second parameter of **VLOOKUP** is called as This is the table/array of columns where the **lookup_value** will be searched. Parameter value **Customer!\$A\$1:\$D\$4** denotes **A1** to **D4** column of **Customer** sheet as can be seen in [figure](#) **A1** denotes *column A* and *row* **D4** denotes *column D* and *row*

\$ denotes that the formula is freezed and when it'll be copied to another cell, the formula won't change. This is what we need. Lookup value should change for each row but the table array should not. That is why the first parameter does not have the **\$**. So **C2** denotes the first row of *column* You must be wondering *why so?* Because the first row contains the header. The data row starts in excel from the second row onwards.

2 is the third parameter of the **VLOOKUP** function. The third parameter of **VLOOKUP** is called as It is the column number of the column that is to be fetched, if there is a match of the lookup value. **Col_index_number** starts with It pertains to the **table_array** that is the second parameter. Since in our example, the **table_array** is **A1:D4** so there are four columns and **Col_index_number** will be **1** for **2** for for and **4** for

Exact match is the fourth and last parameter of the **VLOOKUP** function. Forth parameter of **VLOOKUP** is an optional one, which means the function will work even if it's not supplied. The default value of this parameter is **FALSE** – exact match. It denotes the lookup value should match exactly in the table array. Other value is **True** – approximate match.

As the name suggested this option can be used if the approximate match is desired. Generally, the default option that is **FALSE** – the exact match is used widely:

Order ID	Transaction Date	Customer ID	Product ID	Quantity	Rate	Amount	Customer Name	ProductName
1	01-Apr-18	1	1	1	1,80,000.00	1,80,000.00	John	Electra
2	10-Apr-19	2	2	1	2,20,000.00	2,20,000.00	Kiya	Thunderbird
3	20-May-19	3	3	1	2,00,000.00	2,00,000.00	Reema	Classic
4	18-Jun-19	1	2	1	2,25,000.00	2,25,000.00	John	Thunderbird
5	20-Dec-19	1	3	1	2,10,000.00	2,10,000.00	John	Classic

Figure 6.3: PurchaseOrder sheet

The following screenshot shows the **VLOOKUP** formula for the **CustomerName** column of the **PurchaseOrder** sheet:


```
,
1 alias>.name>
,
2 alias>.name>
,
2 alias>.name>
FROM
1 name>
1 alias>

| LEFT | RIGHT | JOIN
```

```
2 name>
2 alias>
ON
1 alias>.name>
2 alias>.name>
|
1 alias>.name>
2 alias>.name>
WHERE
1 alias>.name>
|
2 alias>.name>
```

The explanation of the syntax is as follows:

SELECT is the keyword which is an instruction to retrieve the records.

Alias are optional but are generally used to define the name of each table in the query. It helps when there are multiple tables in the query by the way join. When we define the alias for the tables

then instead of writing the table name we can use the **alias.column** name to refer to the column of the respective table. You can give any name as alias but it's recommended to keep it short.

FROM is the keyword that is used to supply the table name from which the records are to be retrieved.

name> has to be replaced with the actual table name.

JOIN is the keyword used to join two tables. There are four types of joins such as and You can specify any of these as desired. However, if the type of join is not specified then SQL Server will consider it as default join which is **INNER**

ON is the keyword used to specify the **JOIN** condition. operators can be used to add multiple join conditions.

WHERE is the keyword used to filter the data. List of operators that can be used with **WHERE** clause is discussed in [Chapter 4](#).

AND/OR operators to combine multiple filter conditions.

GROUP BY as we already discussed in previous chapters can be used in the same manner as we've already learnt.

INNER JOIN

INNER JOIN can be understood with the help of the following screenshot. You could see two circles **A** and **B** intersecting each other. Each circle represents a table. Circle-A represents **Table-A** and circle-B represents **Table-B**. As we've already understood in the preceding topic *JOIN is always between 2 tables at a time*. However, you can JOIN multiple tables.

The intersection of both the circles (as highlighted) in the following screenshot represents the INNER JOIN. In other words, the intersection of two tables is called **INNER**. It returns the matching records from both tables, participated in the JOIN:

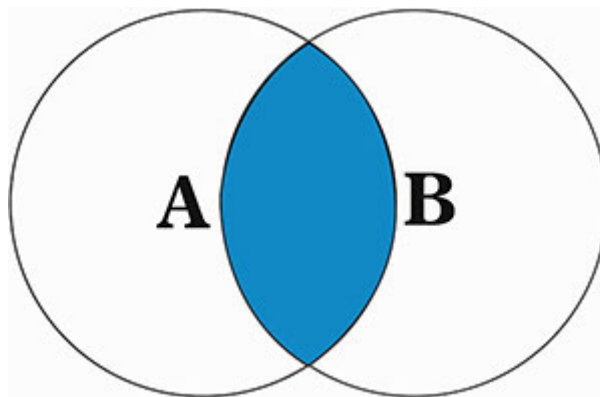


Figure 6.6: Inner Join

We'll understand the INNER JOIN with three scenarios – **one-to-one**, **one-to-many** and **many-to-many**. Understand them

carefully since these will be referred all across the various joins. Although, the behavior of different joins would be different but the concept learnt here would apply everywhere.

Following is the scenario of one-to-one relationship.

One-to-one relationship

If *Table-A* has 10 records and *Table-B* has 8 records then INNER JOIN will return the 8 considering the column value used for JOIN presents in both tables and has unique values in both tables. It means there has to be one to one relationship. It can be understood with the following example.

Table 6.4 can be related to which has two columns - **ID** and ID is the primary key:

[illegible]

Table 6.4: Table-A for one-to-one relationship

[Table 6.5](#) can be related to which has two columns as well - **ID** and ID is foreign key:

key:
key:
key:

key:
key:
key:
key:
key:
key:

Table 6.5: Table-B for one-to-one relationship

If you'll observe then *Table-A* has 10 rows of 10 unique whereas *Table-B* has 8 rows of 8 unique You would not see any duplicate values in *Table-B* that is the foreign key table. Primary key table can never have duplicate values. This is one-to-one relationship. In this case, INNER JOIN will return 8

The following is the scenario of one-to-many relationship.

One-to-many relationship

Similarly, if there is one-to-many relationship in which the foreign key table has duplicate values in the column used for JOIN, then the number of records will be a number of records in the foreign key table. It is to be noted that the number of records referred belongs to the column values which is common across both tables. It can be understood with the following example.

Table 6.6 can be related to which has two columns - **ID** and ID is the primary key:

[illegible]

key:

Table 6.6: Table-A for one-to-many relationship

[Table 6.7](#) can be related to which has two columns as well - **ID** and ID is foreign key:

key:
key:

key:
key:
key:
key:
key:
key:
key:
key:
key:

Table 6.7: Table-B for one-to-many relationship

If you'll observe then *Table-A* has 10 rows of 10 unique whereas *Table-B* has 10 rows of 8 unique You can see two duplicate values in *Table-B* that is the foreign key table for ID

as 3 and The primary key table can never have the duplicate values. This is one-to-many relationship. In this case, INNER JOIN will return 10

The following is the scenario of many-to-many relationships.

Many-to-many relationship

The last scenario to discuss is many-to-many relationship. In this case, there are two tables involved but there is no primary key and foreign key relationship involved in the column used for the join. It can be understood with following example.

Table 6.8 can be related to which has two columns - **ID** and It is to be noted that ID is not a primary key and has duplicate values:

[illegible]

Table 6.8: Table-A for many-to-many relationship

[Table 6.9](#) can be related to which has two columns - **ID** and
It is to be noted that ID is not a foreign key and has
duplicate values:

values:
values:

values:
values:
values:
values:
values:
values:
values:
values:
values:

Table 6.9: Table-B for many-to-many relationship

If you'll observe then *Table-A* has 10 rows of 5 unique
Whereas, *Table-B* has 10 rows of 5 unique You can see
duplicate values in both tables. This is many-to-many

relationship. In this case, INNER JOIN will return 20 You would be wondering *how?* *The following explanation should help!*

*(Number of rows in Table-A for each ID) * (number of values in Table-B for each ID)*

The calculation will go like as shown in [table](#) There are five distinct values in **ID** column of *Table-A* each having two records, whereas *Table-B* has two values for each ID:

ID: ID: ID: ID: ID: ID: ID: ID: ID: ID:
ID:
ID:

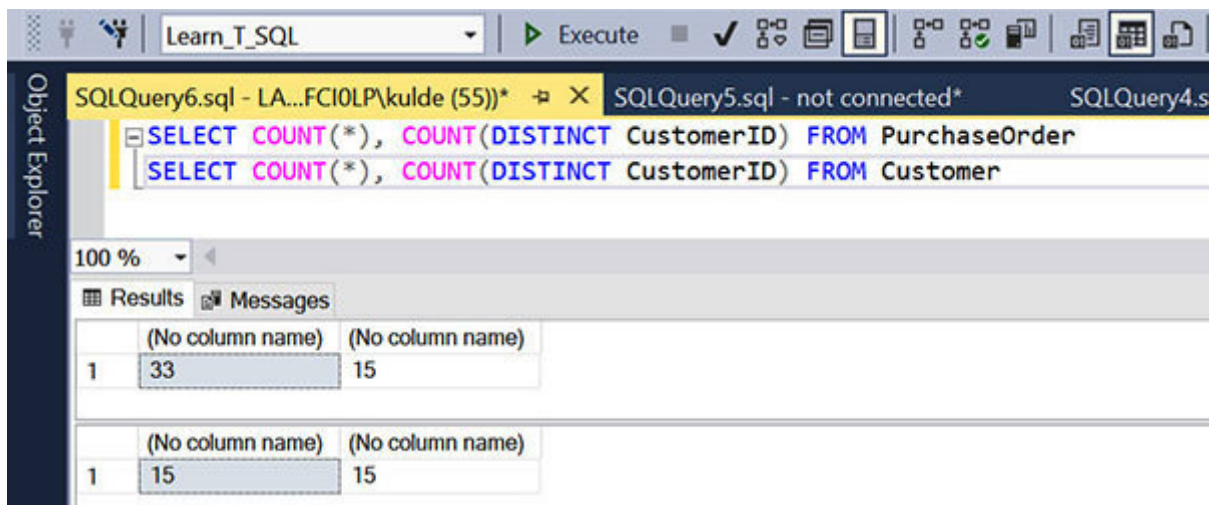
ID:
ID:
ID:
ID:

Table 6.10: *Calculation of number of rows returned by Inner Join in case of many-to-many relationship*

We'll understand it in a better way with the help of the tables we already created in **Learn_T_SQL** database.

The example is as follows:

Here is the example showing the use of INNER JOIN in T-SQL query. We'll use the **PurchaseOrder** and **Customer** tables in our example. Before we see the example, let's see the number of records in these tables. The following screenshot shows the simple **SELECT** query with the resultant output of the query:



The screenshot displays the SQL Server Enterprise Manager interface. The top pane shows two SQL queries. The first query, in 'SQLQuery6.sql', counts the total rows and distinct CustomerIDs in the 'PurchaseOrder' table. The second query, in 'SQLQuery5.sql', counts the total rows and distinct CustomerIDs in the 'Customer' table. The bottom pane shows the results of these queries in a table format.

	(No column name)	(No column name)
1	33	15

	(No column name)	(No column name)
1	15	15

Figure 6.7: Count of total rows and unique CustomerIDs in PurchaseOrder and Customer tables

The following screenshot shows the data in the **Customer** table:

SQLQuery2.sql - LA...FCI0LP\kulde (52))* SQLQuery1.sql - LA...FCI0LP\kulde (53))*

SELECT * FROM Customer

100 %

Results Messages

	CustomerID	CustomerName	CustomerAddress	CustomerMobile
1	1	Customer 1	Customer 1 Address	1111111111
2	2	Customer 2	Customer 2 Address	2222222222
3	3	Customer 3	Customer 3 Address	3333333333
4	4	Customer 4	Customer 4 Address	4444444444
5	5	Customer 5	Customer 5 Address	5555555555
6	6	Customer 6	Customer 6 Address	6666666666
7	7	Customer 7	Customer 7 Address	7777777777
8	8	Customer 8	Customer 8 Address	8888888888
9	9	Customer 9	Customer 9 Address	9999999999
10	10	Customer 10	Customer 9 Address	1010101010
11	11	Customer 11	Customer 11 Address	1111111111
12	12	Customer 12	Customer 12 Address	1212121212
13	13	Customer 13	Customer 13 Address	1313131313
14	14	Customer 14	Customer 14 Address	1414141414
15	15	Customer 15	Customer 15 Address	1515151515

Figure 6.8: Records in the Customer table

The following screenshot shows the data in **PurchaseOrder** table by running the simple **SELECT * FROM PurchaseOrder** query:

	OrderID	TransactionDate	CustomerID	ProductID	Quantity	Rate	Amount
1	1	2019-01-01	1	1	1	150000.00	150000.00
2	2	2019-05-15	1	2	1	250000.00	250000.00
3	3	2019-08-20	1	3	2	150000.00	300000.00
4	4	2019-10-20	1	4	1	350000.00	350000.00
5	5	2019-12-20	1	5	3	100000.00	300000.00
6	6	2019-01-01	2	1	1	150000.00	150000.00
7	7	2019-08-15	2	2	1	250000.00	250000.00
8	8	2019-12-20	3	3	2	150000.00	300000.00
9	9	2020-02-20	3	4	1	350000.00	350000.00
10	10	2020-03-20	3	5	3	100000.00	300000.00
11	11	2019-01-20	4	5	3	100000.00	300000.00
12	12	2019-05-01	5	1	1	150000.00	150000.00
13	13	2019-10-15	5	2	1	250000.00	250000.00
14	14	2019-04-20	6	3	2	150000.00	300000.00
15	15	2020-01-20	7	4	1	350000.00	350000.00
16	16	2020-02-20	7	5	3	100000.00	300000.00
17	17	2020-03-20	8	1	3	100000.00	300000.00
18	18	2019-01-20	8	3	3	100000.00	300000.00
19	19	2019-05-01	8	5	1	150000.00	150000.00
20	20	2019-10-15	9	2	1	250000.00	250000.00
21	21	2019-04-20	10	2	2	150000.00	300000.00
22	22	2020-01-20	11	4	1	350000.00	350000.00
23	23	2020-02-20	12	5	3	100000.00	300000.00
24	24	2019-01-01	13	1	1	150000.00	150000.00
25	25	2019-05-15	13	2	1	250000.00	250000.00
26	26	2019-08-20	13	3	2	150000.00	300000.00
27	27	2019-10-20	14	4	1	350000.00	350000.00
28	28	2019-12-20	14	5	3	100000.00	300000.00
29	29	2019-01-10	15	1	1	160000.00	160000.00
30	30	2019-05-25	15	2	1	200000.00	200000.00
31	31	2019-08-22	15	3	2	150000.00	300000.00
32	32	2020-01-13	15	4	1	350000.00	350000.00
33	33	2020-03-25	15	5	3	100000.00	300000.00

Figure 6.9: Records in the PurchaseOrder table

If we review the data as shown in the preceding screenshot, then you could see that **PurchaseOrder** table has total of 33 rows of 15 unique. Similarly, the **Customer** table has 15 rows of 15 unique. **CustomerID** is the common linkage between both tables. **CustomerID** column of **Customer** table has primary key constraint defined on it, whereas **CustomerID** column of **PurchaseOrder** table has foreign key constraint defined.

If we've to join **PurchaseOrder** table with **Customer** table on **CustomerID** column then the query will look like as shown here:

SELECT

,

FROM PurchaseOrder P

INNER JOIN Customer C

ON =

The output of previously mentioned query will look like as shown in the following screenshot:

Results Messages											
	CustomerID	CustomerName	CustomerAddress	CustomerMobile	OrderID	TransactionDate	CustomerID	ProductID	Quantity	Rate	Amount
1	1	Customer 1	Customer 1 Address	1111111111	1	2019-01-01	1	1	1	150000.00	150000.00
2	1	Customer 1	Customer 1 Address	1111111111	2	2019-05-15	1	2	1	250000.00	250000.00
3	1	Customer 1	Customer 1 Address	1111111111	3	2019-08-20	1	3	2	150000.00	300000.00
4	1	Customer 1	Customer 1 Address	1111111111	4	2019-10-20	1	4	1	350000.00	350000.00
5	1	Customer 1	Customer 1 Address	1111111111	5	2019-12-20	1	5	3	100000.00	300000.00
6	2	Customer 2	Customer 2 Address	2222222222	6	2019-01-01	2	1	1	150000.00	150000.00
7	2	Customer 2	Customer 2 Address	2222222222	7	2019-08-15	2	2	1	250000.00	250000.00
8	3	Customer 3	Customer 3 Address	3333333333	8	2019-12-20	3	3	2	150000.00	300000.00
9	3	Customer 3	Customer 3 Address	3333333333	9	2020-02-20	3	4	1	350000.00	350000.00
10	3	Customer 3	Customer 3 Address	3333333333	10	2020-03-20	3	5	3	100000.00	300000.00
11	4	Customer 4	Customer 4 Address	4444444444	11	2019-01-20	4	5	3	100000.00	300000.00
12	5	Customer 5	Customer 5 Address	5555555555	12	2019-05-01	5	1	1	150000.00	150000.00
13	5	Customer 5	Customer 5 Address	5555555555	13	2019-10-15	5	2	1	250000.00	250000.00
14	6	Customer 6	Customer 6 Address	6666666666	14	2019-04-20	6	3	2	150000.00	300000.00
15	7	Customer 7	Customer 7 Address	7777777777	15	2020-01-20	7	4	1	350000.00	350000.00
16	7	Customer 7	Customer 7 Address	7777777777	16	2020-02-20	7	5	3	100000.00	300000.00
17	8	Customer 8	Customer 8 Address	8888888888	17	2020-03-20	8	1	3	100000.00	300000.00
18	8	Customer 8	Customer 8 Address	8888888888	18	2019-01-20	8	3	3	100000.00	300000.00
19	8	Customer 8	Customer 8 Address	8888888888	19	2019-05-01	8	5	1	150000.00	150000.00
20	9	Customer 9	Customer 9 Address	9999999999	20	2019-10-15	9	2	1	250000.00	250000.00
21	10	Customer 10	Customer 9 Address	1010101010	21	2019-04-20	10	2	2	150000.00	300000.00
22	11	Customer 11	Customer 11 Addr...	1111111111	22	2020-01-20	11	4	1	350000.00	350000.00
23	12	Customer 12	Customer 12 Addr...	1212121212	23	2020-02-20	12	5	3	100000.00	300000.00
24	13	Customer 13	Customer 13 Addr...	1313131313	24	2019-01-01	13	1	1	150000.00	150000.00
25	13	Customer 13	Customer 13 Addr...	1313131313	25	2019-05-15	13	2	1	250000.00	250000.00
26	13	Customer 13	Customer 13 Addr...	1313131313	26	2019-08-20	13	3	2	150000.00	300000.00
27	14	Customer 14	Customer 14 Addr...	1414141414	27	2019-10-20	14	4	1	350000.00	350000.00
28	14	Customer 14	Customer 14 Addr...	1414141414	28	2019-12-20	14	5	3	100000.00	300000.00
29	15	Customer 15	Customer 15 Addr...	1515151515	29	2019-01-10	15	1	1	160000.00	160000.00
30	15	Customer 15	Customer 15 Addr...	1515151515	30	2019-05-25	15	2	1	200000.00	200000.00
31	15	Customer 15	Customer 15 Addr...	1515151515	31	2019-08-22	15	3	2	150000.00	300000.00
32	15	Customer 15	Customer 15 Addr...	1515151515	32	2020-01-13	15	4	1	350000.00	350000.00
33	15	Customer 15	Customer 15 Addr...	1515151515	33	2020-03-25	15	5	3	100000.00	300000.00

Figure 6.10: *Output of Inner Join between PurchaseOrder and Customer tables*

Never use * unless you need all the columns of a table.
Instead, specify the individual columns that you need.

OUTER JOIN

OUTER JOIN returns all the rows from the participating tables satisfying the join condition. It also returns the rows which do not satisfy the join conditions. There are three different types of OUTER JOIN. We'll understand each of them in the succeeding topics as follows:

LEFT OUTER JOIN

LEFT OUTER JOIN is also called as **LEFT**. These two terminologies are used interchangeably. It can be understood with the help of the following diagram. You could see two circles A and B intersecting each other. Each circle represents a table. Circle-A represents *Table-A* and circle-B represents as we already understood while learning INNER JOIN.

There are two tables involved in any join. One before the **JOIN** keyword and another after. The table before **JOIN** keyword is called *left table* and the table after **JOIN** keyword is called *right*

LEFT JOIN returns all the rows from the left table and matching rows (based on the join condition) from the right table. The following diagram also denotes the same thing:

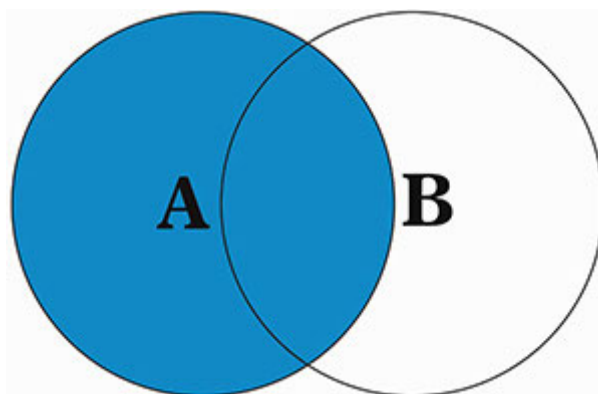


Figure 6.11: *Left Outer Join – All the rows from left table and only matching rows from the right table*

LEFT JOIN can be also used to get the rows from left table, if the join condition does not have any matching row in the right table. It can be understood from the following diagram:

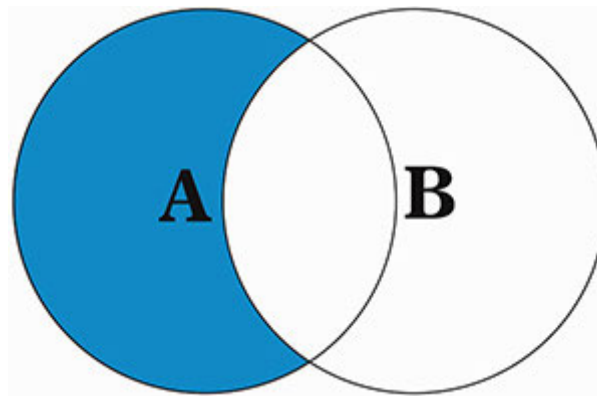


Figure 6.12: *Left Outer Join – All the rows from left table, only if no matching rows in the right table*

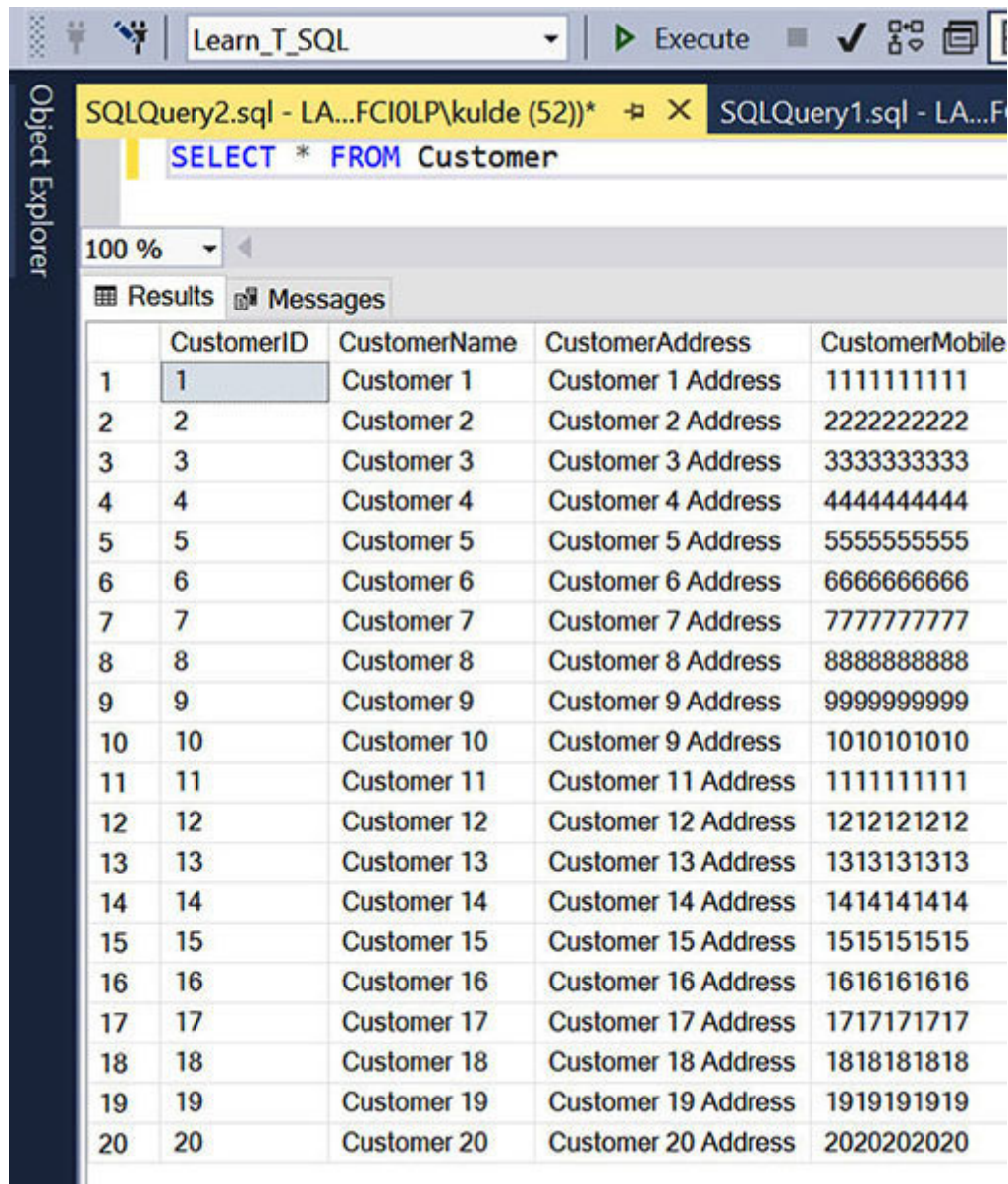
We'll now see the working of LEFT JOIN with **Customer** and **PurchaseOrder** tables. But if you remember the **Customer** table has 15 rows of 15 unique customers and **PurchaseOrder** table has 33 rows of 15 unique Both tables have 15 So, we would not be able to observe the behavior of LEFT JOIN with the current data set. Either we'll have to add new records **Customer** table or delete some from **PurchaseOrder** table. It'll be better if we add new customers that we can use later.

We'll use the following script to add five more customers:

```
INSERT INTO Customer
VALUES 'Customer 'Customer 16
, 'Customer 'Customer 17
, 'Customer 'Customer 18

, 'Customer 'Customer 19
, 'Customer 'Customer 20
```

Running the preceding script will add five more records in the **Customer** table. Now the table contains 20 *records* as can be seen in the following screenshot:



SQLQuery2.sql - LA...FCI0LP\kulde (52))* X SQLQuery1.sql - LA...F

SELECT * FROM Customer

100 %

Results Messages

	CustomerID	CustomerName	CustomerAddress	CustomerMobile
1	1	Customer 1	Customer 1 Address	1111111111
2	2	Customer 2	Customer 2 Address	2222222222
3	3	Customer 3	Customer 3 Address	3333333333
4	4	Customer 4	Customer 4 Address	4444444444
5	5	Customer 5	Customer 5 Address	5555555555
6	6	Customer 6	Customer 6 Address	6666666666
7	7	Customer 7	Customer 7 Address	7777777777
8	8	Customer 8	Customer 8 Address	8888888888
9	9	Customer 9	Customer 9 Address	9999999999
10	10	Customer 10	Customer 9 Address	1010101010
11	11	Customer 11	Customer 11 Address	1111111111
12	12	Customer 12	Customer 12 Address	1212121212
13	13	Customer 13	Customer 13 Address	1313131313
14	14	Customer 14	Customer 14 Address	1414141414
15	15	Customer 15	Customer 15 Address	1515151515
16	16	Customer 16	Customer 16 Address	1616161616
17	17	Customer 17	Customer 17 Address	1717171717
18	18	Customer 18	Customer 18 Address	1818181818
19	19	Customer 19	Customer 19 Address	1919191919
20	20	Customer 20	Customer 20 Address	2020202020

Figure 6.13: Records in the Customer table

We can now try our query with **LEFT** Here is the query with **LEFT JOIN** between **Customer** and **PurchaseOrder** tables. The customer table has 20 Whereas the **PurchaseOrder** table has data for only 15 Our join will return all the 20 customers in the form of 38 33 rows are part of **PurchaseOrder** table, belonging to 15 whereas five new customers are added to the **Customer** table. If you recollect the explanation of one-to-one,

one-to-many, and many-to-many relationships we already talked about, then you would be able to relate *why 38 rows are* This is the simple case of one-to-one relationship.

It is to be noted that there would be 38 rows belonging to all the 20 customers that would be returned as part of the output of the following query. However, not all columns will have the data. You would see columns of the **PurchaseOrder** table showing **NULL** values for the newly created five customers. The reason is simple because the theory says, **LEFT JOIN** will return all the rows from the left table and only matching rows from the right table. Since it has to return all the rows from the left table and only matching rows from the right table. That is why you would see the columns of left table showing the data it has, whereas right table showing the data as

```
SELECT
```

```
,
```

```
FROM Customer C
```

```
LEFT JOIN PurchaseOrder P
```

```
ON =
```

Running the preceding query will return the output as shown in the following screenshot. All the 38 rows were not coming in the single image. We divided them into two screenshots:

	CustomerID	CustomerName	CustomerAddress	CustomerMobile	OrderID	TransactionDate	CustomerID	ProductID	Quantity	Rate	Amount
1	1	Customer 1	Customer 1 Address	1111111111	1	2019-01-01	1	1	1	150000.00	150000.00
2	1	Customer 1	Customer 1 Address	1111111111	2	2019-05-15	1	2	1	250000.00	250000.00
3	1	Customer 1	Customer 1 Address	1111111111	3	2019-08-20	1	3	2	150000.00	300000.00
4	1	Customer 1	Customer 1 Address	1111111111	4	2019-10-20	1	4	1	350000.00	350000.00
5	1	Customer 1	Customer 1 Address	1111111111	5	2019-12-20	1	5	3	100000.00	300000.00
6	2	Customer 2	Customer 2 Address	2222222222	6	2019-01-01	2	1	1	150000.00	150000.00
7	2	Customer 2	Customer 2 Address	2222222222	7	2019-08-15	2	2	1	250000.00	250000.00
8	3	Customer 3	Customer 3 Address	3333333333	8	2019-12-20	3	3	2	150000.00	300000.00
9	3	Customer 3	Customer 3 Address	3333333333	9	2020-02-20	3	4	1	350000.00	350000.00
10	3	Customer 3	Customer 3 Address	3333333333	10	2020-03-20	3	5	3	100000.00	300000.00
11	4	Customer 4	Customer 4 Address	4444444444	11	2019-01-20	4	5	3	100000.00	300000.00
12	5	Customer 5	Customer 5 Address	5555555555	12	2019-05-01	5	1	1	150000.00	150000.00
13	5	Customer 5	Customer 5 Address	5555555555	13	2019-10-15	5	2	1	250000.00	250000.00
14	6	Customer 6	Customer 6 Address	6666666666	14	2019-04-20	6	3	2	150000.00	300000.00
15	7	Customer 7	Customer 7 Address	7777777777	15	2020-01-20	7	4	1	350000.00	350000.00
16	7	Customer 7	Customer 7 Address	7777777777	16	2020-02-20	7	5	3	100000.00	300000.00
17	8	Customer 8	Customer 8 Address	8888888888	17	2020-03-20	8	1	3	100000.00	300000.00
18	8	Customer 8	Customer 8 Address	8888888888	18	2019-01-20	8	3	3	100000.00	300000.00
19	8	Customer 8	Customer 8 Address	8888888888	19	2019-05-01	8	5	1	150000.00	150000.00
20	9	Customer 9	Customer 9 Address	9999999999	20	2019-10-15	9	2	1	250000.00	250000.00
21	10	Customer 10	Customer 9 Address	1010101010	21	2019-04-20	10	2	2	150000.00	300000.00
22	11	Customer 11	Customer 11 Address	1111111111	22	2020-01-20	11	4	1	350000.00	350000.00
23	12	Customer 12	Customer 12 Address	1212121212	23	2020-02-20	12	5	3	100000.00	300000.00
24	13	Customer 13	Customer 13 Address	1313131313	24	2019-01-01	13	1	1	150000.00	150000.00
25	13	Customer 13	Customer 13 Address	1313131313	25	2019-05-15	13	2	1	250000.00	250000.00
26	13	Customer 13	Customer 13 Address	1313131313	26	2019-08-20	13	3	2	150000.00	300000.00
27	14	Customer 14	Customer 14 Address	1414141414	27	2019-10-20	14	4	1	350000.00	350000.00
28	14	Customer 14	Customer 14 Address	1414141414	28	2019-12-20	14	5	3	100000.00	300000.00
29	15	Customer 15	Customer 15 Address	1515151515	29	2019-01-10	15	1	1	160000.00	160000.00
30	15	Customer 15	Customer 15 Address	1515151515	30	2019-05-25	15	2	1	200000.00	200000.00
31	15	Customer 15	Customer 15 Address	1515151515	31	2019-08-22	15	3	2	150000.00	300000.00
32	15	Customer 15	Customer 15 Address	1515151515	32	2020-01-13	15	4	1	350000.00	350000.00
33	15	Customer 15	Customer 15 Address	1515151515	33	2020-03-25	15	5	3	100000.00	300000.00

Figure 6.14: Output of Left Join between Customer and PurchaseOrder tables, all the rows from Customer table and only matching rows from the PurchaseOrder table

The following screenshot shows the next five rows:

	CustomerID	CustomerName	CustomerAddress	CustomerMobile	OrderID	TransactionDate	CustomerID	ProductID	Quantity	Rate	Amount
34	16	Customer 16	Customer 16 Address	1616161616	NULL	NULL	NULL	NULL	NULL	NULL	NULL
35	17	Customer 17	Customer 17 Address	1717171717	NULL	NULL	NULL	NULL	NULL	NULL	NULL
36	18	Customer 18	Customer 18 Address	1818181818	NULL	NULL	NULL	NULL	NULL	NULL	NULL
37	19	Customer 19	Customer 19 Address	1919191919	NULL	NULL	NULL	NULL	NULL	NULL	NULL
38	20	Customer 20	Customer 20 Address	2020202020	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Figure 6.15: Output of Left Join between Customer and PurchaseOrder tables, all the rows from Customer table and only matching rows from the PurchaseOrder table continued

If we'll simply put a filter clause, which is a **WHERE** condition, in the query to look for only non-null values in the

column of the right table, which will generally not be null, then we'll get the data from the left table, which do not have any rows in the right table, based on the **JOIN** condition. The following is the revised query after adding a filter clause.

PurchaseOrder is the right table in our query. **OrderID** column is the primary key of the table which will never be null. That's why we choose it for the filter clause:

```
SELECT  
,  
FROM Customer C  
LEFT JOIN PurchaseOrder P  
ON =  
WHERE IS NULL
```

Running the preceding query will return the output of five rows as shown in the following screenshot. The returned by the query only exists in the left table You won't find any matching rows against these in the right table

	CustomerID	CustomerName	CustomerAddress	CustomerMobile	OrderID	TransactionDate	CustomerID	ProductID	Quantity	Rate	Amount
1	16	Customer 16	Customer 16 Address	1616161616	NULL	NULL	NULL	NULL	NULL	NULL	NULL
2	17	Customer 17	Customer 17 Address	1717171717	NULL	NULL	NULL	NULL	NULL	NULL	NULL
3	18	Customer 18	Customer 18 Address	1818181818	NULL	NULL	NULL	NULL	NULL	NULL	NULL
4	19	Customer 19	Customer 19 Address	1919191919	NULL	NULL	NULL	NULL	NULL	NULL	NULL
5	20	Customer 20	Customer 20 Address	2020202020	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Figure 6.16: Output of Left Join between Customer and PurchaseOrder tables, all the rows from Customer table, only if no matching rows in the PurchaseOrder table

RIGHT OUTER JOIN

RIGHT OUTER JOIN is also called as **RIGHT**. These two terminologies are used interchangeably. It can be understood with the help of the following diagram. RIGHT JOIN is the exactly opposite of LEFT JOIN. RIGHT JOIN is rarely used because the desired results can be achieved even with LEFT JOIN that is widely used.

RIGHT JOIN returns all the rows from the right table and matching rows (based on the join condition) from the left table. The following diagram also denotes the same thing:

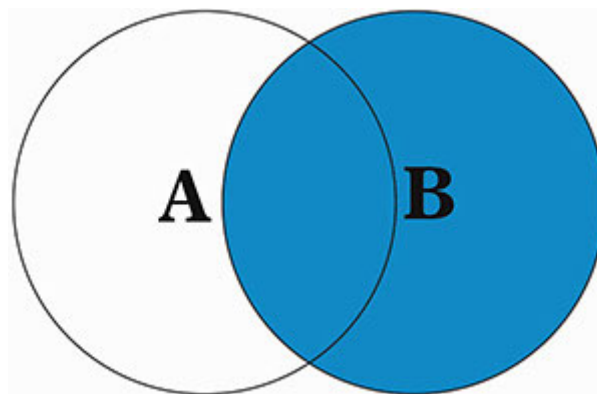


Figure 6.17: Right Outer Join – All the rows from right table and only matching rows from the left table

RIGHT JOIN can be also used to get the rows from right table, if the join condition does not have any matching row

in the left table. It can be understood from the following diagram:

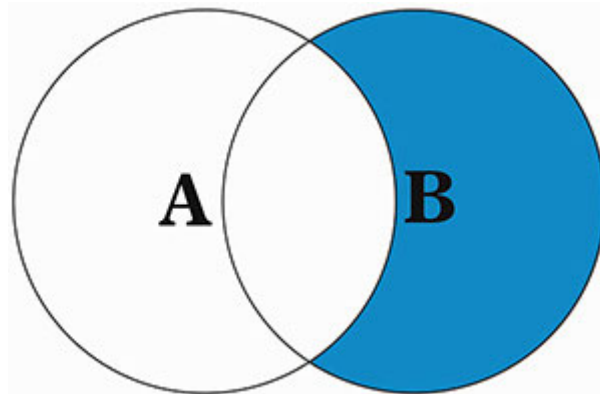


Figure 6.18: *Right Outer Join – All the rows from right table, only if no matching rows in the left table*

Let's now try query with RIGHT JOIN between **Customer** and **PurchaseOrder** tables. Here is the query:

```
SELECT  
,  
FROM Customer C  
RIGHT JOIN PurchaseOrder P  
ON =
```

Running the preceding query will return the output as shown in the following screenshot. The query has returned 33 rows and the output is similar to what has come with INNER JOIN. You would be wondering *why?* Because theory says, *RIGHT JOIN returns all the rows from the right table and only matching rows from the left table.*

The right table has 33 rows of 15 unique whereas the left table has 20 rows of 20 unique Customer tables have five customers that have no records in the **PurchaseOrder** table:

	CustomerID	CustomerName	CustomerAddress	CustomerMobile	OrderID	TransactionDate	CustomerID	ProductID	Quantity	Rate	Amount
1	1	Customer 1	Customer 1 Address	1111111111	1	2019-01-01	1	1	1	150000.00	150000.00
2	1	Customer 1	Customer 1 Address	1111111111	2	2019-05-15	1	2	1	250000.00	250000.00
3	1	Customer 1	Customer 1 Address	1111111111	3	2019-08-20	1	3	2	150000.00	300000.00
4	1	Customer 1	Customer 1 Address	1111111111	4	2019-10-20	1	4	1	350000.00	350000.00
5	1	Customer 1	Customer 1 Address	1111111111	5	2019-12-20	1	5	3	100000.00	300000.00
6	2	Customer 2	Customer 2 Address	2222222222	6	2019-01-01	2	1	1	150000.00	150000.00
7	2	Customer 2	Customer 2 Address	2222222222	7	2019-08-15	2	2	1	250000.00	250000.00
8	3	Customer 3	Customer 3 Address	3333333333	8	2019-12-20	3	3	2	150000.00	300000.00
9	3	Customer 3	Customer 3 Address	3333333333	9	2020-02-20	3	4	1	350000.00	350000.00
10	3	Customer 3	Customer 3 Address	3333333333	10	2020-03-20	3	5	3	100000.00	300000.00
11	4	Customer 4	Customer 4 Address	4444444444	11	2019-01-20	4	5	3	100000.00	300000.00
12	5	Customer 5	Customer 5 Address	5555555555	12	2019-05-01	5	1	1	150000.00	150000.00
13	5	Customer 5	Customer 5 Address	5555555555	13	2019-10-15	5	2	1	250000.00	250000.00
14	6	Customer 6	Customer 6 Address	6666666666	14	2019-04-20	6	3	2	150000.00	300000.00
15	7	Customer 7	Customer 7 Address	7777777777	15	2020-01-20	7	4	1	350000.00	350000.00
16	7	Customer 7	Customer 7 Address	7777777777	16	2020-02-20	7	5	3	100000.00	300000.00
17	8	Customer 8	Customer 8 Address	8888888888	17	2020-03-20	8	1	3	100000.00	300000.00
18	8	Customer 8	Customer 8 Address	8888888888	18	2019-01-20	8	3	3	100000.00	300000.00
19	8	Customer 8	Customer 8 Address	8888888888	19	2019-05-01	8	5	1	150000.00	150000.00
20	9	Customer 9	Customer 9 Address	9999999999	20	2019-10-15	9	2	1	250000.00	250000.00
21	10	Customer 10	Customer 9 Address	1010101010	21	2019-04-20	10	2	2	150000.00	300000.00
22	11	Customer 11	Customer 11 Address	1111111111	22	2020-01-20	11	4	1	350000.00	350000.00
23	12	Customer 12	Customer 12 Address	1212121212	23	2020-02-20	12	5	3	100000.00	300000.00
24	13	Customer 13	Customer 13 Address	1313131313	24	2019-01-01	13	1	1	150000.00	150000.00
25	13	Customer 13	Customer 13 Address	1313131313	25	2019-05-15	13	2	1	250000.00	250000.00
26	13	Customer 13	Customer 13 Address	1313131313	26	2019-08-20	13	3	2	150000.00	300000.00
27	14	Customer 14	Customer 14 Address	1414141414	27	2019-10-20	14	4	1	350000.00	350000.00
28	14	Customer 14	Customer 14 Address	1414141414	28	2019-12-20	14	5	3	100000.00	300000.00
29	15	Customer 15	Customer 15 Address	1515151515	29	2019-01-10	15	1	1	160000.00	160000.00
30	15	Customer 15	Customer 15 Address	1515151515	30	2019-05-25	15	2	1	200000.00	200000.00
31	15	Customer 15	Customer 15 Address	1515151515	31	2019-08-22	15	3	2	150000.00	300000.00
32	15	Customer 15	Customer 15 Address	1515151515	32	2020-01-13	15	4	1	350000.00	350000.00
33	15	Customer 15	Customer 15 Address	1515151515	33	2020-03-25	15	5	3	100000.00	300000.00

Figure 6.19: Output of Right Join between Customer and PurchaseOrder tables, all the rows from PurchaseOrder table and only matching rows from the Customer table

Remember, it was said that RIGHT JOIN is exactly the opposite of LEFT JOIN? We'll prove it now. Let's simply change the position of the tables in the query. We'll make the left table right and vice versa. Here we go!

SELECT

,

```
FROM PurchaseOrder P  
RIGHT JOIN Customer C  
ON =
```

Running the preceding query will return the output as shown in the following screenshot. It returned 38 rows and if you would compare the output with what has come in LEFT JOIN then you would not see any difference.

All the 38 rows were not coming in the single image, hence, we'd to cut it into two screenshots. The following screenshot shows the first 33

	CustomerID	CustomerName	CustomerAddress	CustomerMobile	OrderID	TransactionDate	CustomerID	ProductID	Quantity	Rate	Amount
1	1	Customer 1	Customer 1 Address	1111111111	1	2019-01-01	1	1	1	150000.00	150000.00
2	1	Customer 1	Customer 1 Address	1111111111	2	2019-05-15	1	2	1	250000.00	250000.00
3	1	Customer 1	Customer 1 Address	1111111111	3	2019-08-20	1	3	2	150000.00	300000.00
4	1	Customer 1	Customer 1 Address	1111111111	4	2019-10-20	1	4	1	350000.00	350000.00
5	1	Customer 1	Customer 1 Address	1111111111	5	2019-12-20	1	5	3	100000.00	300000.00
6	2	Customer 2	Customer 2 Address	2222222222	6	2019-01-01	2	1	1	150000.00	150000.00
7	2	Customer 2	Customer 2 Address	2222222222	7	2019-08-15	2	2	1	250000.00	250000.00
8	3	Customer 3	Customer 3 Address	3333333333	8	2019-12-20	3	3	2	150000.00	300000.00
9	3	Customer 3	Customer 3 Address	3333333333	9	2020-02-20	3	4	1	350000.00	350000.00
10	3	Customer 3	Customer 3 Address	3333333333	10	2020-03-20	3	5	3	100000.00	300000.00
11	4	Customer 4	Customer 4 Address	4444444444	11	2019-01-20	4	5	3	100000.00	300000.00
12	5	Customer 5	Customer 5 Address	5555555555	12	2019-05-01	5	1	1	150000.00	150000.00
13	5	Customer 5	Customer 5 Address	5555555555	13	2019-10-15	5	2	1	250000.00	250000.00
14	6	Customer 6	Customer 6 Address	6666666666	14	2019-04-20	6	3	2	150000.00	300000.00
15	7	Customer 7	Customer 7 Address	7777777777	15	2020-01-20	7	4	1	350000.00	350000.00
16	7	Customer 7	Customer 7 Address	7777777777	16	2020-02-20	7	5	3	100000.00	300000.00
17	8	Customer 8	Customer 8 Address	8888888888	17	2020-03-20	8	1	3	100000.00	300000.00
18	8	Customer 8	Customer 8 Address	8888888888	18	2019-01-20	8	3	3	100000.00	300000.00
19	8	Customer 8	Customer 8 Address	8888888888	19	2019-05-01	8	5	1	150000.00	150000.00
20	9	Customer 9	Customer 9 Address	9999999999	20	2019-10-15	9	2	1	250000.00	250000.00
21	10	Customer 10	Customer 9 Address	1010101010	21	2019-04-20	10	2	2	150000.00	300000.00
22	11	Customer 11	Customer 11 Address	1111111111	22	2020-01-20	11	4	1	350000.00	350000.00
23	12	Customer 12	Customer 12 Address	1212121212	23	2020-02-20	12	5	3	100000.00	300000.00
24	13	Customer 13	Customer 13 Address	1313131313	24	2019-01-01	13	1	1	150000.00	150000.00
25	13	Customer 13	Customer 13 Address	1313131313	25	2019-05-15	13	2	1	250000.00	250000.00
26	13	Customer 13	Customer 13 Address	1313131313	26	2019-08-20	13	3	2	150000.00	300000.00
27	14	Customer 14	Customer 14 Address	1414141414	27	2019-10-20	14	4	1	350000.00	350000.00
28	14	Customer 14	Customer 14 Address	1414141414	28	2019-12-20	14	5	3	100000.00	300000.00
29	15	Customer 15	Customer 15 Address	1515151515	29	2019-01-10	15	1	1	160000.00	160000.00
30	15	Customer 15	Customer 15 Address	1515151515	30	2019-05-25	15	2	1	200000.00	200000.00
31	15	Customer 15	Customer 15 Address	1515151515	31	2019-08-22	15	3	2	150000.00	300000.00
32	15	Customer 15	Customer 15 Address	1515151515	32	2020-01-13	15	4	1	350000.00	350000.00
33	15	Customer 15	Customer 15 Address	1515151515	33	2020-03-25	15	5	3	100000.00	300000.00

Figure 6.20: Output of Right Join between PurchaseOrder and Customer tables, all the rows from Customer table and only matching rows from the PurchaseOrder table

The following screenshot shows the next five rows:

	CustomerID	CustomerName	CustomerAddress	CustomerMobile	OrderID	TransactionDate	CustomerID	ProductID	Quantity	Rate	Amount
34	16	Customer 16	Customer 16 Address	1616161616	NULL	NULL	NULL	NULL	NULL	NULL	NULL
35	17	Customer 17	Customer 17 Address	1717171717	NULL	NULL	NULL	NULL	NULL	NULL	NULL
36	18	Customer 18	Customer 18 Address	1818181818	NULL	NULL	NULL	NULL	NULL	NULL	NULL
37	19	Customer 19	Customer 19 Address	1919191919	NULL	NULL	NULL	NULL	NULL	NULL	NULL
38	20	Customer 20	Customer 20 Address	2020202020	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Figure 6.21: Output of Right Join between PurchaseOrder and Customer tables, all the rows from Customer table and only matching rows from the PurchaseOrder table continued

If we'll simply put a filter clause, which is a **WHERE** condition, in the query to look for only non-null values in the column of the left table, which will generally not be NULL, then we'll get the data from the right table, which do not have any rows in the left table, based on the **JOIN** condition. The following is the revised query after adding a filter clause.

PurchaseOrder is the left table in our query. **OrderID** column is the primary key of the table which will never be null. That's why we choose it for the filter clause:

SELECT

,

FROM PurchaseOrder P

RIGHT JOIN Customer C

ON =
WHERE IS NULL

Running the preceding query will return the output of five rows as shown in the following screenshot. The returned by the query only exists in the right table You won't find any matching rows against these in the left table

	CustomerID	CustomerName	CustomerAddress	CustomerMobile	OrderID	TransactionDate	CustomerID	ProductID	Quantity	Rate	Amount
1	16	Customer 16	Customer 16 Address	1616161616	NULL	NULL	NULL	NULL	NULL	NULL	NULL
2	17	Customer 17	Customer 17 Address	1717171717	NULL	NULL	NULL	NULL	NULL	NULL	NULL
3	18	Customer 18	Customer 18 Address	1818181818	NULL	NULL	NULL	NULL	NULL	NULL	NULL
4	19	Customer 19	Customer 19 Address	1919191919	NULL	NULL	NULL	NULL	NULL	NULL	NULL
5	20	Customer 20	Customer 20 Address	2020202020	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Figure 6.22: Output of Right Join between PurchaseOrder and Customer tables, all the rows from Customer table, only if no matching rows in the PurchaseOrder table

FULL OUTER JOIN

FULL OUTER JOIN is also called as **FULL**. These two terminologies are used interchangeably. FULL JOIN can be considered the mixture of both LEFT and RIGHT JOIN's. It can be understood with the help of the following diagram.

FULL JOIN returns all the rows from both left and right tables. If the left table will not have the matching rows, then its columns will have NULL's. Similarly, if the right table will not have the matching rows, then its columns will have NULL's:

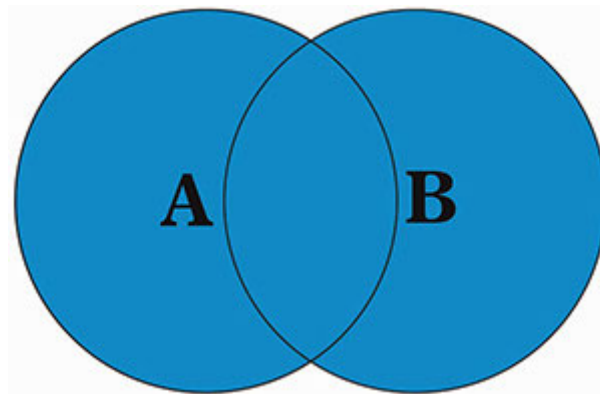


Figure 6.23: Full Outer Join – All the rows from both left and right tables

FULL JOIN can also be used to get the rows from both left and right tables that do not have any matching row in both tables based on the join condition. It means FULL JOIN can

be used to get all rows except the intersecting ones. It can be understood from the following diagram:

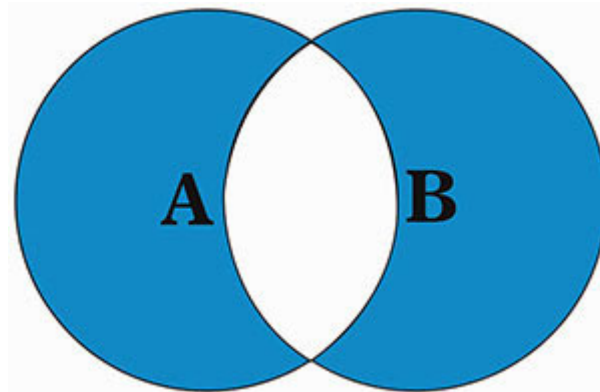


Figure 6.24: Full Outer Join – All the rows from both left and right tables, except the intersecting ones

Since we've already discussed LEFT JOIN and RIGHT JOIN in detail hence it doesn't make sense to repeat similar cases again. We'll only see various examples in the form of T-SQL queries, without talking much about the data and scenarios.

The following is one flavor of FULL JOIN. It will return all the rows from left and right tables, irrespective of whether is join condition is met or not:

```
SELECT
```

```
,
```

```
FROM Customer C
```

```
FULL JOIN PurchaseOrder P
```

```
ON =
```


Figure 6.26: *Output of Full Join between Customer and PurchaseOrder tables, all the rows from Customer and PurchaseOrder tables continued*

The query can be modified as we discussed in LEFT and RIGHT JOIN to have a filter clause to get the different results.

APPLY

APPLY clause is another feature that can be used as an alternative to join. The only difference is **APPLY** does not have any join condition. You need to specify all the conditions within the query itself.

JOIN can be done on table and on query. The query can be used in the join as the derived table and can be given an alias, whereas **APPLY** clause can't be used with the tables.

APPLY clause can be used with the table-valued functions. Whereas, **JOIN** cannot be directly used with the table-valued function. Table-valued function needs to be first converted into a **SELECT** query and then the query can be used in the join. We'll talk about this more in [Chapter 10, View and User Defined](#) For now, we'll only focus on the query and application of **APPLY** with query.

There are two variants of **APPLY** clause – **CROSS APPLY** and **OUTER CROSS APPLY** behaves like INNER JOIN, whereas **OUTER APPLY** behaves like LEFT JOIN. Let us see the practical implementation of each of these.

Although, **APPLY** clause is another way to perform the join. But it should be used wisely, and only if **JOIN** doesn't serve the purpose. **JOIN** should be given the preference over **APPLY**

clause. APPLY clause makes a query correlated. Correlated queries are slower.

CROSS APPLY

We've already discussed about **APPLY** and we also understood that **CROSS APPLY** can be related to INNER JOIN. Without talking much about the theory, let's learn to use it in T-SQL.

Suppose, we want to return the **CustomerName** and the number and values of the total order against each customer from the **PurchaseOrder** table. Here, we don't want the customers who do not have any order against it. The query can be written in various ways. However, we'll see three versions of the query – one using **JOIN** with table, second using **JOIN** with query, and third using **APPLY** clause.

The following is the query using **JOIN** with the table:

```
SELECT  
, AS NumberOfOrders  
, AS ValueOfOrders  
FROM Customer C  
INNER JOIN PurchaseOrder P  
ON =  
GROUP BY  
ORDER BY ASC
```

The following is the query using JOIN with the query:

```

SELECT
,
,
FROM Customer C
INNER JOIN
(
SELECT
, AS NumberOfOrders

, AS ValueOfOrders
FROM PurchaseOrder P
GROUP BY
) P
ON =
ORDER BY ASC

```

The following is the query using **APPLY** clause:

```

SELECT
,
,
FROM Customer C
CROSS APPLY
(
SELECT
, AS NumberOfOrders
, AS ValueOfOrders
FROM PurchaseOrder P
WHERE =
GROUP BY

```

) P

ORDER BY ASC

All the preceding three queries will return the same result as can be seen in the following screenshot:

	CustomerName	NumberOfOrders	ValueOfOrders
1	Customer 1	5	1350000.00
2	Customer 10	1	300000.00
3	Customer 11	1	350000.00
4	Customer 12	1	300000.00
5	Customer 13	3	700000.00
6	Customer 14	2	650000.00
7	Customer 15	5	1310000.00
8	Customer 2	2	400000.00
9	Customer 3	3	950000.00
10	Customer 4	1	300000.00
11	Customer 5	2	400000.00
12	Customer 6	1	300000.00
13	Customer 7	2	650000.00
14	Customer 8	3	750000.00
15	Customer 9	1	250000.00

Figure 6.27: Same output for the Inner Join and Cross Apply clause

OUTER APPLY

We've already understood that **OUTER APPLY** can be related to LEFT JOIN. Let us now understand the application of **OUTER**

Suppose, we want to return the number and values of the total order against each customer from the **PurchaseOrder** table, for all the customers in the **Customer** table. The scenario we discussed in **CROSS APPLY** is different from this scenario. In **CROSS** we were expecting the result-set of only those customers who have the order against it, but here we need all the customers from the **Customer** table and their respective order details. There is a possibility that few customers won't have any order, but still we need them in our result-set.

The query can be written in various ways. However, we'll see three versions of the query – one using **JOIN** with table, second using **JOIN** with query, and third using **APPLY** clause.

The following is the query using **JOIN** with table:

```
SELECT  
    , AS NumberOfOrders  
    , AS ValueOfOrders  
FROM Customer C
```

```
LEFT JOIN PurchaseOrder P
ON =
GROUP BY
ORDER BY ASC
```

The following is the query using **JOIN** with the query:

```
SELECT
,
,

FROM Customer C
LEFT JOIN
(
SELECT
, AS NumberOfOrders
, AS ValueOfOrders
FROM PurchaseOrder P
GROUP BY
) P
ON =
ORDER BY ASC
```

The following is the query using **APPLY** clause:

```
SELECT
,
,
FROM Customer C
OUTER APPLY
(
```



```

SELECT
, AS NumberOfOrders
, AS ValueOfOrders
FROM PurchaseOrder P
WHERE =
GROUP BY
) P
ORDER BY ASC

```

All the preceding three queries will return the same result as can be seen in the following screenshot:

	CustomerName	NumberOfOrders	ValueOfOrders
1	Customer 1	5	1350000.00
2	Customer 10	1	300000.00
3	Customer 11	1	350000.00
4	Customer 12	1	300000.00
5	Customer 13	3	700000.00
6	Customer 14	2	650000.00
7	Customer 15	5	1310000.00
8	Customer 16	NULL	NULL
9	Customer 17	NULL	NULL
10	Customer 18	NULL	NULL
11	Customer 19	NULL	NULL
12	Customer 2	2	400000.00
13	Customer 20	NULL	NULL
14	Customer 3	3	950000.00
15	Customer 4	1	300000.00
16	Customer 5	2	400000.00
17	Customer 6	1	300000.00
18	Customer 7	2	650000.00
19	Customer 8	3	750000.00
20	Customer 9	1	250000.00

***Figure 6.28:** Same output for the Left Join and Outer Apply clause*

If you would have noticed, there are five customers which have NULL's in **NumberOfOrders** and **ValueOfOrders** columns. This is because there is no record for these customers in the **PurchaseOrder** tables. These customers have come in the result-set as we've used **LEFT JOIN** and **OUTER**

Subquery.

You must be aware of sub-set, sub-section, sub-data, and so on. All these represent the child. For example, if there is a book having an index that has 15 Each chapter has multiple topics that can be also considered as sub-chapters. Other example if there are 100 rows in a table and you need only 10 rows out of 100 then those 10 rows are referred to as sub-set of

Similarly, subquery can be understood as a query within another query. There are various kinds of subqueries such as a **nested inline scalar simple** and **correlated** and so on.

Types of subqueries

The following query covers all kinds of subqueries as we discussed. The query will return the customer name, number of orders, value of orders, quantity of orders, of all the customers whose contact number is The query will also return the total quantity of all the orders across all the customers. The result-set will be sorted by customer name in the ascending order.

In the following example shown, we've used a simple query. However, it could be complex too, and can have joins, aggregated functions, group by, having clause, and so on:

SELECT

,

,

, FROM PurchaseOrder WHERE CustomerID = AS

QuantityOfOrders

, FROM AS TotalQuantityOfOrders

FROM Customer C

LEFT JOIN

(

SELECT

, AS NumberOfOrders

, AS ValueOfOrders

FROM PurchaseOrder P

GROUP BY

) P

ON =

WHERE IN CustomerID FROM Customer WHERE
CustomerMobile =
ORDER BY ASC

The following are the types of subqueries:

Nested subquery: The subquery appears in the **WHERE** clause of the SQL. As can be seen in the preceding screenshot, you can see a **WHERE** clause using **IN** operator. The query used with **IN** operator is called nested subquery.

Inline view: The subquery appears in the **FROM** clause of the SQL. The query used in the **LEFT JOIN** of the **FROM** clause as highlighted in the preceding screenshot is called inline view.

Scalar subquery: The subquery appears in the **SELECT** clause of the SQL. You will go through the preceding screenshot, then you would see two derived columns – **QuantityOfOrders** and You will also find these columns are derived from the queries. Such queries are called **scalar** Scalar means single and scalar subquery will always return a single value. There are two different varieties of scalar subqueries as follows:

Simple subquery: A simple subquery is evaluated once only for each table. The query for **TotalQuantityOfOrder** column

mentioned in the **SELECT** clause of the preceding screenshot is a simple subquery. This kind of scalar subquery will never refer to any column from the outer query.

Correlated subquery: This is a type of nested subquery that uses columns from the outer query in its **WHERE** clause. A correlated subquery is evaluated once for each row. The query for **QuantityOfOrders** column mentioned in the **SELECT** clause of the preceding screenshot is a correlated subquery. This kind of scalar subquery will always refer to columns from the outer query.

The following screenshot highlights the types of subqueries, inline in the query itself. The following screenshot contains the same query as mentioned previously. The screenshot also includes the output of the query:

The screenshot displays a SQL query in SQL Server Enterprise Manager. The query is as follows:

```
SELECT C.CustomerName
      , P.NumberOfOrders
      , P.ValueOfOrders
      , (SELECT COUNT(Quantity) FROM PurchaseOrder WHERE CustomerID = C.CustomerID) AS QuantityOfOrders
      , (SELECT COUNT(Quantity) FROM PurchaseOrder) AS TotalQuantityOfOrders
FROM Customer C
LEFT JOIN
(
    SELECT P.CustomerID
          , COUNT(P.OrderID) AS NumberOfOrders
          , SUM(P.Amount) AS ValueOfOrders
    FROM PurchaseOrder P
    GROUP BY P.CustomerID
) P
ON C.CustomerID = P.CustomerID
WHERE C.CustomerID IN (SELECT CustomerID FROM Customer WHERE CustomerMobile = '1111111111')
ORDER BY C.CustomerName ASC
```

The query is annotated with red boxes and callouts identifying different types of subqueries:

- Scalar subqueries:** Points to the subquery for `QuantityOfOrders`.
- Correlated subquery:** Points to the subquery for `QuantityOfOrders`.
- Simple subquery:** Points to the subquery for `TotalQuantityOfOrders`.
- Inline view:** Points to the subquery for `NumberOfOrders` and `ValueOfOrders`.
- Nested subquery:** Points to the subquery for `CustomerID` in the `WHERE` clause.

The results of the query are shown in the bottom pane:

	CustomerName	NumberOfOrders	ValueOfOrders	QuantityOfOrders	TotalQuantityOfOrders
1	Customer 1	5	1350000.00	5	33
2	Customer 11	1	350000.00	1	33

Figure 6.29: Types of subqueries

Conclusion

Well, we talked about quite insightful features of SQL Server and T-SQL that are – and subqueries. You cannot do advanced analytics without these features. You need them to combine multiple tables to get a single unified view. If you really want to master the T-SQL then you need to practice these features very hard to be on top of it.

In the next chapter, we'll explore more SQL Server offerings and learn various other advanced built-in functions.

Points to remember

LEFT JOIN and LEFT OUTER JOIN are the same and can be used interchangeably.

RIGHT JOIN and RIGHT OUTER JOIN are the same and can be used interchangeably.

FULL JOIN and FULL OUTER JOIN are the same and can be used interchangeably.

FULL JOIN is the combination of LEFT JOIN and RIGHT JOIN.

CROSS APPLY behaves like INNER JOIN.

OUTER APPLY behaves like LEFT JOIN.

There is no **ON** clause in In order to specify the conditions, you need to specify the **WHERE** clause in the query used in **APPLY** clause. However, you need to specify the join conditions in the **ON** clause in the case of

You cannot use the columns from the outer query in
However, it can be used in the

Correlated subqueries are executed once for each row, whereas other kinds of subqueries are executed once for the entire query.

There are two tables involved in any join. One before the **JOIN** keyword and another after. The table before **JOIN** keyword is called *left table* and table after **JOIN** keyword is called *right*

Never use * unless you need all the columns of a table. Instead, specify the individual columns that you need.

Although, **APPLY** clause is another way to perform the join. But it should be used wisely, and only if **JOIN** doesn't serve the purpose. **JOIN** should be given the preference over **APPLY** clause. **APPLY** clause makes a query correlated. Correlated queries are slower.

Multiple choice questions

When to use the INNER JOIN?

When data from multiple tables are required.

When matching data from two tables is required.

When one table does not have any data.

When data only from the right table are required.

Can you use subquery instead of APPLY?

Yes

No

Answers

B

A

Questions

What is the difference between INNER JOIN and LEFT JOIN?

What is the difference between RIGHT JOIN and FULL JOIN?

When to use subquery?

When to use

How to achieve the same purpose by using the LEFT JOIN instead of RIGHT JOIN?

What are the different joins in OUTER JOIN?

What is the difference between subquery and

Key terms

INNER JOIN returns the matching records from both tables, participated in the JOIN.

LEFT JOIN returns all the rows from the left table and matching rows (based on the join condition) from the right table.

RIGHT JOIN returns all the rows from right table and matching rows (based on the join condition) from the left table.

FULL JOIN can be considered the mixture of both LEFT and RIGHT JOIN's. It returns all the rows from both left and right tables. If the left table will not have the matching rows, then its columns will have NULL's. Similarly, if the right table will not have the matching rows, then its columns will have NULL's.

CROSS APPLY is used to join with queries and it behaves like INNER JOIN.

OUTER APPLY is used to join with queries and it behaves like LEFT JOIN.

Correlated subquery is executed once for each row. This kind of query refers the columns from the outer query.

CHAPTER 7

Built-In Functions – Part 2

SQL Server offers various advanced built-in functions that can provide additional helping hands for advanced analytics. We already learnt about various basic and regularly used built-in functions in [Chapter 5, Built-In Functions - Part 1](#). Those functions operate on the entire table. However, the functions we'll talk about in the chapter operate on a set of rows.

This chapter will take you deeper into the advanced analytical functions of SQL Server. With the help of the functions, you'll learn in this chapter, you would be able to partition the data and get various kinds of data very easily and quickly.

Structure

In this chapter, we will cover the following topics:

Window functions

Aggregate functions

Ranking functions

Analytic functions

Objective

You will learn various advanced built-in functions that would come in handy in your day-to-day analytical and reporting requirements. You would be able to rank a dataset, get the aggregated value and out of partitioned dataset, you would be able to traverse through a dataset and be able to retrieve the first, next, previous, and last value of partitions from the dataset.

Window functions

As we all know, a table can have multiple rows, in the sense lot, a huge lot of rows. General aggregate functions operate on the entire table. But *what if we need to aggregate the set of rows?* It is where the window functions also called the **Windowing functions** come into play.

There are various categories of window functions. We'll talk about them as we'll progress in this chapter. We'll start with aggregate windowing functions and understand more of them. The concepts and basics discussed here will apply to all other categories of windowing functions.

Aggregate functions

Window functions operate on a set of rows called A table can have multiple partitions. Partitioning is done based on the column(s) assigned to the **PARTITION BY** clause. We'll understand this overall theory with the help of the following example. This is a very familiar example of a purchase order we've already talked about in the previous chapters too:

too:
too:
too:
too:
too:
too:

Table 7.1: *PurchaseOrder table*

You must be wondering *when and why to use these functions?*

Let's say, we want the running total, average, and so on in the same example as mentioned in [table](#) The output will look like as shown in the following table:

table: table: table: table:
table:
table:
table:
table:
table:

table:

Table 7.2: *Running total and average example*

Output as shown in [table 7.2](#) can be fetched with the help of windowing aggregate functions. This is an example of a simple windowing function without the partition.

Let's say, now we want a subtotal of the **ProductID** against respective rows of the sample example mentioned in [table](#)
There are three

has only one row with amount of

has two rows with amounts of **2,20,000.00** and **2,25,000.00**, respectively. Summing up these will make

Similarly, also has **2** rows with amounts as **2,00,000.00** and
Summing up these will make The output will look like as follows:

follows: follows: follows:
follows:
follows:
follows:
follows:
follows:

Table 7.3: Subtotal example

Output as shown in [table 7.3](#) can also be fetched with the help of windowing functions. This is also an example of a windowing function with partition. Here the partitioning was done on **ProductID** column. So out of five rows, there are three partitions, each for individual You can partition the table based on n number of columns.

You must have got an overview of what the windowing functions are. It is important to note that windowing functions are nothing different. They are normal aggregate functions turned into a windowing functions. A simple syntax change turns a normal aggregate function into windowing function.

When **OVER()** clause is supplied to aggregate functions such as and it becomes a window or windowing function. **OVER()** clause accepts two arguments – **PARTITION BY** and **ORDER BY** **PARTITION BY** is optional but **ORDER BY** is mandatory.

You would be wondering *what is the purpose of ORDER BY?* That's a very valid question. I would refer to the example mentioned in [table 7.1](#) again. *Is it possible to get the correct running total, without supplying the sort order?* I would say Because in order to calculate the running total, average, and so on, it is important to know the starting point, current position, and the sequence in which the summation should happen.

The starting point is always the value of the first row from the sorted result set. The current position is the current row. All the calculation of windowing functions runs from these two most important components, most importantly derived with the help of sorting or **ORDER**

Sort order is of two types – **ascending** and The following tables depicts the working of the windowing function with ascending sort order. The following example represents the working of We are calculating the running total of the **Amount** column in the example mentioned in the following table.

You can see the formula in the **Running total formula** column in the following table that goes like

Where:

SUM(Amount) represents the summation of the **Amount** column.

Starting_point represents the starting point.

Current_position represents the current position.

Let's understand the **starting_point** and **current_position** in the context of the example discussed in [table](#). It is an example of ascending sort order in the **OVER()** clause of the windowing function. In the example mentioned in [table](#), the sort order is specified on the **OrderID** column in the ascending order.

If we'll sort the **OrderID** in the ascending order then **Order ID 1** is the first value. **OrderID 1** is the starting point. Let us now understand *how the running total would be calculated?* It is recommended to relate the following explanation with the details available in [table](#).

Running total of **OrderID 1** will be arrived by summing up the amount of only **OrderID**. In this case, the current position is **OrderID**.

Running total of **OrderID 2** will be arrived by summing up the amounts of **OrderID 1** and In this case, the current position is **OrderID**.

Running total of **OrderID 3** will be arrived by summing up the amounts of **OrderID** and In this case, the current

position is **OrderID**

Running total of **OrderID 4** will be arrived by summing up the amounts of **OrderID** and In this case, the current position is **OrderID**

Similarly, the calculation will go on for all the further rows:

rows: rows:
rows:
rows:
rows:
rows:

Table 7.4: *Illustration of running total with ascending sort order*

If you’ve understood the behavior of running total (**SUM windowing function**) as we discussed in the preceding table, you can easily understand the other aggregate functions to derive:

Running count (**COUNT windowing function**)

Running average (**AVG windowing function**)

Running minimum (**MIN windowing function**)

Running maximum (**MAX windowing function**)

The underlying logic of starting point and current position would remain the same. It's just the aggregate function would be different.

For example:

COUNT(Amount)(Starting_point:Current_position)

AVG(Amount)(Starting_point:Current_position)

MIN(Amount)(Starting_point:Current_position)

MAX(Amount)(Starting_point:Current_position)

In the previous example, we understood the running total calculation using the **SUM** windowing function with sort order defined on **OrderID** column in ascending order. Let us now understand how the same example will look like if we'll define the sort order on the **OrderID** column in the descending order.

The following tables depicts the working of the windowing function with descending sort order.

If we'll sort the **OrderID** in the descending order then **Order ID 4** is the first value. **OrderID 4** is the starting point. Let us

now understand *how the running total would be calculated?* It is recommended to relate the following explanation with the details available in [table](#)

Running total of **OrderID 4** will be arrived by summing up the amount of only In this case, the current position is **OrderID**

Running total of **OrderID 3** will be arrived by summing up the amounts of **OrderID 4** and In this case, the current position is **OrderID**

Running total of **OrderID 2** will be arrived by summing up the amounts of **OrderID** and In this case, the current position is **OrderID**

Running total of **OrderID 1** will be arrived by summing up the amounts of **OrderID** and In this case, the current position is **OrderID**

Similarly, the calculation will go on for all the further rows:

rows: rows:
rows:

rows:
rows:
rows:

Table 7.5: *Illustration of running total with descending sort order*

The **ORDER BY** argument of the **OVER** clause defines the sort order in which the rows are to be treated for the respective functions – and In the absence of **ORDER** you would get the running values such as running total, running average, and so on, but it's correct we can't guarantee. We may also get the correct result but that will be luck.

SQL Server made **ORDER BY** clause mandatory so that we don't have to rely on our luck, and we get the desired result.

It's time to explore various windowing ranking functions and learn their syntax:

to be aggregated> **BY** columns>] separated order by columns>

to be aggregated> **BY** columns>] separated order by columns>

to be aggregated> **BY** columns>] separated order by columns>

to be aggregated> **BY** columns>] separated order by columns>

to be aggregated> **BY** columns>] separated order by columns>

T-SQL example of aggregate windowing functions

Following is the T-SQL query depicting the implementation of aggregate windowing function:

```
SELECT OrderID
, CustomerID
, Amount
, BY CustomerID ORDER BY OrderID AS [SUM]
, BY CustomerID ORDER BY OrderID AS [COUNT]
, BY CustomerID ORDER BY OrderID AS [AVG]
, BY CustomerID ORDER BY OrderID AS [MIN]
, BY CustomerID ORDER BY OrderID AS [MAX]
FROM PurchaseOrder
WHERE OrderID <= 5
```

Executing the preceding T-SQL query will return the result as can be seen in the following screenshot:

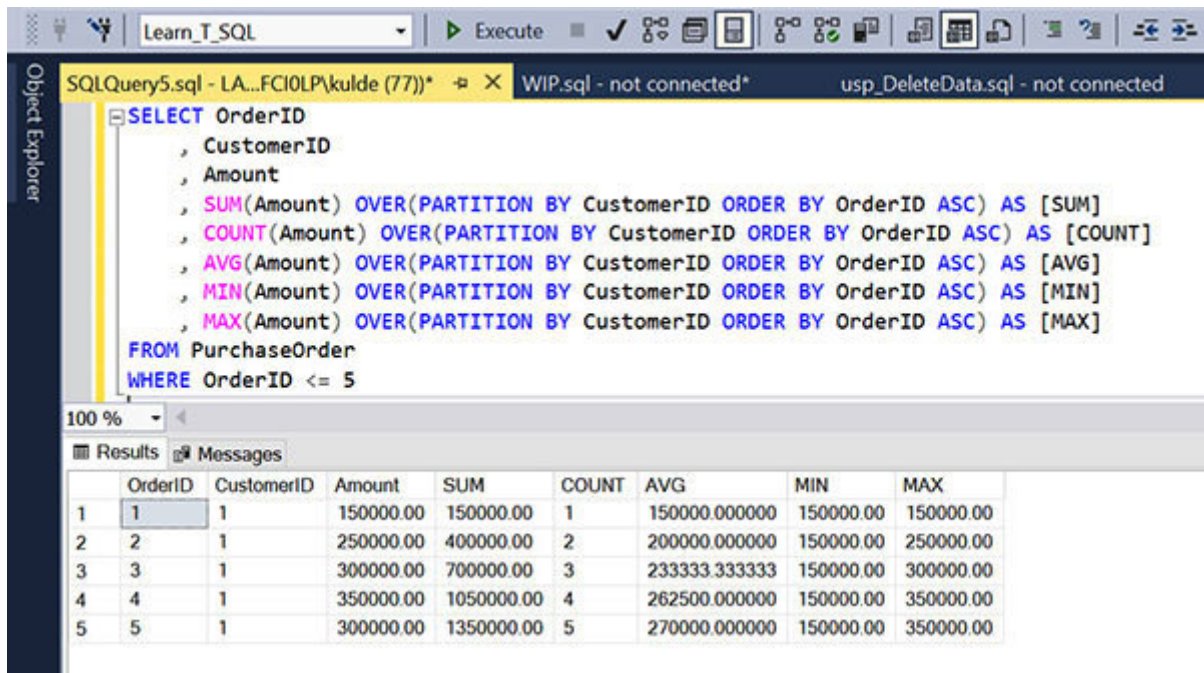


Figure 7.1: Output of query with aggregate window function

You would have observed that we've frequently used *windowing* words with aggregate functions. The use of **OVER** clause with an aggregate function makes it windowing function. It is very important to know that aggregate windowing functions are different than normal aggregate functions.

Here are few other differences which are worth knowing:

Aggregate function applies on the filtered rows of a table. Whereas, aggregate windowing functions apply to the partition. If no partitioning is defined in the windowing function, then the entire table is considered a single partition.

Aggregate function requires a **GROUP BY** clause for all non-aggregated columns in the **SELECT** list. Whereas, aggregate windowing function doesn't require a **GROUP BY** clause, for non-aggregated columns. However, it is to be noted that, if any aggregate function is used in the **SELECT** list along with the aggregate windowing function, then it becomes mandatory to specify **GROUP BY** clause with all the non-aggregated columns. Even the column used with an aggregate windowing function is considered non-aggregated column.

Aggregate functions can be used in the **HAVING** clause to filter the aggregate result. Whereas, you cannot use the **HAVING** clause with aggregated windowing functions.

Ranking functions

Ranking functions are a kind of window functions used to generate the ranks for the rows. Here as well we can use partitioning and generate the ranks by partitioning the table. We can generate the rank out of the whole table, or we can partition the table and generate the rank out of the partitions.

There are various ranking functions – and The purpose of each of them is the same – to generate the ranks. Ranks are self-explanatory. Rank can be understood as the unique integer assigned to each row. But when the partition is used, the rank will be unique for each partition. So, it's possible to have the same rank amongst different partitions. The behavior of all these ranking functions is different. So, it is also possible that some of these functions will generate the same rank value even in a partition. We'll see it in the further explanations.

Although, all these ranking functions generate the rank, but the way they generate rank is different. We can understand it with the following table showing rank generated with different ranking functions.

The following table shows the values generated by each kind of ranking function with partitioning column as **CustomerID** and sorting based on **OrderID** column in ascending order.

If you'll notice, the rank values of and **DENSE_RANK** are exactly the same:

same:

same:
same:
same:
same:
same:

Table 7.6: Illustration of ranking functions with sorting on OrderID in ascending order

The following table shows the values generated by each kind of ranking function with partitioning column as **CustomerID** but sorting based on **Amount** column in ascending order. Please note, the raw data that include and **Amount** columns have exactly the same values as that of the earlier example as shown in [table](#)

If you'll notice here, the rank values generated by each ranking function are different:

different:
different:

different:
different:
different:
different:

Table 7.7: *Illustration of ranking functions with sorting on Amount in ascending order*

Since you've now seen the working of each of these ranking functions, now we can define and understand the theories behind each of them.

ROW_NUMBER()

ROW_NUMBER() function as the name suggests assigns the row number to each row, but the important thing to note is the row number or the rank value generated will be always unique in a partition. If no partition is defined then the entire table is considered as one partition.

ORDER BY plays an important role in all kinds of windowing functions, including ranking functions and also in The rank value is tightly dependent on the column values of the sorting columns. If the sorting column value is and the sort order is ascending then the rank values will be **1** for **2** for and **3** for respectively. But with descending sort order on the same dataset will return **1** for **2** for and **3** for

Similarly, if the sorting column value is and the sort order is ascending even then the rank values will be **1** for **2** for and **3** for respectively. Making the sort order descending will generate the rank values like **1** for **2** for and **3** for 100.

You must be wondering why there is no difference for the same set of values. *That's right!* This is the beauty of this function. It will always generate a new and unique rank for each row, irrespective of whether the values are same or not. In case of same values in the sorting column, SQL Server

randomly treats either of the first and remaining next, and so on.

Syntax of **ROW_NUMBER**

```
ROW_NUMBER () OVER ( BY columns>] ORDER BY  
separated order by columns> ASC | DESC )
```

RANK ()

You would be feeling confused with names of the ranking functions and it is obvious if you feel so! I was also confused when I started with these.

I'll give you hints that you can refer to differentiate them. **RANK ()** also generates the rank values for each partition, similar to **ROW_NUMBER ()** but here the rank value generated will be the same for the same sorting column(s) values. It also has one more distinctive difference and that is if sorting column(s) of two rows have the same values then although the rank values will be same for both these two rows but the next row will have rank value as the *previous rank value + number of* We'll understand it with the help of the following examples.

If the sorting column value is and the sort order is ascending then the rank values will be **1** for **2** for and **3** for **102** respectively. But with descending sort order on the same dataset will return **1** for **2** for and **3** for

Similarly, if the sorting column value is and the sort order is ascending Here, the value **101** is twice. Hence **101** has the repetition once. In this scenario, the rank values will be **1** for **2** for and **2** for but **4** for respectively. Making the sort order

descending will generate the rank values like **1** for **2** for **2** for and **4** for

Syntax of **RANK**

```
RANK ( ) OVER ( BY columns>] ORDER BY separated order  
by columns> ASC | DESC )
```

DENSE_RANK ()

DENSE_RANK () and **RANK ()** are exactly the same with just one difference and that is – **RANK ()** increments the next rank value with the *previous rank value + number of* but **DENSE_RANK** increments the rank value with the *previous rank value* + In the case of repetitions of the rank values really do not matter.

It'll generate the same rank value for the same set of sort column(s) values, but the moment the value will be different, it will increment the rank value by We'll understand it with the help of the following examples.

If the sorting column value is and the sort order is ascending Here the value **101** is twice. Hence **101** has the repetition once. But the rank values will be **1** for **2** for **2** for and **3** for respectively. Making the sort order descending will generate the rank values like **1** for **2** for **2** for and **3** for

Syntax of **DENSE_RANK**

DENSE_RANK () **OVER** (**BY** columns>] **ORDER BY** separated order by columns> **ASC** | **DESC**)

NTILE ()

NTILE () is also a ranking function. It distributes rows of an ordered partition into a specified number of approximately equal groups.

NTILE () function is not very famous and favorite amongst the SQL Server developers and administrators. At least, this is what we've seen so far. But that doesn't imply it's not useful. *It's very useful indeed for the use-case it's meant for!*

You can read and learn about **NTILE ()** function and explore more of ranking functions on the Microsoft official documentation at the following URL:

<https://docs.microsoft.com/en-us/sql/t-sql/functions/ranking-functions-transact-sql?view=sql-server-ver15>

T-SQL example of ranking functions

We hope you understood the differences between **ROW_NUMBER RANK** and **DENSE_RANK ()** functions clearly. Let's now see how the query looks like for the example we've discussed in [tables 7.6](#) and

The T-SQL query to get the data as mentioned in [table 7.6](#) will look like as follows:

```
SELECT OrderID
, CustomerID
, Amount
, BY CustomerID ORDER BY OrderID AS [ROW_NUMBER]
, BY CustomerID ORDER BY OrderID AS [RANK]
, BY CustomerID ORDER BY OrderID AS [DENSE_RANK]
FROM PurchaseOrder
WHERE OrderID <= 5
```

You would see a filter condition **OrderID <= 5** That's to match the table output as mentioned in [table 7.6](#) since we've only 5 orders from **OrderID 1** to **OrderID 5**. When you'll run this query, it'll return the same result as available in [table 7.6](#)

The T-SQL query to get the data as mentioned in [table 7.7](#) will look like as follows:


```
SELECT OrderID
, CustomerID
, Amount
, BY CustomerID ORDER BY Amount AS [ROW_NUMBER]

, BY CustomerID ORDER BY Amount AS [RANK]
, BY CustomerID ORDER BY Amount AS [DENSE_RANK]
FROM PurchaseOrder
WHERE OrderID <= 5
```

You would see a filter condition **OrderID <= 5** here too. The purpose of this filter condition is the same here as well, to match the table output as mentioned in [table](#) since we've only **5** orders from **OrderID 1** to When you'll run this query, it'll return the same result as available in [table](#)

Analytic functions

We again land up to a new category of windowing functions called **analytical**. You would be thinking again – *what is analytical function and why it is categorized separately?* Possibly they would have been released by Microsoft to enhance the BI (part of OLAP) capabilities of their SQL Server product, and that's why called analytic functions.

If you remember, we learnt that SQL Server can be used for both OLTP and OLAP. Although, the feature of SQL Server remains the same in either of the implementations. But the end purpose and domain are different. Microsoft has done quite descent enhancements and bundled many things to offer as a unified BI solution.

Although, these functions are named as analytic functions, but they can also be used in OLTP workload. Similarly, the aggregate and ranking windowing functions can also be used in OLAP workload and vice versa. Even the aggregate and ranking windowing functions are used for analytics and OLAP, which is actually a kind of analytical workload.

The great *Shakespeare* has said, *What's there in the name?* Please don't be confused with the names and categorization of the functions. Understand them and their purpose. Most

importantly, understand the differentiating factors, so that you do not get confused when you want to use them.

There are various analytic functions, each of them having their specific purpose. We'll talk about four of them in the chapter. They are really cool and can come in handy in various scenarios. They help to reduce the various lines of code. In the absence of these functions, you'll have to write a big query/code to achieve the same purpose.

You would be feeling excited to know them, *isn't it?* Anyone would be excited to know them after such a good compliment. They are **FIRST_VALUE** **LEAD** **LAG** and **LAST_VALUE** Although, we'll talk about them individually. Still to make you comfortable, here is the purpose of each of them:

FIRST_VALUE Returns the first value of the partition.

LEAD Returns the next value, after the current row, of the partition.

LAG Returns the previous value, before the current row, of the partition.

LAST_VALUE Returns the last value of the partition.

It can be better understood with the help of the following table, showing our familiar raw data, and values of each

function against each row:

row:
row:
row:
row:
row:
row:

Table 7.8: *Illustration of analytic functions*

These functions also operate in the same way the aggregate windowing function does. Here as well you need to supply the column name to function as an argument. The column specified as the argument is the column on which these functions operate.

Here is the syntax of the analytics functions we've talked about:

FIRST_VALUE (whose first value to be retrieved > BY
columns>] separated order by columns>

LEAD whose next value to be retrieved > BY columns>]
separated order by columns>

whose previous value to be retrieved > BY columns>]
separated order by columns>

LAST_VALUE whose last value to be retrieved > BY columns>]
separated order by columns>

If you'll compare the syntax with that of the aggregate
windowing function, then you'll find them almost the same.
It's just the name of function is different.

T-SQL example of analytic functions

We've talked about the analytic functions. We've also learnt various types of analytic functions. We've understood the syntax and seen the example. We've observed that the syntax of these functions is similar to aggregate windowing functions.

It's the time that we talk about the T-SQL query to implement these functions. We'll not take a new example. Instead, we'll write the T-SQL query to fetch the exact data as we've seen in [table](#)

Following is the T-SQL query showing the implementation of and **LAST_VALUE()** functions:

```
SELECT OrderID
, CustomerID
, Amount
, BY CustomerID ORDER BY OrderID AS [FIRST_VALUE]
, BY CustomerID ORDER BY OrderID AS [LEAD]
, BY CustomerID ORDER BY OrderID AS [LAG]
, BY CustomerID ORDER BY OrderID AS [LAST_VALUE]
FROM PurchaseOrder
WHERE OrderID <= 5
```

The output of this query will match exactly with the data available in [*Table*](#)

We've talked about `FIRST_VALUE()`, `LEAD()`, `LAG()`, and `LAST_VALUE()` functions. They are widely used by database developers and administrators. There are other sets of analytic functions, which we have not considered in this chapter. You can read about them on the Microsoft official documentation at the following URL:

<https://docs.microsoft.com/en-us/sql/t-sql/functions/analytic-functions-transact-sql?view=sql-server-ver15>

Conclusion

Windows or windowing functions come in variety, each for specific purposes such as aggregate windowing functions, ranking functions, and analytic functions. They come in handy and are extremely helpful for complex data analysis.

These functions can also be used in normal requirements. They can help minimize the number of lines of code and the dependent code stabilization efforts. For example, in the absence of a ranking function, you'll have to write your own custom logic for ranking, and also spend efforts to test and stabilize it.

You may find the purpose and benefits discussed for each of these functions very familiar. For example, in an educational intuition, there may be a requirement to show the list of students with their rank. A company needs to generate a list of agents in the order of their sales target achievements and distribute the bonus accordingly. In these examples, ranking and analytic functions may be useful.

There are many other examples too where you will find this function useful. If a company needs to generate the balance sheet, then the aggregated windowing functions can come handy to get the aggregated results at various levels such as by month, quarter, half-yearly, or year. In the absence of

aggregated windowing functions, one will have to write the custom logic, which would be lengthy and complex.

In the next chapter, we'll learn to deal with semi-structured data in the form of XML and JSON using T-SQL.

Points to remember

Use of **OVER** clause with a function is the hint that the function is a window or windowing function.

Applying **OVER** clause to aggregate functions such as and makes it a window or windowing function.

OVER clause accepts two arguments – **PARTITION BY** and **ORDER**. Both of these arguments accept multiple columns. That is, you can assign multiple columns in both of these arguments.

PARTITION BY in **OVER** clause is optional. If so, it means the entire table is considered one partition.

Aggregate function applies on the filtered rows of a table. Whereas, aggregate windowing functions apply on the partition. If no partitioning is defined in the windowing function, then the entire table is considered as a single partition.

Aggregate function requires a **GROUP BY** clause for all non-aggregated columns in the **SELECT** list. Whereas, aggregate windowing function doesn't require a **GROUP BY** clause, for non-aggregated columns.

If any aggregate function is used in the **SELECT** list along with the aggregate windowing function, then it becomes mandatory to specify **GROUP BY** clause with all the non-aggregated columns.

The column used with an aggregate windowing function is considered non-aggregated column.

Aggregate functions can be used in the **HAVING** clause to filter the aggregate result. Whereas, you cannot use **HAVING** clause with aggregated windowing functions.

ROW_NUMBER () function generates unique rank values for a partition.

DENSE_RANK () and **RANK ()** functions can generate duplicate rank values for a partition, if the column(s) used in **ORDER BY** clause has duplicate values.

RANK () increments the next rank value with the *previous rank value + number of*

DENSE_RANK increments the rank value with the *previous rank value* + In the case of repetitions of the rank values really do not matter.

Multiple choice questions

If we need the first value of a column against each row of the partition, which function to use?

DENSE_RANK ()

LEAD ()

LAG ()

FIRST_VALUE ()

Which of the following statements is not true about the LAG () function?

It returns the previous value, before the current row, of a partition.

It is an analytic function.

It will return the value for the last row of the partition.

It will return the value for the first row of the partition.

Which of the following statements is not true about the LEAD () function?

It returns the next value, after the current row, of a partition.

It will return the value for the last row of the partition.

It will return the value for the first row of the partition.

It is an analytic function.

If the column used in the ORDER BY of OVER clause has values 1, 2, 2, 2, 3, 4, 4 then what would be the rank values generated by the RANK () function?

1, 2, 2, 2, 3, 4, 4

1, 2, 2, 2, 3, 3, 3

1, 2, 2, 2, 5, 6, 6

1, 2, 2, 2, 3, 1, 4

Answers

D

D

B

C

Questions

Can we use a windowing function without supplying the **PARTITION BY** argument to **OVER** clause?

Which ranking functions may generate the duplicate rank value for a partition?

What is the difference between **LEAD ()** and **LAG**

What is the difference between **RANK ()** and **DENSE_RANK**

What are the distinctive factors between normal aggregate functions and aggregate windowing functions?

Can we use aggregate functions along with windowing function?

Which ranking function to use to generate the unique rank values?

If the column(s) used in the **PARTITION BY** has **10** distinct values then how many partitions will be there?

When it becomes mandatory to use **GROUP** Please describe in detail.

Can we use **HAVING** clause with the windowing function?
Please describe in detail.

Key terms

Ranking functions are **ROW_NUMBER** **RANK** and **NTILE**

OVER clause is used with windowing functions. It accepts two arguments – **PARTITION BY** and **ORDER**

PARTITION BY is used to partition the result-set. Partitioning can be done on multiple columns.

ORDER BY is used to sort the data. Sorting can be done on multiple columns.

Analytic functions are **FIRST_VALUE** **LEAD** **LAG** **LAST_VALUE**

FIRST_VALUE () function returns the first value of the partition.

LEAD () function returns the next value, after the current row, of the partition.

LAG () function returns the previous value, after the current row, of the partition.

LAST_VALUE () function returns the last value of the partition.

CHAPTER 8

Dealing with XML and JSON

Data could be in various forms such as – **structured** , **semi-structured** , and **unstructured** . Structured data mean the data in the proper structure such as in the tables with a fixed set of columns. In the case of semi-structured data, the columns could or could not be fixed. XML and JSON are referred to as semi-structured data. All other data such as Word files, PowerPoint presentation files, image files, audio, and video files, and so on are considered unstructured data. In this chapter, we'll learn to deal with the semi-structured data using T-SQL.

Structure

In this chapter, we will cover the following topics:

Dealing with XML data.

Dealing with JSON data.

Objective

When we deal with data, we cannot say we will only work with relational data. Relational data are actually structured data. You'll also get XML and JSON data and would need to process and query it.

You will learn the methods to deal with semi-structured data in the form of XML and JSON. You would learn to query the XML and JSON data. You would also learn to convert the data from a table to XML or JSON, and vice versa.

Dealing with XML data

In this topic, we'll talk about various features offered by the SQL Server to query XML data. We'll also learn to convert table data to XML and vice versa. Since we are going to talk about querying the XML data, hence let's first understand the XML.

XML data is also referred as XML document. It has various attributes such as document attributes and element attributes. XML Header includes the information of the XML document as a whole, whereas the element attributes are the attributes of the data in the XML document.

The top node is called the **root**. Every node, except the root, has exactly one parent node. A node can have multiple child nodes. If there are no child nodes of a node then it's called a leaf node. Nodes having the same parent node are called sibling nodes.

Following is sample XML data. It represents the **Product** table we've already created in [Chapter 4](#).

The first line that includes the **xml** version is called XML header, and rest is data. XML is all about nodes. and in the following example are called is the root node. is the child node of and are the child nodes of

node is optional in the following example. You may or may not have it, and the XML data will still be valid. But from a presentation perspective, it's better to have descriptive data, for easy interpretation.

Terms *node* and *element* are used interchangeably in the XML data. For example, root element and the root node. They have the same meaning.

XML data can be represented in various formats. The following is another way to represent the aforesaid same data. It could be represented in other formats too.

XML is case sensitive. Case sensitive means upper and lower case of an alphabet would not be same. For example, *A* and *a* would be treated different.

/>
/>
/>
/>
/>

We hope you would have got some idea about the XML data. Let's now understand how to query it using T-SQL.

XQuery.

XQuery in the SQL Server helps to query the XML data. You may have XML data in a table, in the form of a column, or you would get it as an argument. It doesn't matter where the data is. Be it in the variable, parameter, or in a column of a table. XQUERY can be used to query all kinds of XML data, from any object such as variable, parameter, table, output of a view, or function, and so on.

XQuery is very simple, if you know the **SELECT** statement, and have a basic understanding of the fundamentals of relational databases, and the XML. You just need to make use of the **nodes()** method by supplying it with node hierarchy as the argument. The **nodes()** method can be used with any column or variable of type XML.

SQL Server has designated special datatype for XML and is named XML. This datatype is BLOB type, can and store the data up to 2 GB.

In the example of XML data, we've discussed previously, the node hierarchy to fetch the product details would be If you only need **ProductID** then it will be *Isn't it quite simple? Yes, it is!*

The following is the T-SQL example with XQuery. In this example, we've assigned the same XML data discussed previously, to a variable and querying from the variable itself.

Explanation of important considerations for using the XQuery, in the context of the following example. It'll apply everywhere:

The **nodes()** method must be used, on the variable, or column to be queried. The data type of variable or column has to be XML.

The node hierarchy to be supplied as an argument to the **node()** method.

The variable or column with **node()** method to be treated as a table by assigning it an alias. **X(Y)** is the alias in the below example. It has to be in this format only. Although instead of **X** and you can specify your own custom name.

In the **SELECT** statement, while retrieving the XML data, you need to specify the alias in the format **X.Y** (although you can have a different name, but it should match with the alias specified).

The **value()** method should be used to read the data of the XML attribute. Alias should be specified before the **value()** method, in the format as discussed in the preceding point.

The **value()** method accepts two arguments – **XQuery** and

XQuery would be the name of the attribute with a complete path, to be retrieved, followed by a static value. The static value **[1]** is mandatory and represents that only one value is to be returned. This is a kind of hard-coded value you need to specify. In the following scenario, we've already specified the complete path in the **nodes()** method in the **FROM** clause, hence we just need to specify the name of the attribute. **ProductID** and **ProductName** in the following example are the names of the attribute.

XQuery is not just the attribute name or the node path. It's a language in itself. If you are dealing with XML with the help of XQuery, you need to follow the specific designated syntax and rules.

SQLType is the data type of the value (data). If the resulting value is not compatible with the then an error would be returned. and **5** in the following example are the values (data):

DECLARE

SET @XMLData version="1.0"?>

1

Product 1

2

Product 2

3

Product 3

4

Product 4

5

Product 5

```
SELECT ProductID, ProductName  
FROM
```

As we already discussed, the XML data can be presented in different custom formats. But the possible layouts would be only two as following:

The elements have inline attributes. It means all the attributes of an element would be in the same line. For example:

```
ProductID="1" ProductName="Product 1" />
```

The elements will not have inline attributes. It means the attributes of an element may be in different lines. For example:

1

Product 1

The following is the T-SQL example of XQuery for XML data with elements having the inline attributes. In this example too, we've assigned the XML data to a variable and querying from the variable itself.

If you'll closely look at the example below then you would hardly see any difference, except @ is specified before the attribute name. Whereas, in the preceding example we discussed, which do not have the inline attributes, doesn't require @ before the attribute name. This is a major difference that has to be taken care, based on the structure of the XML:

```
DECLARE @XMLData
SET @XMLData = 'version="1.0"?>
ProductID="1" ProductName="Product 1" />
ProductID="2" ProductName="Product 2" />
ProductID="3" ProductName="Product 3" />
ProductID="4" ProductName="Product 4" />
ProductID="5" ProductName="Product 5" />
SELECT AS ProductID, AS ProductName
FROM AS
```

The output of both the queries can be seen in the following screenshot:

	ProductID	ProductName
1	1	Product 1
2	2	Product 2
3	3	Product 3
4	4	Product 4
5	5	Product 5

Figure 8.1: Output of querying XML variable using XQuery

The following is another T-SQL example of XQuery, to query the XML data from a column of a table. The method would remain the same. Instead of a variable name, you'll have to use the column name.

From a querying perspective, the physical table and table variable are the same. That is why we've leveraged the table variable in the following example, instead of creating a physical table. We've inserted two records in the table variable. The first row has five products and the second has only three. But the products in both of these rows are common.

From a syntax perspective, you can see there is only one difference. Here, we have table name in the **FROM** clause and the column name with **nodes()** method is specified in a **CROSS APPLY** statement. Remember, we talked about **CROSS APPLY** in [Chapter 6, Join, Apply, and](#) You cannot use **JOIN** here so you need to only use the **APPLY** clause. You can use either of **CROSS** or **OUTER** apply. It won't make a difference in the case of XQuery, and the result would be the same:

```
DECLARE @MyTable TABLE
(
    Branch
    ,
);
```

```
INSERT INTO @MyTable
(
```

```

Branch
, Products
)
VALUES
(
'London'
, 'version="1.0"?>
ProductID="1" ProductName="Product 1" />
ProductID="2" ProductName="Product 2" />
ProductID="3" ProductName="Product 3" />
ProductID="4" ProductName="Product 4" />
ProductID="5" ProductName="Product 5" />
,
)
,
(
'Mumbai'
, 'version="1.0"?>
ProductID="1" ProductName="Product 1" />
ProductID="2" ProductName="Product 2" />
ProductID="5" ProductName="Product 5" />
,
);

```

```

SELECT
, AS ProductID
, AS ProductName
FROM @MyTable T
CROSS APPLY AS

```

Output of the query can be seen in the following screenshot:

	Branch	ProductID	ProductName
1	London	1	Product 1
2	London	2	Product 2
3	London	3	Product 3
4	London	4	Product 4
5	London	5	Product 5
6	Mumbai	1	Product 1
7	Mumbai	2	Product 2
8	Mumbai	5	Product 5

Figure 8.2: *Output of querying XML column of a table using XQuery*

Filtering data using XQuery.

Since we've all the required columns, both from the table and the XML in the **SELECT** statement, we can use them in the **WHERE** clause too. When I say column, it include the followings:

The columns from the table. For example, **T.Branch** in the following query.

The complete code of the **value()** method applied on the XML data, for a specific XML attribute, derived as the column in the **SELECT** query. For example, **X.Y.value('@ProductID')[1], 'INT')** and **X.Y.value('@ProductName')[1], 'VARCHAR(50)')** in the following query.

The following T-SQL query shows the implementation of **WHERE** clause in the **SELECT** query with XQuery. It is the extension of the last query. Still mentioning the complete query to avoid the confusion:

```
DECLARE @MyTable TABLE
(
,
);
```

```
INSERT INTO @MyTable
```

```
(  
  Branch  
  , Products  
)
```

```
VALUES
```

```
(  
  
  'London'  
  , 'version="1.0"?>  
    ProductID="1" ProductName="Product 1" />  
    ProductID="2" ProductName="Product 2" />  
    ProductID="3" ProductName="Product 3" />  
    ProductID="4" ProductName="Product 4" />  
    ProductID="5" ProductName="Product 5" />  
    '  
  ),  
  (  
    'Mumbai'  
    , 'version="1.0"?>  
      ProductID="1" ProductName="Product 1" />  
      ProductID="2" ProductName="Product 2" />  
      ProductID="5" ProductName="Product 5" />  
      '  
  );
```

```
SELECT
```

```
  , AS ProductID  
  , AS ProductName  
FROM @MyTable T
```

```
CROSS APPLY AS  
WHERE = 'Mumbai'  
AND =
```

Output of the query can be seen in the following screenshot:

	Branch	ProductID	ProductName
1	Mumbai	1	Product 1

Figure 8.3: Output of querying XML column of a table and filtering XML data using XQuery

Other XQuery methods

SQL Server offers few additional methods as a part of XQuery offering. They are as follows:

exists() method: This method returns and Following is the explanation of each of them:

1: Represents and is generally returned if the attribute name supplied exists and has non-empty value (or, has some value).

o: Represents and is generally returned if the attribute name supplied exists but has empty value.

Represents the attribute name supplied does not exists.

You can read more about it at the following URL of Microsoft official documentation:

<https://docs.microsoft.com/en-us/sql/t-sql/xml/exist-method-xml-data-type?view=sql-server-ver15>

modify() method: Modifies the contents of an XML document.

You can read more about it at the following URL of Microsoft official documentation:

<https://docs.microsoft.com/en-us/sql/t-sql/xml/modify-method-xml-data-type?view=sql-server-ver15>

query() method: Returns the XML contents for the specified XQuery.

You can read more about it at the following URL of Microsoft official documentation:

<https://docs.microsoft.com/en-us/sql/t-sql/xml/query-method-xml-data-type?view=sql-server-ver15>

XQuery is the language offering of SQL Server. As a part of XQuery offering, SQL Server provides methods such as `modify()` and `query()` methods to query into XML document. All these methods accept XQuery as the argument. There could be other arguments as well, but XQuery will be definitely one of them.

OPENXML

OPENXML is another feature of SQL Server to transform the XML data into tabular format. The tabular format means the data will be presented in row and column format.

The following is the T-SQL code showing the implementation of OPENXML.

You'll find it easy, as compared to the XQuery. But XQuery has more flexibility than the OPENXML. With OPENXML, you can only convert the data of a particular XML node in the tabular format. Whereas, with XQuery you can do more than what you can do with OPENXML.

XQuery can be used in the SELECT statement to query the XML data of a column in the table. Whereas, OPENXML cannot. If you wish to query the column's XML data, then you need to first assign it to a variable table variable) and then you can transform its XML contents into a tabular format.

The syntax and T-SQL code of OPENXML is quite straight forward. You just need to follow a few specific instructions as follows, and you are done:

Two variables should always exist – one of type **INT** and another of type **VARCHAR**. You can give them the name you want. There is no rule as such when it comes to naming these variables:

INT type variable is document handle. It's the internal representation of XML document and is created by calling the **sp_xml_preparedocument** as shown in the following T-SQL code.

VARCHAR type variable is the actual XML document, holding the XML data/content to be converted in tabular format.

After you've the variable and the value in the **VARCHAR** type variable, you need to write the following line of code. Although, the name of variables can be different, if you choose to name them differently. Rest should be exactly the same as mentioned as follows:

```
EXEC sp_xml_preparedocument @idoc
```

After preparing the document as discussed in the preceding point, you need to write a **SELECT** statement using **OPENXML()** function in the **FROM** clause as shown in the following T-SQL code. The **OPENXML()** function accepts three arguments – **document handle (INT variable declared for the document node path (also called row and The first two arguments as mandatory ones, but the last one which is flag is optional. Optional means even if you'll not supply it, the function would work with default flag**

You then need to specify the table structure using Table structure represents the format in which the data are to be returned. It is to be noted that the data type assigned to the columns in **WITH** statement has to match with the type of data the XML attributes hold. If the type of data is not compatible with the data type assigned to the columns in the **WITH** statement then error would be returned.

Finally, you need to write the following line of code to remove the document. You may have to change the name of the **INT** variable used for document handle, if a different name has been used. Rest will be as is:

```
EXEC sp_xml_removedocument
```

Sequence of the code is important. So, it has to be in the correct order:

```
DECLARE @idoc @doc
SET @doc version="1.0"?>
ProductID="1" ProductName="Product 1" />
ProductID="2" ProductName="Product 2" />
ProductID="3" ProductName="Product 3" />
ProductID="4" ProductName="Product 4" />
ProductID="5" ProductName="Product 5" />
```

```
EXEC sp_xml_preparedocument @idoc
```


WITH

(

);

The output of the query can be seen in the following screenshot:

	ProductID	ProductName
1	1	Product 1
2	2	Product 2
3	3	Product 3
4	4	Product 4
5	5	Product 5

Figure 8.4: Output of querying XML using OPENXML

Table to XML

SQL Server also provides the feature to convert a table data (or, output of a query) into XML. For this purpose, there is a specific syntax called **FOR XML** clause can be used with **SELECT** and **DELETE** statements. It can be even used in the variable value assignment statements.

FOR XML supports the following modes. You can specify one of these modes:

RAW

AUTO

EXPLICIT

PATH

The following T-SQL example shows the use of **FOR XML** clause. We've taken an example of **PATH** modes.

For the mode as we will see two examples, each for - with and without **ELEMENTS** clause.

For the mode as we will see three examples, each for –
without the path with the path specified as and **with the path specified as**

T-SQL example with **FOR XML**

Product

When you'll run the preceding query, the following XML data would be generated. Data in the XML will be purely depending upon the data in the table:

/>

/>

/>

/>

/>

T-SQL example with **FOR XML**

Product

When you'll run the preceding query, the following XML data would be generated. Data in the XML will purely depend upon the data in the table:

/>

/>

/>

/>

/>

T-SQL example with **FOR XML**

Product

ELEMENTS

When you'll run the preceding query, the following XML data would be generated:

T-SQL example with **FOR XML**

Product

When you'll run the preceding query, the following XML data would be generated.

T-SQL example with **FOR XML PATH** (empty):

Product

When you'll run the preceding query, the following XML data would be generated:

T-SQL example with **FOR XML PATH** (non-empty):

Product

When you'll run the preceding query, the following XML data would be generated:

You can read more about **FOR XML** clause and **EXPLICIT** mode at the following URL of Microsoft official documentation:

<https://docs.microsoft.com/en-us/sql/relational-databases/xml/for-xml-sql-server?view=sql-server-ver15>

Dealing with JSON data

JSON too is used for unstructured data, but is quite lightweight as compared to the XML. It is considered as the best substitute for XML. The full form of JSON is **JavaScript Object**

You can read more about JSON at the following official website of JSON: <https://www.json.org/>

The representation of data in JSON is different than XML. The following is a sample JSON for the same **Product** table:

```
[
{
"ProductID": 1,
"ProductName": "Product 1"
},
{
"ProductID": 2,
"ProductName": "Product 2"
},
{
"ProductID": 3,
"ProductName": "Product 3"
},
{
```

```
"ProductID": 4,  
"ProductName": "Product 4"  
},  
{  
"ProductID": 5,  
"ProductName": "Product 5"  
}  
  
]
```

As you can see from the preceding JSON example, the representation of data is quite simple in JSON, as compared to the XML.

JSON is also case sensitive. Case sensitive means the upper and lower case of an alphabet would not be the same. For example, A and a would be treated differently.

JSON is all about array and objects:

An object is an unordered set of name/value pairs. An object begins with { (left brace) and ends with } (right brace). Each name/value pair has a colon as a separator to separate name with value. The left side of colon is called **name (property)** and the right side is called **value (property)**. The property name is specified in the double quotes. The property value is also specified in the double quotes if it contains the data of type string or date time. There could be multiple objects in a JSON.

The following is an example of objects from the aforesaid JSON data. **ProductID** and **ProductName** are the names (property names), and **5** and **Product 5** are the values (property values):

```
{  
  "ProductID": 5,  
  "ProductName": "Product 5"  
}
```

An array is an ordered set of values. An array begins with [(left bracket) and ends with] (right bracket). Values in an array are separated by , (comma). A value can be an object, or an array, a string (in double quotes), or a number, or true or false or null.

I hope you would have got some idea about the JSON. Let's now understand how to query it using T-SQL.

OPENJSON

OPENJSON is the feature of SQL Server to transform the JSON data into a tabular format. *Do you remember the OPENXML used to transform the XML data into a tabular format?* OPENJSON too has the same purpose as OPENXML. But its implementation is simple, as compared to OPENXML.

In OPENXML, you need to have a document handle. You also need to prepare the document using **EXEC** before using the OPENXML, and you need to remove the document at the end using **EXEC**. If you remember, you also need to define the resulting structure using

But here in OPENJSON, you do not need to have a document handle. You also do not need to prepare the document, and so the removed document is not needed. You can define the explicit structure with the help of but that is not compulsion. If **WITH** is not specified in OPENJSON then the output will be in the default structure. The default structure has three columns – and You may not need the data in default structure, so you need to explicitly define the structure of the output you seek.

The following is the T-SQL code showing the implementations of OPENJSON. As we discussed, there are two options to use

OPENJSON – **one with default structure** and **another with explicit** We'll see each of them one by one.

Let us first see the sample T-SQL code for OPENJSON with the default structure. The following is an example:

```
DECLARE @json
SET @json = '[

{
"ProductID": 1,
"ProductName": "Product 1"
},
{
"ProductID": 2,
"ProductName": "Product 2"
},
{
"ProductID": 3,
"ProductName": "Product 3"
},
{
"ProductID": 4,
"ProductName": "Product 4"
},
{
"ProductID": 5,
"ProductName": "Product 5"
}
]'

SELECT *
FROM OPENJSON
```

The output of the preceding T-SQL code will be similar to [Figure](#)

	key	value	type
1	0	{ "ProductID": 1, "ProductName"...	5
2	1	{ "ProductID": 2, "ProductName"...	5
3	2	{ "ProductID": 3, "ProductName"...	5
4	3	{ "ProductID": 4, "ProductName"...	5
5	4	{ "ProductID": 5, "ProductName"...	5

Figure 8.5: Output of querying JSON using OPENJSON without specifying the structure

Let us now see the sample T-SQL code for OPENJSON with explicit structure using **WITH** clause. Following is an example:

```
DECLARE @json
SET @json = '[
{
"ProductID": 1,
"ProductName": "Product 1"
},
{
"ProductID": 2,
"ProductName": "Product 2"
},
{
"ProductID": 3,
"ProductName": "Product 3"
```

```

},
{
  "ProductID": 4,
  "ProductName": "Product 4"
},
{

  "ProductID": 5,
  "ProductName": "Product 5"
}
}
SELECT * FROM
OPENJSON
WITH
(
  ProductID
);

```

	ProductID	ProductName
1	1	Product 1
2	2	Product 2
3	3	Product 3
4	4	Product 4
5	5	Product 5

Figure 8.6: Output of querying JSON using OPENJSON by specifying the structure

The preceding query of explicit structure can be also written as follows. It will return the same result. Here, you have the flexibility to map the columns with the resulting property name (with path) from the JSON. When doing so, you need

to assign the name (with hierarchy) in the following format, within single quotes

`$.property name>.property name>`

\$ (dollar) and dot are hardcoded. Parent and child property names should be specified according to the JSON. If the parent/child is an array then you need to also specify the index (position of the object in an array). If there are multiple parents then you need to accordingly specify all the parents (in the sequence of their hierarchy), with index (if applicable). Each parent should have a separator dot

In the JSON example, we've discussed so far, we don't have the parent. So, we do not need to specify the parent. But when there will be a parent, you need to specify it. This becomes very useful when dealing with hierarchical JSON data:

```
DECLARE @json
SET @json = '[
{
"ProductID": 1,
"ProductName": "Product 1"
},
{
"ProductID": 2,
"ProductName": "Product 2"
},
{
```

```
"ProductID": 3,  
"ProductName": "Product 3"  
},  
{  
"ProductID": 4,  
"ProductName": "Product 4"  
},  
{
```

```
"ProductID": 5,  
"ProductName": "Product 5"  
}
```

```
SELECT * FROM
```

```
OPENJSON
```

```
WITH
```

```
(  
ProductID    '$.ProductID' ,  
);
```

Other JSON functions

SQL Server offers few additional methods as part of XQuery offering. They are:

ISJSON() function: Tests whether the supplied expression (as argument) is a valid JSON, and return **1** or **0** or

1: Represents the expression is a valid JSON.

0: Represents the expression is not a valid JSON.

If the expression itself is null.

You can read more about it at the following URL of Microsoft official documentation:

<https://docs.microsoft.com/en-us/sql/t-sql/functions/isjson-transact-sql?view=sql-server-ver15>

JSON_VALUE() function: Returns the value of the supplied path. It accepts two arguments – **expression (JSON)** and
Following is the sample T-SQL code showing the implementation of **JSON_VALUE()** function.

In the following example, we have an array of objects. The array has total of five objects. Index always starts with where **0** means the first entry in an array. As you can see in the following T-SQL code, we have specified It means we are referring to the **ProductID** property of the first object in the array. The following T-SQL code will return the value of the supplied property path, which is the **ProductID** of the first object in the array:

```
DECLARE @json
SET @json
{
  "ProductID": 1,

  "ProductName": "Product 1"
},
{
  "ProductID": 2,
  "ProductName": "Product 2"
},
{
  "ProductID": 3,
  "ProductName": "Product 3"
},
{
  "ProductID": 4,
  "ProductName": "Product 4"
},
{
  "ProductID": 5,
  "ProductName": "Product 5"
}
```


You can read more about it at the following URL of Microsoft official documentation:

<https://docs.microsoft.com/en-us/sql/t-sql/functions/json-value-transact-sql?view=sql-server-ver15>

JSON_MODIFY() function: Updates the value of a property in a JSON string and returns the updated JSON string. It accepts three arguments – **expression (the original path (of the** and **new**

Following is the sample T-SQL code showing the implementation of **JSON_MODIFY()** function. The following T-SQL code will update the value of the **ProductID** property of the first object to **2** (new value) in the array and will return the updated JSON string:

```
DECLARE @json
SET @json
{
  "ProductID": 1,
  "ProductName": "Product 1"
},
{
  "ProductID": 2,
  "ProductName": "2"
},
{
```

```
"ProductID": 3,  
"ProductName": "Product 3"  
},  
{  
"ProductID": 4,  
"ProductName": "Product 4"  
},  
{  
"ProductID": 5,  
"ProductName": "Product 5"  
}
```

You can read more about it at the following URL of Microsoft official documentation:

<https://docs.microsoft.com/en-us/sql/t-sql/functions/json-modify-transact-sql?view=sql-server-ver15>

JSON_QUERY() function: Extracts an object or an array from a JSON string. The value returned would be a JSON. It accepts two arguments – **expression (JSON)** and Following is the sample T-SQL code showing the implementation of **JSON_QUERY()** function. The following T-SQL code will return the first object from the array:

```
DECLARE @json  
SET @json  
{  
"ProductID": 1,  
"ProductName": "Product 1"
```

```
},  
{  
  "ProductID": 2,  
  "ProductName": "Product 2"  
},  
{  
  "ProductID": 3,  
  "ProductName": "Product 3"  
},  
{  
  "ProductID": 4,  
  "ProductName": "Product 4"  
},  
{  
  
  "ProductID": 5,  
  "ProductName": "Product 5"  
}
```

If a scalar value is to be returned then use **JSON_VALUE()** instead.

You can read more about it at the following URL of Microsoft official documentation:

<https://docs.microsoft.com/en-us/sql/t-sql/functions/json-query-transact-sql?view=sql-server-ver15>

We've seen the implementations of various JSON functions on the variable. But it can be also used on the columns, in the SELECT, INSERT, UPDATE, and DELETE queries.

Table to JSON

The way we can convert a table (or query) output in the XML, similarly, we can also convert a table (or query) output in the JSON string. **FOR JSON** clause is used for this purpose. Following is the sample T-SQL code showing implementation of **FOR JSON** clause:

```
SELECT * FROM Product  
FOR JSON PATH
```

The output of the preceding query will be the JSON string as can be seen as follows:

```
[  
{  
  "ProductID": 1,  
  "ProductName": "Product 1"  
},  
{  
  "ProductID": 2,  
  "ProductName": "Product 2"  
},  
{  
  "ProductID": 3,  
  "ProductName": "Product 3"  
}]
```

]

You can read more on dealing with JSON data in SQL Server at the following URL of official Microsoft documentation:

<https://docs.microsoft.com/en-us/sql/relational-databases/json/json-data-sql-server?view=sql-server-ver15>

Conclusion

XML and JSON are very important data sources. You may have to deal with them when working on a production system. The features, concepts, and functionalities learnt in this chapter would be very useful in dealing with the unstructured data of the form XML and JSON.

This is an ideal example of an evolved database management system. In the absence of the functionalities discussed in this chapter, you may have to search for other solutions, or write the custom logic to process the XML and JSON data. But since SQL Server extends support to XML and JSON data, you do not need to think about the other tool, or write the custom logic. SQL Server is one-stop solution for both relational as well as unstructured data in the form of XML and JSON.

In the next chapter, you will learn variables and control flow statements.

Points to remember

The data type assigned to the columns in **WITH** statement has to match with the type of data the XML attributes holds. If the type of data is not compatible with the data type assigned to the columns in the **WITH** statement then an error would be returned.

Terms *node* and *element* are used interchangeably in the XML data. For example, root element and the root node. They have the same meaning.

Case sensitive means upper and lower case of an alphabet would not be same. For example, *A* and *a* would be treated different.

SQL Server has designated special datatype for XML, and is named XML. This datatype is BLOB type, can and store the data up to 2 GB.

XQuery is not just the attribute name or the node path. It's a language in itself. If you are dealing with XML with the help of XQuery, you need to follow the specific designated syntax and rules.

XQuery is the language offering of SQL Server. As part of XQuery offering, SQL Server provides methods such as **query()** methods to query into XML document. All these methods accept XQuery as the argument. There could be other arguments as well, but XQuery will be definitely one of them.

XQuery can be used in the **SELECT** statement to query the XML data of a column in the table. Whereas, OPENXML cannot. If you wish to query the column's XML data, then you need to first assign it to a variable **table** variable) and then you can transform its XML contents into a tabular format.

JSON is also case sensitive. Case sensitive means upper and lower case of an alphabet would not be the same. For example, *A* and *a* would be treated differently.

We've seen the implementations of various JSON functions on the variable. But it can be also used on the columns, in the **INSERT** and **DELETE** queries.

There is no data type dedicated for JSON. JSON is a string that can be stored in **VARCHAR** or **NVARCHAR** data types.

Multiple choice questions

XQuery is used for?

Querying XML data.

Querying JSON data.

Can we use OPENXML without WITH clause?

Yes.

No.

Which of the following JSON function should be used for retrieving the scalar values?

ISJSON() function.

OPENJSON() function.

JSON_VALUE() function.

JSON_QUERY() function.

Which of the following type of data requires double quotes in JSON?

Integer.

Decimal.

Boolean.

Date and/or time.

Answers

A

B

C

D

Questions

What are the different features to query the XML data?

What is the data type offered by SQL Server to store XML data?

When not to use OPENXML?

What is the difference between XQuery and OPENXML?

What are the different options that FOR XML supports?

Can we modify an XML document using T-SQL? If yes, then how?

What is the explicit structure and how to specify it with OPENJSON?

Can we use OPENJSON without an explicit structure?

Compare **JSON_QUERY()** and **JSON_VALUE()** function?

How to check if a JSON string is valid?

How to query a particular object in an array?

Key terms

XQUERY is the language in itself. If you are dealing with XML with the help of XQuery, you need to follow the specific designated syntax and rules. It offers the following methods:

exists() method

modify() method

query() method

nodes() method

value() method

exists() method returns and Following is the explanation of each of them:

1: Represents and is generally returned if the attribute name supplied exists and has non-empty value (or, has some value).

o: represents and is generally returned if the attribute name supplied exists but has an empty value.

Represents the attribute name supplied does not exists.

modify() method modifies the contents of an XML document.

query() method returns the XML contents for the specified XQuery.

nodes() method is used to select the specific nodes in a XML document based on the supplied path.

value() method is used to get the value of the supplied attribute from the XML document.

OPENXML is the feature of SQL Server to transform the XML data into a tabular format.

FOR XML clause is used to convert the table (or query) data into XML.

OPENJSON is the feature of SQL Server to transform the JSON data into tabular format.

ISJSON() function is used to test whether the supplied string is a valid JSON.

JSON_VALUE() function is used to fetch the value of the supplied path from a JSON string.

JSON_MODIFY() function updates the value of a property in a JSON string and returns the updated JSON string.

JSON_QUERY() function extracts an object or an array from a JSON string. The value returned would be a JSON.

FOR JSON clause is used to convert the table (or query) data into JSON.

CHAPTER 9

Variables and Control Flow Statements

*Remember the full form of T-SQL? It's **Transact – Structured Query Language** . We have learnt querying so far, now let's get started with the language and programming. In this chapter and onwards, we'll learn the programmability features and objects.*

In this chapter, you will learn about the variables. In any programming language, variables are the founding stone. It's nearly impossible that any programming language exists without variables. You will also learn about loop, case, and control flow statements. All these features form the basics of programming.

If you are familiar with any programming language, then you will find the concepts discussed in this chapter nothing different. Concepts of programming are the same everywhere, in every programming language. The difference is just the syntax.

Structure

In this chapter, we will cover the following topics:

Variables

Tiny, yet powerful keywords

CASE statement

IF statement

WHILE loop

Objective

By the end of this chapter, you will know all the features that form the basics of T-SQL programming. You'll be able to make use of the variables. You'll be able to implement the loops for performing iterations. You will be able to implement the case statement to check for specific conditions in the query (more specifically in the select, insert, and update statements) and return the desired values. Similarly, you will be able to make use of the control flow statement to define the flow of the code. For example, if a particular condition is true then do X else

You can build huge programs with the help of queries learned so far, and with these programming features.

Variables

Variables can be treated as the temporary storage, which has limited scope in which they are defined. Variable declaration too has a similar definition as that of columns. Each variable should have a data type assigned during declaration, similar to the columns. The data types define the type of data that the variable can hold.

A variable has three stages – **declaration of assignment of values in the** and **reading the values from the** Let's understand each of these stages.

Declaring the variables

The **DECLARE** command is used to declare variables. A variable first needs to be declared before it can be used further. All the variable names should have a prefix. The variable name should always start with

In T-SQL, the variable name always starts with @. Anything you see which starts with @ is a variable.

The following is the syntax to declare a variable:

```
DECLARE @name>    type>
```

The following are the examples depicting the variable declaration:

```
DECLARE  
DECLARE  
DECLARE  
DECLARE  
DECLARE
```

The variable **@A** in the preceding example is of type **INT**. It won't accept anything other than an integer value. Variable **@B** is of type of **DECIMAL** and it will accept only decimal values.

Variable **@C** is of type **DATE** and it won't accept anything other than Similarly, other variables will also accept the value according to their data types.

Assigning values to the variables

As we already discussed, the variables are used as temporary storage to store the values for a temporary purpose. There are two commands to assign value to the variables. They are – **SET** and **SELECT** commands.

Further, the value of the variables can be static or dynamic.

The **SET** command is used to assign the static value to the variables. Static values also include the other variables or calculations such as addition, multiplication, subtraction, and division, and so on of static values or variables. When you are assigning another variable as a value to a variable, then it has to be of the same type. It means the variable to which the value is being assigned, and the variable which is being assigned as a value should have the same/compatible data types.

Here is the syntax to assign value to the variables using the **SET** command:

```
SET @name> =
```

```
SET @name> = 1> 2>
```

```
SET @name> = @1 name>
```



```
SET @name> = @1 name> @2 name>
```

Here are the examples showing variable value assignment using the **SET** command:

```
SET @A = 1
```

```
SET @B = 2
```

```
SET @C = @A + @B
```

```
SET @D = 'Test'
```

The **SELECT** command is used to assign the dynamic values to the variable. Dynamic values are the column values of a table. The column assigned as the value to a variable should have the same data type. The **SELECT** command can be also used to assign the static values.

It is to be noted that the variables always hold only a single value unless it is of type table. There is a special type of variable called **table**. Its features and functioning are similar to that of the physical tables, but it does not hold the data permanently. It stores the data in row and column format. As the name suggests, the table variables are the variables of the type table, to store the data in the tabular format.

Since a variable (except the table variables) always store a single value, the **SELECT** statement should return only one row. In case it returns multiple rows, then only the column value of the last row will be assigned to the variable.

Here are the syntaxes to assign value to the variables using the **SELECT** command:

```
SELECT @name> =  
SELECT @name> = 1> 2>
```

```
SELECT @name> = @1 name>  
SELECT @name> = @1 name> @2 name>
```

```
SELECT @name> = name> FROM
```

name>

```
SELECT @name> = 1 name> 2 name>
```

FROM

name>

Here are the examples showing variable value assignment using **SELECT** command:

```
SELECT @CustomerName = CustomerName FROM Customer  
WHERE CustomerID = 1  
SELECT @Amount = Quantity * Rate FROM PurchaseOrder  
WHERE CustomerID = 1
```

Here is the syntax to create a table variable:

```
DECLARE @name> TABLE  
(  
1 name> type>
```

```
, 2 name>          type>
.
.
.
, n name>    type>
)
```

Here is the sample example showing the declaration of table variable:

```
DECLARE @MyTable TABLE
(
Col1    INT
, Col2
, Col3   DATE
)
```

In order to assign the value to the table variable, you need to follow the same syntax, as you followed to insert data in a physical table. Sample example is shown as follows:

@MyTable

Reading values from the variables

The purpose of storing values in a variable is to access it when needed. If we cannot access the value stored in a variable, then it is of no use.

There are various ways a variable can be used. It can be read with the help of the **SELECT** command. It can be printed in the result pane of the SSMS. It can be used as a value to the other variable. It can be used in calculations and so on.

The following are a few T-SQL examples, showing different ways in which, the variables can be read:

```
SELECT @A
SELECT @A AS @B AS * AS [AB]
SET @B = @A
PRINT @A
```

The following are the examples to read the table variable:

```
SELECT * FROM @MyTable
SELECT Col3 FROM @MyTable
```

You can also assign the values from a table variable to other variables.

Tiny, yet powerful keywords

Since we are talking about the T-SQL programming in this chapter, we need to understand few tiny yet powerful keywords. You would use them very frequently when you'll start programming with T-SQL.

Keywords are the specific words designated for the specific purpose by SQL Server. That's the definition of the keywords. However, in the context of T-SQL, we can also treat them as T-SQL commands or statements. So, please don't get confused when you find these terminologies used interchangeably.

BEGIN ... END

It is a T-SQL programmability feature. It is used to group the set of codes. **BEGIN** represents the start of the code block and **END** represents the end of the code block. So, **BEGIN ... END** represents code block (or, block of code).

It is mandatory to specify the T-SQL code of a **WHILE** loop and **IF** statement in the **BEGIN ...** We'll see it in the upcoming topics. It can also be used to logically group the relevant lines of T-SQL code, for better readability, maintainability, and easy debugging of the T-SQL code.

It does not require much explanation. Let's say you have few lines of code and you would like to group it, then you can simply write **BEGIN** keyword before the start of the code, and the **END** keyword after the end of the code. You are done, *that's it!*

Semicolon (;)

It is the statement terminator. It represents that the statement ends here. You can relate the semicolon of T-SQL with the full stop of the English language. It is used to terminate a T-SQL statement, the very same way, the full stop is used to terminate a sentence.

When you see a semicolon in a T-SQL statement, you can treat it as the end of the statement.

RETURN

RETURN seems to be a tiny word, but is very powerful. The **RETURN** keyword implies *stop further*. So, *what did you understand from this?*

Let's understand it with the following example. Consider, we've the following T-SQL code in which we are declaring two variables **@A** and **@B**. We are then assigning the values to both variables and fetching the output from each of them. Wait, there is a **RETURN** statement in between. *What would be the output of the following code? Any guesses?*

DECLARE

DECLARE

SET @A =

SELECT @A

SET @B =

SELECT @B

Well, only one row will be returned, that too for the **@A** variable. Since we've the **RETURN** keyword, which implies *stop the further execution* or *stop* the code written after the

RETURN keyword won't be executed. It is as good as we don't have the code at all after the **RETURN** keyword.

GOTO

It is another classic example of tiny yet powerful keywords. The **GOTO** command can be used to actually go to a specific line of code, during the code execution. For example, you have 10 lines of code, but based on a certain condition at the *fifth* line, you would like to start again from the line of the code. The **GOTO** keyword can be used for this purpose.

Here is the example depicting the implementation of the **GOTO** keyword. You can see This is the placeholder or bookmark for **GOTO** command. You need to specify the bookmark(s) along with the **GOTO** keyword. You can traverse through the bookmark(s) programmatically using but one at a time:

DECLARE

```
SET @A =
```

```
A:
```

```
SET @A = @A +
```

```
SELECT @A
```

```
IF <
```

```
GOTO A
```

The output of the preceding query will be as shown in the following diagram:

	(No column name)
1	1
	(No column name)
1	2
	(No column name)
1	3
	(No column name)
1	4
	(No column name)
1	5

Figure 9.1: Output of the query with GOTO statement

WAITFOR

You must have noticed the red traffic signal. *Isn't it?* A small red color signal can make 100s of vehicles to wait until it becomes green. Don't you think it is also an example of tiny, *yet powerful!*

The **WAITFOR** statement also has similar behavior. As the name suggests, it is used to cause the delay (pause) in the execution. You can either define the time until when you want to wait, or you can also specify the duration for which you want to wait.

For example, you can specify that you want to wait (pause) until a specific time such as 10:00 You can also specify that you want to wait (pause) for a particular duration such as 00:30:00 (30

Consider you have five lines of code. The third line specifies the **WAITFOR** command. In this case, the first two lines will be executed and the third line will cause the delay (pause) for the specified duration/time (whatever is specified). Once the time or duration will elapse, the remaining lines of code (the fourth and fifth lines from the preceding scenario) would be executed.

The syntax is as follows:

WAITFOR TIME
WAITFOR DELAY

Where:

hh is hours.

mm is minutes.

ss is seconds.

nnn is millisecond.

Time and duration can be also specified in **hh:mm** or **hh:mm:ss** formats.

Examples are as follows:

In the following example, the first line will be executed, and then it will wait till **00:35** h at the second line, to execute the third line:

SELECT
SELECT

In the following example, the first line will be executed, and then it will wait for 10 s at the second line, to execute the third line:

```
SELECT
```

```
SELECT
```

CASE statement

Before we talk about the **CASE** statement let's see an example. After this example, you would not need much explanation of the case statements. Suppose you want a banana milkshake, but if it's not available then you would be happy with apple juice. Hypothetically, if both are not available then you would manage with just a glass of water. You would like to specify this priority to the restaurant waiter.

The priority mentioned in the previous example, if converted to a **CASE** statement, will look something like as mentioned follows:

```
CASE
WHEN Item = 'Banana Milkshake'
THEN 1
WHEN Item = 'Apple Juice'
THEN 2
ELSE 3
END
```

This **CASE** statement can also be alternatively written as follows:

```
CASE Item
WHEN 'Banana Milkshake'
```



```
THEN 1  
WHEN 'Apple Juice'  
THEN 2  
ELSE 3  
END
```

Banana Milkshake is given priority **Apple Juice** as and rest as 3 here means, both **Banana Milkshake** and **Apple Juice** are not available.

Let us see another scenario with similar lines. Consider that you are a restaurateur. Your product catalog has the following products:

Milkshake

Banana milkshake

Juice

Apple juice

You do not want to change your catalog, but wish to print **Banana Milkshake** as just milkshake, and **Apple Juice** as just juice in the invoice to the customer. If we'll convert this into a **CASE** statement, it will look like similar to as shown as follows:

```
CASE
WHEN Item = 'Banana Milkshake'
THEN 'Milkshake'
WHEN Item = 'Apple Juice'
THEN 'Juice'
ELSE Item
END
```

This **CASE** statement can be alternatively be written as follows:

```
CASE Item
WHEN 'Banana Milkshake'
THEN 'Milkshake'
WHEN 'Apple Juice'
THEN 'Juice'
ELSE Item
END
```

By now you would have got some idea of what a **CASE** statement is. The best part of case statement is you can use it in all kinds of **SELECT** and **UPDATE** queries. It can be even used in the variable value assignments and in the **VALUE** section of the **INSERT** statement.

The syntax we've already seen in the previous example. So, would not be repeating the same. Let us understand the components of a **CASE** statement. The following are the key attributes of a case statement:

The **CASE** keyword is used to define the **CASE** statement, and it ends with the **END** keyword. Both these keywords are must.

WHEN keyword is used specify the condition.

THEN keyword is used to specify the action, if the condition of the **WHEN** is true.

ELSE is optional, if not specified then conditions beyond ones specified in the **WHEN** will return

Condition check starts from the top to bottom, hence it's important to specify the conditions in proper sequence.

There could be nested **CASE** statements too. You can add another **CASE** statement in the action

Let us see few T-SQL examples depicting the application of the **CASE** statement.

The following is an example of the **CASE** statement with the **SET** command. As you already know, even **SELECT** can be used instead of the **SET** command. So, the functioning and behavior of the **CASE** statement would be the same in both cases. You can try it out yourselves by replacing **SET** with
Executing this code will return the output as milkshake:

```
DECLARE
```

```
SET @Item = 'Banana
```

```
SET @Item = CASE
```

```
WHEN @Item = 'Banana Milkshake'
```

```
THEN 'Milkshake'
```

```
WHEN @Item = 'Apple Juice'
```

```
THEN 'Juice'
```

```
SELECT
```

A simple change in the input value of the preceding query will result in different outputs, based on the conditions specified therein. Instead of **Banana** we'll supply the input value as *fruit*. When you'll run the query with this input, the resulting output will be fruit salad only.

The following is the example of using a **CASE** statement in the formula:

Suppose there is a discount of 50% for **Banana Milkshake** and you would like to calculate the final price after the discount. The output of the following query would be. However, if you would change the items to anything else (other than **Banana** then the output would be

```
DECLARE @Item
```

```
DECLARE
```

```
DECLARE
```

```
SET 'Banana
```

```
SET
```

```
SET @Quantity =
```

```
SELECT @Quantity * CASE
```

```
WHEN @Item = 'Banana Milkshake'
```

```
THEN - * 50 /
```

The following is the example of using a **CASE** statement in the **INSERT** statement:

```
DECLARE @Item
```

```
DECLARE
```

```
DECLARE
```

```
DECLARE @MyTable TABLE
```

```
(
```

```
Item
```

```
,
```

```
,
```

```
,
```

```
,
```

```
);
```

```
SET @Item = 'Banana
```

```
SET @Price =
```

```
SET @Quantity =
```

```
INSERT INTO @MyTable
```

```
VALUES
```

```
(
```

```
@Item
```

```
, @Price
```

```

, CASE
WHEN @Item = 'Banana Milkshake'
THEN 50
END

, @Quantity
, @Quantity * - *
WHEN @Item = 'Banana Milkshake'
THEN 50
/
SELECT * FROM @MyTable;

```

The output of the preceding T-SQL code would look like as can be seen in the following screenshot:

	Item	Price	Discount	Quantity	Amount
1	Banana Milkshake	10.00	50	2	10.00

Figure 9.2: Output of the query with CASE statement

The previous query can alternatively be written as follows, with no change in the resulting output. By looking at the following query, you can easily relate in what all scenarios you can use it in the and **UPDATE** queries. In our example, we've used the variables. But you can also use the columns in the **CASE** statement. You can use the column in both condition as well as action section of the **CASE** statement:

```

DECLARE @Item
DECLARE

```

DECLARE

DECLARE @MyTable TABLE

(

Item

,

,

,

,

);

SET 'Banana

SET

SET @Quantity =

INSERT INTO @MyTable

SELECT @Item AS Item

, @Price AS Price

, CASE

WHEN @Item = 'Banana Milkshake'

THEN 50

END AS Discount

, @Quantity AS Quantity

, @Quantity * - *

WHEN @Item = 'Banana Milkshake'

THEN 50

/ AS Amount;

SELECT * FROM @MyTable;

Always follow the BODMAS rule when writing a formula for any arithmetic operation, involving multiple arithmetic operations. Appropriately make use of parenthesis to get the right result. You can refer to the use of parenthesis in the formula used in the previous T-SQL code.

IF statement

If you have understood the **CASE** statement, then you would not find it difficult to understand the **IF** statement. From a layman's perspective, the **IF** and **CASE** statements work in the same fashion, but they do have different purposes.

IF statement is used to specify the condition and action, the same very way, **CASE** statement does. But you cannot use the **IF** statement in any query such as and so on. Also, you cannot use the **IF** statement with the **SET** and **SELECT** commands for variable value assignment.

You would be wondering that the **CASE** and **IF** statements have similar functioning, *but IF does not support the features which CASE does, then why to use the IF statement?* That's a very valid question. So, let's understand when and why to use the **IF** statement.

If you wish to perform an action based on a certain condition such as – assigning the value to a variable, executing a query and or wish to execute any T-SQL code, then you can do it only with **IF** statement. The **CASE** statement will not help. So, if you want to manipulate the data in a T-SQL code or query, then you should use But if you wish to define control over the flow of the code (running

a specific or set of T-SQL code based on a condition) then the **IF** statement is the only solution.

Let us quickly have a look into the syntax of the **IF** statement:

```
IF ()  
ELSE IF ()  
.  
.  
.  
ELSE IF ()  
ELSE
```

Let us understand the attributes of the **IF** statement:

IF statement can be simple or **IF ... ELSE** or **IF ...** or **IF ... ELSE IF ...**

IF and **ELSE IF** are used to define the condition. The resulting action of **ELSE** and **ELSE** has to be defined in a **BEGIN ... END** block.

IF and **ELSE IF** can be related to the **WHEN** keyword we discussed in the **CASE** statement.

ELSE is same in both **IF** and **CASE** statements.

In an **IF** statement, only one action would be triggered, based on the evaluation of the condition. Obviously, the action would be triggered only when the condition is true. But if there are multiple conditions that may be true, even then only one action would be triggered, for the first condition that would be true.

IF there are multiple conditions that needs to be evaluated, irrespective of each other, then simple **IF** statement should be used, with an **ELSE IF** or

Here is the example that should help you understand the application of **IF** statement in a much better way.

Hope you remember the and **PurchaseOrder** example that we discussed multiple times. You have two inputs – **ReportType** and Possible values for report type are – **1** for **2** for and **3** for **UniqueID** will be **CustomerID** for the customer, **ProductID** for product, and **OrderID** for purchase order. If either of the **ReportType** is incorrect, then show the message as **Incorrect Report**

The following is the T-SQL code that denotes the **IF** statement for the preceding scenario:

```
DECLARE @ReportType
```

```
DECLARE
```

```
SET
```

SET

```
IF =  
BEGIN  
SELECT * FROM  
END
```

```
ELSE IF =  
BEGIN  
SELECT * FROM  
END  
ELSE IF =  
BEGIN  
SELECT * FROM  
END  
ELSE  
BEGIN  
SELECT 'Incorrect Report'  
END
```

WHILE loop

Loop is a programmability feature that is used for iteration. If you need to iterate through values or dataset, then you would be needing a loop.

It is desired from the readers of this book that they should have basic knowledge of any programming language. The functioning of loop is common in all the programming languages so does in T-SQL. We'll not talk about the theory of loop in this book.

Let me take you through a practical scenario depicting the use case of the loop. Consider you want the running total of numbers starting from 1 to 10. The output should look as follows:

[illegible]

follows:
follows:
follows:
follows:

Table 9.1: *Running total example with formula*

We hope you can understand the scenario by looking at the preceding table. There are three columns – **Running** and **Number** column represents the number from 1 to **Running Total** shows the running total value. **Formula** column shows the formula to calculate the running total. Running total is simply the summation of all the values less than or equals to the current value.

There's one more interesting thing you would see in the formula column. *Can you see a triangle?* Yes, there it is. The values of the formula column form a triangle.

If you are to derive the running total and show the internal formula used, then *how can you do it?* One way to calculate the running total is through the windowing function, but if you are specifically told not to use the window function then you are left with only one option that is, loop.

SQL Server offers the **WHILE** command which is used to define the loop. When you see a **WHILE** keyword in a T-SQL

code, then you can easily identify that as a loop. Without talking much of theory and stories, let us see how we can derive the data as shown in the previous *table*

```
DECLARE
```

```
DECLARE @RunningTotal
```

```
DECLARE
```

```
DECLARE @MyTable TABLE
```

```
(  
    , [RunningTotal] INT  
    ,  
);
```

```
SET @Number =
```

```
SET @RunningTotal =
```

```
SET
```

```
WHILE <=
```

```
BEGIN
```

```
SET @RunningTotal = @RunningTotal +
```

```
SET @Formula = AS + '+' + @Formula
```

```
INSERT INTO @MyTable
```

```
(  
    [Number]  
    , [RunningTotal]  
    , [Formula]  
)
```

```
VALUES
```

```
(
```

```
@Number
, @RunningTotal
, -
);
SET @Number = @Number +
END
SELECT * FROM
```

When you'll run the preceding query, the output will match exactly with the data shown in *table*

Loop always has to be definite. It means you should be aware of the start and end of the loop counter. In the previous example discussed, 1 was the starting counter of the loop and 10 was the ending counter of the loop. A loop becoming indefinite or infinite never ends.

Here is the explanation of the previous program. It is advised to read the following explanation in conjunction with the preceding T-SQL code, and relate it with the individual line of the code:

Declaring a variable: We've declared three variables We've also declared a table variable **@MyTable** to store the intermediate data.

Assigning initial values to the variables: We've assigned the initial values to the variables.

Defining the loop: We defined the condition for the loop (**@Number <=** Loop will iterate until the condition is successful. The moment the value of **@Number** variable would be greater than loop will end. Since we've used the **@Number** variable in the loop condition, hence it's mandatory to assign it an initial value. That is why we've assigned it an initial value as So, the loop will start from **@Number = 1** and end at **@Number =** There will be total **10** iterations.

After the loop initialization, you could see two lines of code, each for variable values assignment to **@RunningTotal** and **@Formula** variables.

If you remember, we've assigned the initial value of the **@RunningTotal** variable as So once the iteration will start, it will have the value as the previous value of *@RunningTotal variable* + the current value of **@Number** variable.

Same goes to the **@Formula** variable. It will contain the formula used for the calculation of values of the **@RunningTotal** variable. **@Formula** is of type varchar and the + sign used in the formula of the value assignment of the **@Formula** variable denotes the concatenation. So, if there are two strings **A** and **B** and if + is specified in between them, then the resulting value will be This is called

Once we have the values assigned to **@RunningTotal** and **@Formula** variables, we insert them in the **@MyTable** table variable. You could see a formula in the **VALUES** section for the **Formula** column. The formula is *LEFT(@Formula,*

LEN(@Formula) - Let us understand what does this formula means. **LEN(@Formula)** will return the number of characters/lengths of the values of **@Formula** variable. The **LEFT** function is used to get the starting *LEN(@Formula) – 1* character from the **@Formula** variable. For example, if the value of **@Formula** is 1+2+ then it will return We've used this formula to exclude the last one character + from the **@Formula** variable value.

Now, it comes to the most important section of the loop. This line along with the loop condition drives the loop. If either of them is incorrect, the loop will fail or misbehave. Here, we have incremented the value of the **@Number** variable with So, with each iteration the value of **@Number** variable will increment by If we'll increment the loop counter, then the loop will become infinite and it will keep on iterating with the same loop counter, which is initial value

Finally, we came out of the loop and you could see the last line which returns the data inserted in the table variable.

If you could relate the steps we discussed in the explanation, then you would not find it difficult. It's just the specific keyword that has to be used for specific instruction.

Congratulations on your first T-SQL code! This is the first-ever T-SQL code we've discussed so far.

You can see the variable declaration, variable value assignments, and finally reading the variable values. So, this example shows the practical use of both variables and loop.

You can also see few new keywords such as **BEGIN ... END** block, use of semicolon use of **CAST** keyword, use of string built-in functions such as **LEFT** and table variables, and **SELECT** statement, all in one T-SQL code.

CAST is the new keyword we've talked about. Let us understand it.

CAST

CAST is a data conversion function used to convert the data from one data type to another. You need to specify the value (or the column name, if used with the table) and the data type to which you wish to convert the data. It is to be noted that **CAST** can be used to convert data to only compatible data types. If the destination data type is not compatible with the source data type, then it will throw an error. String data types are considered compatible destination data types for any kind of source data type, but not vice versa. You cannot convert **INT** to **DATE** to **VARCHAR** to and so on.

We'll talk more about data conversion in [Chapter 12, Data Conversion, Cross-Database, and Cross-Server Data](#)

In the previously discussed example, we've seen the working of the **WHILE** loop with a definite set of values. However, you can also iterate through a dataset such as table. The following example shows the implementation of the **WHILE** loop to iterate through all the rows of a table:

```

DECLARE
DECLARE @Start_CustomerID
DECLARE

DECLARE @Missing_Customer_IDs TABLE
(
CustomerID INT
);

SELECT @Start_CustomerID =
, @End_CustomerID =
FROM

SET @CustomerID =

WHILE <=
BEGIN
IF NOT EXISTS 1 FROM Customer WHERE CustomerID =
BEGIN
INSERT INTO @Missing_Customer_IDs VALUES
SET @CustomerID = @CustomerID +
SELECT * FROM

```

Here is the explanation of the preceding program:

We would not discuss the preceding T-SQL code or program in details, the way we discussed earlier. However, we'll talk about the gist of the overall T-SQL code:

Taking the minimum and maximum **CustomerID** from the **Customer** table and assigning it to the variables.

Loop is iterating from start **CustomerID** until end with **CustomerID** being incremented by 1 with each iteration.

NOT EXISTS is the T-SQL function that accepts the **SELECT** query and returns the bit type – 1 and 1 can be treated as **True** and 0 as **False**. If the **SELECT** query supplied to the function will return a value, then the output would be 1 else

If there is no entry against the **CustomerID** in any iteration, then we are inserting that **CustomerID** in the table variable.

Finally, the data written in the table variable would be returned with the help of the **SELECT** query.

This program will return the missing **CustomerIDs** between **Start** and **End**. Let say, if the **Customer** table has three records starting from such as – The **Start CustomerID** is 1 and **End CustomerID** is 10. The missing **CustomerID** between **Start** and **End CustomerIDs** are 2, 3, 4, 5, 6, 7, 8, 9 and This is what our preceding T-SQL code will return.

WHILE loop can also be nested. It means there could be another **WHILE** loop inside a **WHILE** loop.

BREAK

The **BREAK** keyword is used to exit from the loop, even before it reaches the end of the loop counter. For example, if the loop is written to iterate from 1 to 10, you can write a condition to exit from the loop using the **BREAK** keyword, even before it reaches 10. In the absence of the **BREAK** keyword (with a specific condition), there would be a total of 10 iterations. **BREAK** is an optional keyword in the **WHILE** loop, which you can use if you wish to, but there is no compulsion.

The **BREAK** keyword technically stops the execution of the loop further, and any lines of code written after the **BREAK** keyword will not be executed.

The following is T-SQL code to break from the loop, if the loop counter is 4. Running this code will print 4 rows for counters 1 to 4.

```
DECLARE @Counter
```

```
SET @Counter =
```

```
WHILE @Counter <=
```

```
BEGIN
```

PRINT

IF =

BEGIN

END

SET @Counter = @Counter +

END

CONTINUE

The **CONTINUE** keyword is used to skip the code written after the **CONTINUE** keyword and start the loop again. In the programming world, there is a terminology called Control refers to the execution of any line of code. It could be also referred to as a block of code.

Suppose we've a **WHILE** loop shown as follows. It has eight lines of code. The code execution starts line by line for each line of code. First of all, the first line will be executed. In that case, it is said that control is at *Line* Subsequently, the second line of code will be executed, and it is said that control is at *Line* In the following scenario, once the control will be at *Line 5* which is the **CONTINUE** statement, all the further lines will be skipped, and the control will be brought to the *Line* which is the starting point of the loop. We hope this explanation would have helped you to understand the functioning and purpose of the **CONTINUE** keyword:

```
Line 1:  WHILE ()  
Line 2:   BEGIN  
Line 3:   code 1>  
Line 4:   code 1>  
Line 5:   CONTINUE  
Line 6:   code 1>  
Line 7:   code 1>
```


Line 8: END

The following is a sample T-SQL code featuring the **CONTINUE** keyword. Running this code will print ten rows for each counter. It is possible because the **CONTINUE** command has been specified before the **BREAK** command. Similarly, if the code written to increment the loop counter would not have been placed before the **CONTINUE** command, then the loop would become infinite, and would run continuously for the same value (that is

```
DECLARE @Counter
```

```
SET @Counter =
```

```
WHILE <=
```

```
BEGIN
```

```
PRINT
```

```
SET @Counter = @Counter +
```

```
IF =
```

```
BEGIN
```

```
END
```

```
END
```

Conclusion

Every book is made up of alphabets only, arranged in the form of words, sentences, paragraphs, topics, and chapters. Variables, loops, and control flow statements are the basics of any programming language, including T-SQL. They are alphabets of the T-SQL, which you can arrange to build a meaningful, and also a complex program.

All the programs, be it small, or big, simple, or complex. All of them are written with the help of the same syntax and keywords. It's just the purpose and logic which is different.

Programming is all about the logic, and the ability to write the precise instructions in the form code. Hence, it's very important to think, think, and think, unless you are sure of *What* and *What means what you want to achieve with the program, and how means how would you accomplish it? How* can be understood as the logic of the program? *What* can be related to the requirement and purpose of the program?

In this chapter, you've learnt to declare a variable, assign values to the variables, and read the values from the variables. You've also learnt to define the **WHILE** loop, **IF** statements. You've also learnt about few other T-SQL programmability features such as **BEGIN ...** and so on. The features you've learnt in this chapter are the basics of T-SQL

programming. You can use it to develop your own T-SQL program.

In the next chapter, you'll learnt about the temporary tables, and **MERGE** statements.

Points to remember

CAST can be used to convert data to only compatible data types.

In T-SQL the variable name always starts with Anything you see which starts with **@** is a variable.

The variables always hold only a single value, unless it is of type table called table variable.

It is important to specify the conditions in proper sequence in both **IF** and **CASE** statements.

There could be nested case statements too. You can add another case statement in the action

There could be nested **IF** statements too. That means another **IF** statement can be specified in the action of an **IF** statement.

Loop always has to be definite. It means you should be aware of the start and end of the loop counter.

A loop becoming indefinite or infinite never ends.

Multiple choice questions

You have two different variables @A and @B. You want to assign them the value based on a condition. The condition is such that at a point only one variable can be assigned the value. Which of the following options would you choose?

CASE statement

IF statement

Can we use the CASE statement in an UPDATE statement?

Yes

No

Can we use the IF statement in a CASE statement?

Yes

No

Which of the following statement is true about WHILE loop?

It is defined with the help of keywords **WHILE** loop.

The action of the loop can be defined without **BEGIN ...**

Loop condition cannot be a query.

It should have a loop condition, and loop counter should be incremented.

Answers

B

A

B

D

Questions

When to use a **WHILE** loop?

When does a loop become infinite?

Why it is important to increment the loop counter?

What are the different types of **IF** statements?

Can an **IF** statement be without an

What is the difference between **CASE** statements?

What is the purpose of **WHEN** keyword of the **CASE** statement?

What is the purpose of the **BREAK** statement?

Key terms

WHILE is used to define the loop.

CASE statement is used to define the condition and resulting action to manipulate the data.

IF statement is used to define the condition and resulting action to execute a code.

BREAK is used to exit from the loop. The control is returned to the immediate next line of code after the loop.

WAITFOR is used to pause the execution for the specified time or duration.

GOTO statement is used to traverse through the bookmarks (programmatically). These bookmarks need to be explicitly defined in the code.

CHAPTER 10

Temporary Tables, CTE, and MERGE Statement

Most of the times, you would want an intermediate store for the query results. You may want to use this intermediate result in another query. Temporary tables and **Common Table Expression (CTE)** both serve the purpose of an intermediate data layer. Temporary table physically (but temporary) stores the data, whereas CTE is evaluated at the run time. CTE does not store the data physically. Using the **MERGE** command, multiple DML statements can be executed together in a single statement.

Temporary tables, CTE, and **MERGE** statements are useful when writing complex T-SQL code. In this chapter, we'll understand and learn to implement them.

Structure

In this chapter, we will cover the following topics:

Temporary tables

Common Table Expression

MERGE command

Objective

The features discussed in this chapter become very useful to know when you enter the arena of the real programming for the business systems both for OLTP and OLAP workloads.

Temporary tables are used to avoid the repetition of the same code again and again. It's not just the code that is repeated. Each statement in your T-SQL code is executed individually and has associated costs and overheads. Cost includes CPU (core) requirement, memory requirements, IO requirement, network speed requirements, and so on. Although, the cost does not mean money in the programming world, but it may be related to money too because additional hardware requires additional money.

More resource overheads mean more time required for the execution called **execution**. So, if you have a query that takes 10 min to run, and the same query has been used multiple times in your T-SQL code, being executed together. You can use temporary tables to store the intermediate result and refer to this temporary table instead of the actual query resulting in the same data.

If you have a or **DELETE** query, which is having multiple lines of code, involving multiple joins, subqueries, filters, aggregations, and so on. From both readability and processing

perspectives, the code must be modular (small pieces of code, broken logically). CTE can be used to write modular queries.

MERGE command too can be used to modularize the **Data Manipulation Language** statements and execute the relevant DML statements (such as and together.

Temporary tables

This is a very interesting feature of SQL Server. You can do almost everything with temporary tables, which you can do with physical tables. You can define constraints (all kinds of constraints), you can create indexes, statistics, and identity columns on temporary tables. You can do insert, update, delete, and select on a temporary table. You can use them in joining, apply clause, subqueries, where clause, and so on.

You can create, alter, and drop the temporary tables with the same syntax as used for physical tables. The only thing you cannot do with a temporary table is storing the data permanently. *That makes sense too! Why you would need to store the data permanently with temporary tables?*

As the name suggests, these tables store the temporary results in a limited scope. The scope is of two types: **local** and The scope is nothing but the session. *Have you noticed multiple query windows on SSMS?* Each of those query windows is a unique session having a unique session ID called If you would like to check the SPID of any query window in SSMS then you can use **@@SPID** system keyword, as can be seen in the following screenshot. You would see two consecutive screenshots.

The following screenshot is for the first query window with SPID as

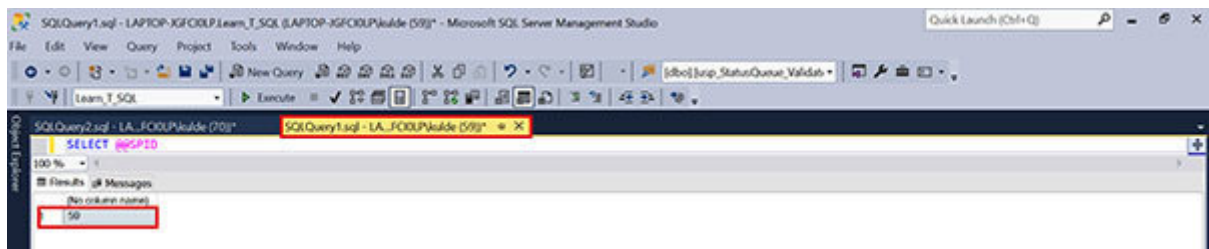


Figure 10.1: SPID 50

The following screenshot is for the second query window with SPID as

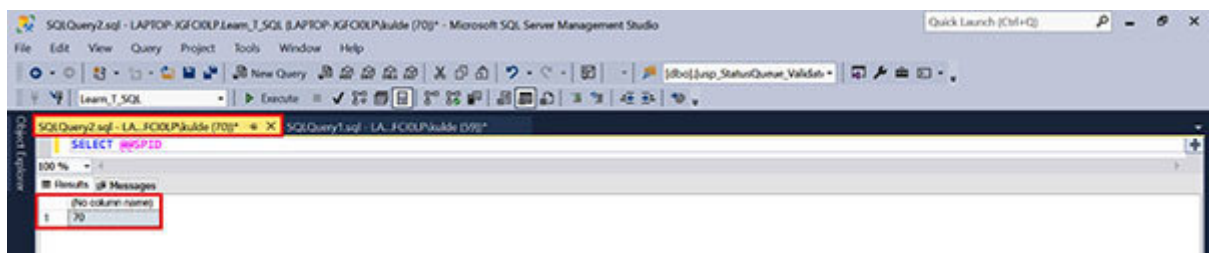


Figure 10.2: SPID 70

These are the results from SSMS on my system. When you'll run it on your system, you may get different SPID's.

As we discussed, SPID represents a unique session. All the queries or T-SQL code executed on a query window will always have the same SPID. We've discussed the SPID in the context of SSMS. *What if the queries/T-SQL code is being executed from the application code?*

The same fundamentals apply there as well. You need to open a SQL Server connection and then execute the T-SQL code. The moment you'll open a connection, a unique session will be generated and it will be assigned to the connection. All the queries executed in this session will have the same SPID. You can even run **SELECT @@SPID** from the application code and test if two different connections/sessions have different SPID's. Although, it is beyond the purview of this book.

I hope you've understood the SPID. Now, let us understand the scope. As we discussed scopes are of two type – **local** and

Local scope is specific to a session/SPID.

Global scope having access to all sessions/SPID.

Temporary tables are also of two types – **local temporary tables** and **global temporary**

Local temporary tables are specific to a specific session. It can't be accessed in another session. They are defined with **#** (hash) sign before the table name.

Global temporary tables have global access. They can be accessed in any session active on the SQL Server. They are defined with **##** (double hash) sign before the table name.

(hash) sign is the identification of the temporary table. You won't find any other distinguishing factor between temporary and physical tables. If you'll see **#** or **##** specified in a table then consider it as a temporary table.

One of the most important things to note in the case of temporary tables - it is created in **tempdb** system database. Whereas the physical tables are created in the respective user database. If you need to create or access a temporary table then you may or may not use the three-part naming. Although temporary tables are created in a separate **tempdb** database, but you need not to specify the database name, when creating or accessing it. When you create or access a table with **#** prefixed, it is treated as the table belongs to the

tempdb is re-created every time SQL Server is started. All the temporary tables including local and global would be cleaned (dropped) automatically, upon a SQL Server restart.

Since temporary tables too have the same syntax as physical tables, hence we'll not talk about much of the syntax, and so on. Instead, we'll see two examples each for local and global temporary tables. Before we create the temporary tables, let us expand and see the **tempdb** contents.

The contents of **tempdb** database on my SQL Server instance can be seen as shown in the following screenshot. If you'll

closely look at the screenshot, then you won't see anything under the **Temporary Tables** folder:

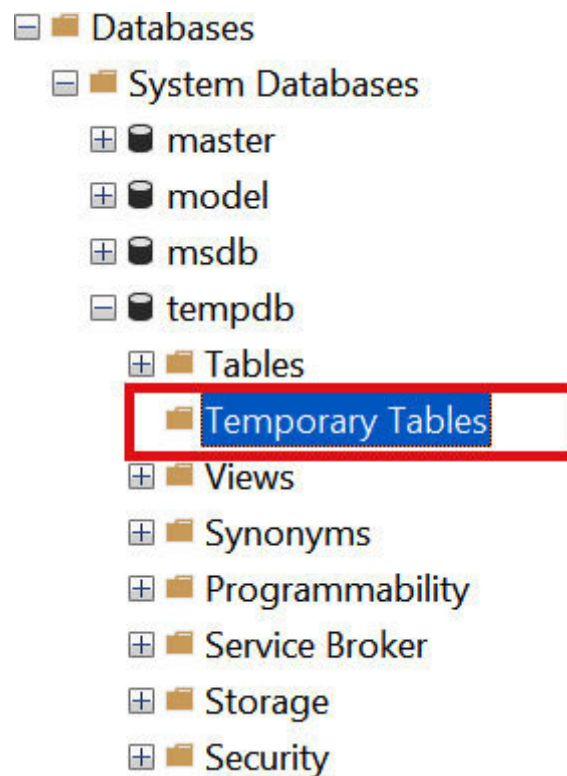


Figure 10.3: Folder in Object Explorer for temporary tables

Now, let us create the temporary tables. Following is an example of a local temporary table:

```
CREATE TABLE #LocalScopedTemporaryTableExample  
(  
    NOT NULL  
  
    , [Code]    NOT NULL  
    ,  
);
```

As you can see in the following screenshot, the local temporary table with the preceding T-SQL code is created in session with SPID. You may get a different SPID when you'll run it in your system:

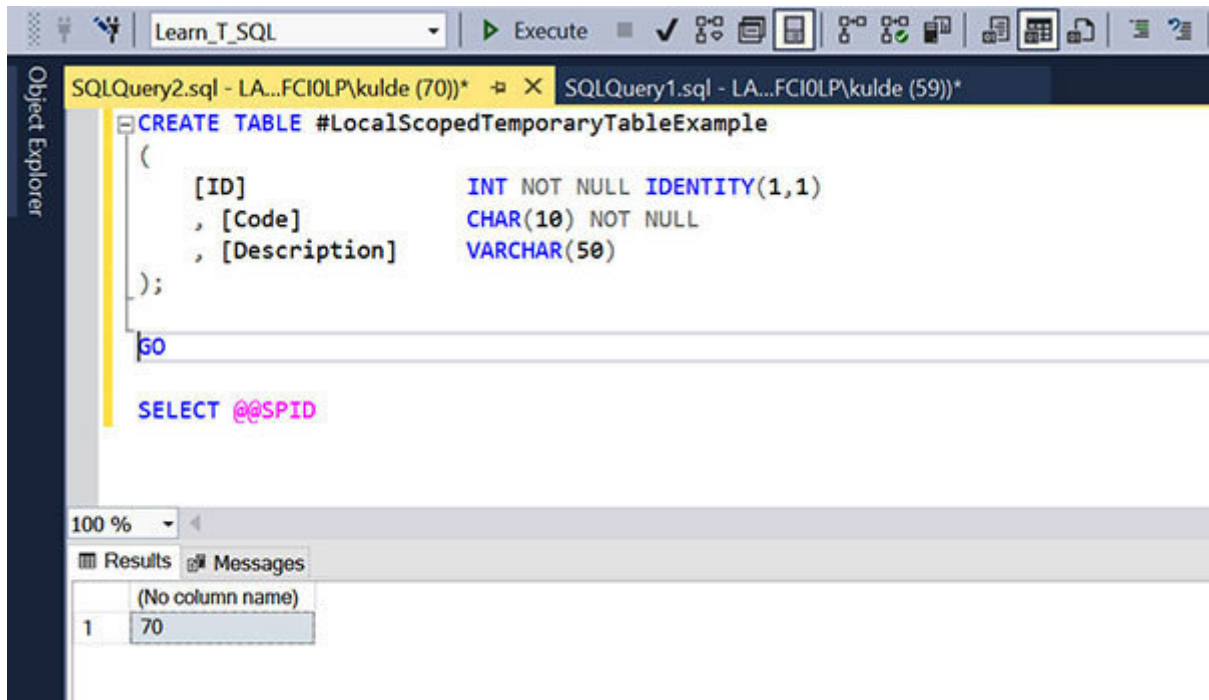


Figure 10.4: Local temporary table created in SPID 70

Following is an example of a global temporary table:

```
CREATE TABLE ##GlobalScopedTemporaryTableExample
(
    NOT NULL
, [Code]    NOT NULL
,
);
```

As you can see in the following screenshot, the global temporary table with the preceding T-SQL code is created in

session with SPID You may get a different SPID when you'll run it in your system:

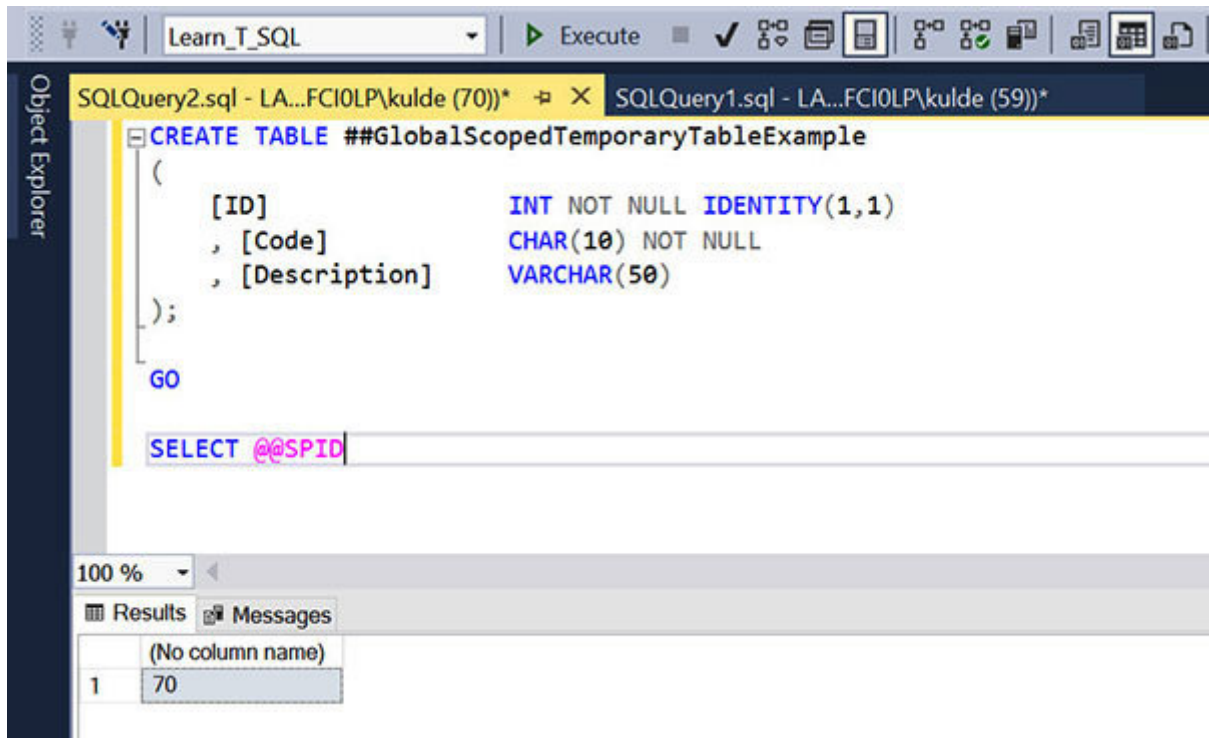


Figure 10.5: Global temporary table created in SPID 70

Let us refresh the **Temporary Tables** folder of **tempdb** and see what it holds. You can see there are two temporary tables reflecting under the folder now, each for the tables (local and global temporary tables) we created.

If you've noticed then the local temporary table has some suffix after the table name. You would be wondering *why*? Because local temporary table has local scope of a session/SPID. So, you can create multiple tables with the same name in different sessions. The table name should be unique in a session.

For example, we can create another local temporary table with the same name in another session (with a different SPID). You can't create two local temporary tables with the same name in the same session. To facilitate this, SQL Server internally suffix some hexadecimal value to make the table name unique. That is why, you can see suffix in case of local temporary table, as can be seen in the following screenshot:



Figure 10.6: Temporary tables in the tempdb database

Now let us test if we can access these tables in different sessions. We'll run a simple **SELECT** statement with both of these local and global temporary tables, individually, on a different session having SPID. Please note the tables we created on the session with SPID.

We'll first run the **SELECT** statement on the global temporary table. As you can see in the following screenshot, it's accessible even in session with SPID.

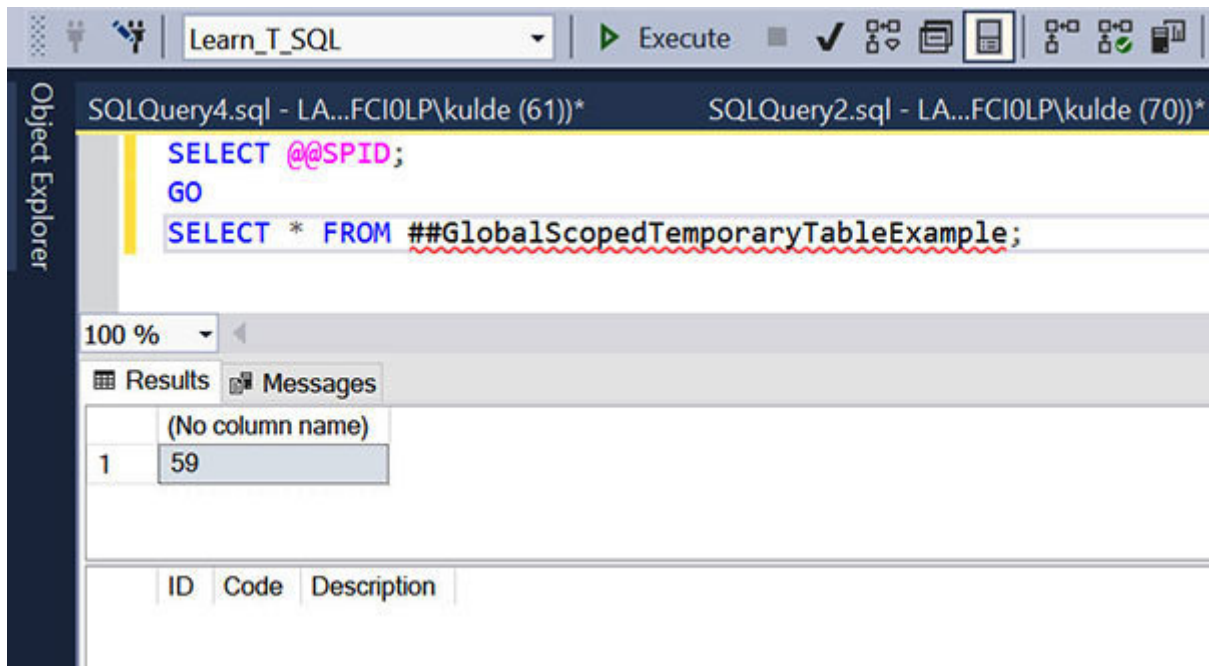


Figure 10.7: Output of querying global temporary table from SPID 59

We'll now run the **SELECT** statement on the local temporary table. As you can see in the following screenshot, it's not accessible in the session with SPID. We can see an error thrown upon running the **SELECT** statement, which clearly states the object is not found:

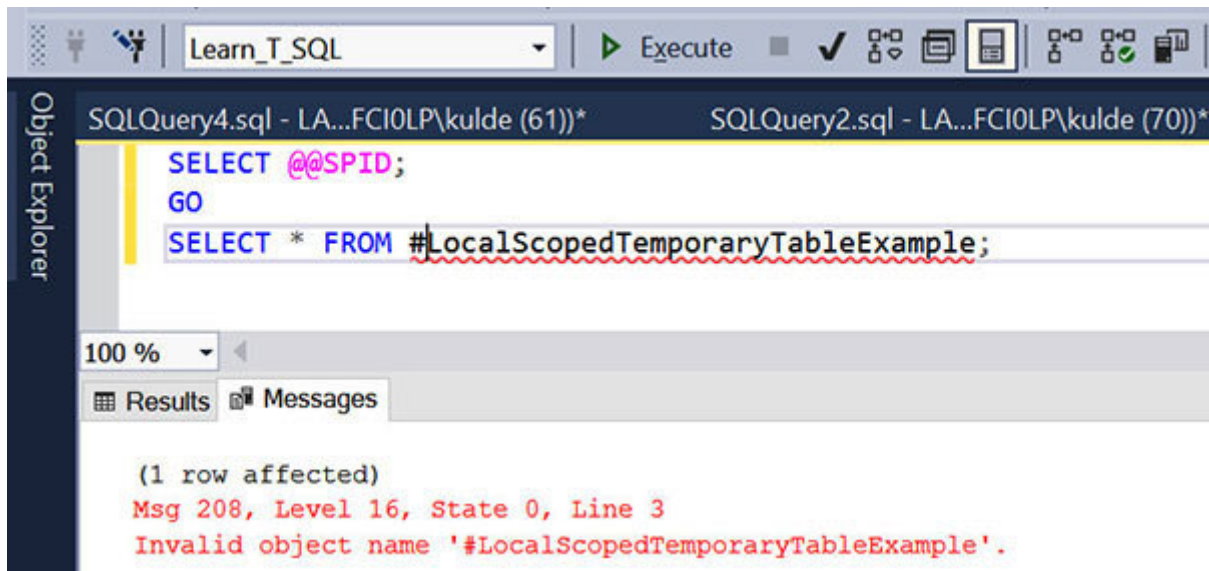


Figure 10.8: Output of querying local temporary table from SPID

59

Since we are not able to see the SPID in the preceding screenshot due to an error, we'll make a simple change and replace `SELECT @@SPID` with `PRINT @@SPID`, and run the query again. Now you can see the SPID with the error message clearly, as can be seen in the following screenshot:

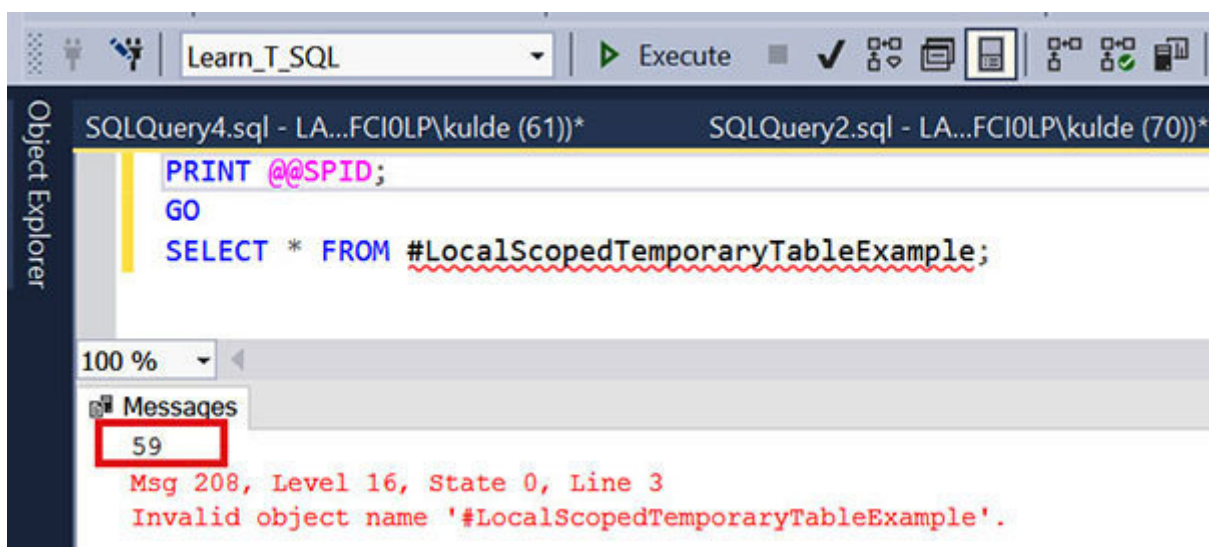


Figure 10.9: Printing the SPID

Common Table Expression (CTE)

CTE specifies a temporary result-set, with a name assigned to it. It is a view that can only return the result-set. You cannot or **DELETE** records in a CTE. CTE can be used in the and **MERGE** statements just like a normal table. But it has limited execution scope. CTE can be referred only in the immediate and **MERGE** statements after the CTE definition. You cannot refer it anywhere else. This is the most important factor that should be noted when planning to use it.

Apart from the purpose of CTE we've discussed, it also helps to make the code modular. A modular code has better readability, is easy to debug, and also performs better. Instead of writing a big piece of program/code, it can be broken into a smaller, but the logical set of instructions (of the program). In the context of T-SQL, a large query can be divided into a smaller, but the logical set of queries, which is easy to understand and debug.

It is not always possible that when you'll break a large query into multiple smaller queries, it will perform better.

Performance depends on multiple factors. Sometimes, breaking a large query into multiple smaller queries will not help, and would poorly perform, as compared to the original large query. So don't just break the code, just for the sake of modularity. Consider other relevant aspects too. It even

applies to CTE. Don't just always use the CTE, unless you really need it. It is purely your decision when to and when not to use the CTE, considering the requirement, and scenario.

Following is the syntax of defining and using the CTE:

```
; WITH of the CTE / temporary result-set>  
(  
Column list of the result-set separated by comma (,). This is  
optional, and you may skip it. But if you are defining it then  
the resulting SELECT query should match the column  
definition.  
)  
AS  
(  
SELECT query, returning the exact set of columns as defined  
in the column list. You can also refer to the previous result  
sets of the CTE in the query specified here.  
)
```

and **MERGE** statements referring to the CTE. You can use the CTE here, but not beyond this statement. This is the execution scope of the CTE.

The first result-set in a CTE should start with **WITH** keyword. The line of code previous to CTE (or **WITH** keyword) should have a semicolon (;) to specify the end of the previous statement. If not then you need to specify the semicolon (;) before **WITH** keyword. For example, ‘;

There could be multiple result-sets in a CTE. The subsequent result sets in a CTE should be separated by a comma For example:

```
; WITH cte_test1  
AS  
(  
SELECT 1 AS ID, 'Test 1' AS Name  
)  
, cte_test2
```

```
(  
SELECT 2 AS ID, 'Test 2' AS Name  
)
```

```
SELECT ID, Name FROM cte_test1  
UNION  
SELECT ID, Name FROM cte_test2
```

UNION is a set operator which combines the results (or rows) to two **SELECT** statements, but the final combined output would be the distinct rows of the results of both the **SELECT** statements. For example, if the **SELECT** query is **SELECT 1 UNION SELECT 1** then the output would be just 1.

There are other set operators such as. All the set operators are used between two **SELECT** statements.

UNION ALL: Combines the results (or rows) of two **SELECT** statements, and returns all the rows from both the **SELECT** statements. For example, **SELECT 1 UNION ALL SELECT 1** would return two rows with values 1 and 2.

EXCEPT: Returns the rows from the top **SELECT** statement, only if the row doesn't exist in the bottom **SELECT** statement. For example, the following query would return two rows with values 1 and 2:

```
; WITH cte_Test
AS
(
SELECT 1 AS ID
UNION
SELECT 2
)
SELECT ID FROM cte_Test
```

```
EXCEPT
SELECT 3
```

INTERSECT: Returns only the matching rows from both (top and bottom) **SELECT** statements. It can be considered as the intersection of both the result sets. For example, the following query would not return any row:

```
; WITH cte_Test
AS
(
SELECT 1 AS ID
```

```
UNION
SELECT 2
)
```

```
SELECT ID FROM cte_Test
INTERSECT
SELECT 3
```

But the following query would return one row with value 1:

```
; WITH cte_Test
AS
(
SELECT 1 AS ID
UNION
SELECT 2
)
SELECT ID FROM cte_Test
INTERSECT
SELECT 1
```

You can read more about the set operators at the official Microsoft official documentation using the following URLs:

<https://docs.microsoft.com/en-us/sql/t-sql/language-elements/set-operators-union-transact-sql?view=sql-server-ver15>

<https://docs.microsoft.com/en-us/sql/t-sql/language-elements/set-operators-except-and-intersect-transact-sql?view=sql-server-ver15>

Following is the T-SQL example of CTE:

```
; WITH cte_Product
(
[ProductID]
, [ProductName]
, [First_ProductID]
)
AS
(
SELECT [ProductID]
, [ProductName]
, BY ProductID AS [First_ProductID]
FROM Product
)

SELECT [ProductID]
, [ProductName]
FROM cte_Product
WHERE [First_ProductID] = 1
```

The preceding query can be alternatively written as follow:

```
; WITH cte_Product
AS
(
SELECT [ProductID]
, [ProductName]
```

```
, BY ProductID AS [First_ProductID]
FROM Product
)
```

```
SELECT [ProductID]
, [ProductName]
FROM cte_Product
WHERE [First_ProductID] = 1
```

Both of these T-SQL codes will do the same things, and also return the same result as can be seen in the following screenshot. Both the preceding T-SQL code will return the first **Product** by sorting the result-set by **ProductID** in the ascending order. The ranking is done with the help of **ROW_NUMBER()** function:

	ProductID	ProductName
1	1	Product 1

Figure 10.10: Output of the query with CTE

There is another variant of CTE called **Recursive** Recursive CTE is used for recursion (iteration), similar to loop. But here in Recursive CTE, you need to define the recursion within the CTE using the **SELECT** statement only. Whereas, in loop, you've much more flexibility. Each of them has a specific use case and should be used accordingly.

You can read more about the CTE (including Recursive CTE) at the following URL of the Microsoft official documentation:

<https://docs.microsoft.com/en-us/sql/t-sql/queries/with-common-table-expression-transact-sql?view=sql-server-ver15>

MERGE command

*Have you ever thought of performing the INSERT, UPDATE, and DELETE on a table, using a single statement? If yes, then **MERGE** command is the only solution in T-SQL. Except for **MERGE** command you cannot perform all these DML operations, together in a single statement. An ideal approach makes you write an individual statement for each of these DML operations.*

MERGE command involves source and target. One of these sources and target can be a query and other should be a table (including a table variable, or temporary table). Because you need to have a table to perform DML operation. You cannot perform DML operations on a view (result-set of a query).

It is quite an advanced feature offered by SQL Server. You can get comprehensive knowledge about **MERGE** command at the following URL of Microsoft official documentation:

<https://docs.microsoft.com/en-us/sql/t-sql/statements/merge-transact-sql?view=sql-server-ver15>

Conclusion

Temporary tables serve the purpose of intermediate store, the very same way, table variables do. Both these table variables and temporary tables physically (but temporarily) stores the data.

CTE also called **Common Table Expression** is also an intermediate layer, but not the store. CTE does not stores the data physically. The queries of the CTE are evaluated at the run time. You can use CTE to break a complex query into smaller queries in the form of CTE.

MERGE command can execute multiple DML (such as and statements together. Although, only logical and relevant DML statements should be combined with the help of **MERGE** command. Do not complicate the T-SQL code just for the sake of using the feature. If you do not see any relevance then it's better not to use a feature, and follow an ideal approach.

Any program including a T-SQL code should be optimized. *What do you understand from an optimized code?* An optimized program/code has the following three characteristics:

Optimal resource utilization: The resource here means the physical hardware resources such as CPU, memory, IO,

network, and so on.

Less execution time: It means the code/program gets executed quickly (or takes lesser time to execute).

Modularity: It means a large code/program is broken into a smaller, yet logical pieces, for better readability and processing. Compilation too becomes efficient with modular code.

Temporary tables, CTE, and **MERGE** commands can indeed help write an optimal code, if properly thought. On the other side, a lot of people also abuse these features and end up writing a poor-performing code. So, it's very important to evaluate when and when not to use them.

In the next chapter, you will learn error handling and transaction management.

Points to remember

Temporary tables are created in **tempdb** system database.

Temporary tables of two types – local and global temporary tables.

You can create multiple local temporary tables with the same name, but in the different sessions. Each session should have unique local temporary tables.

You cannot create multiple global temporary tables with the same name.

CTE can be referred only in the immediate and **MERGE** statements after the CTE definition. You cannot refer it anywhere else. This is the most important factor that should be noted, when planning to use it.

The first result-set in a CTE should start with keyword. The line of code previous to CTE (or **WITH** keyword) should have semicolon to specify the end of previous statement. If not then you need to specify the semicolon before **WITH** keyword. For example,

Multiple choice questions

Is it mandatory to define the temporary table name with three-part naming?

Yes

No

CTE has column list A, B, and C. But the SELECT query used in the CTE return only two columns A and B. Will the CTE work?

Yes

No

CTE has column list A, B, and C. The SELECT query used in the CTE also returns three columns, but with different names – Col A, Col B, and Col C. Will the CTE work?

Yes

No

CTE has column list A, B, and C. A is of type varchar, B is of type DATE, and C is of type INT. The SELECT query used in the CTE also returns three columns, but the sequence of a column is – A, C, and B. Will the CTE work?

Yes

No

Answers

B

B

A

B

Questions

What is the difference between a table variables and temporary tables?

What is the difference between local and global temporary tables?

Can we drop a temporary table using **DROP DDL** command?

Can we alter a temporary table using **ALTER DDL** command?

When not to use CTE?

Can we define a CTE without specifying the semicolon prior to **WITH** clause? If yes, then when?

Can we use a CTE in a **MERGE** command?

What is the execution scope of a CTE?

What is the difference between CTE and temporary tables?

What is the folder name in **tempdb** database which shows the list of temporary tables in the **tempdb** database?

Key terms

Local temporary tables are specific to a specific session. It can't be accessed in another session. They are defined with **#** (hash) sign before the table name.

Global temporary tables have global access. They can be accessed in any session active on the SQL Server. They are defined with **##** (double hash) sign before the table name.

CTE full form is Common Table Expression.

UNION is a set operator which combines the results (or rows) to two **SELECT** statements, but the final combined output would be the distinct rows of the results of both the **SELECT** statements.

UNION ALL combines the results (or rows) of two **SELECT** statements, and returns all the rows from both the **SELECT** statements.

EXCEPT returns the rows from the top **SELECT** statement, only if the row doesn't exist in the bottom **SELECT** statement.

INTERSECT returns only the matching rows from both (top and bottom) **SELECT** statements. It can be considered as the

intersection of both the result-sets.

Error Handling and Transaction Management

None of the programs, code, or software products are error (or, bug) free. Instead, errors (or, bugs) are free with it. Don't take me wrong, but that's the reality. That is why periodic product updates are being released. Be it operating systems such as Windows, Android, and so on, or RDBMS such as SQL Server, Oracle, and so on. All of them fix the errors in their products and release them for the end customer as a part of product updates. T-SQL is also a programming language, and errors are a by-product hereto. We'll learn to handle the errors in T-SQL in this chapter.

In [Chapter 1, Getting](#) we talked about transactions and ACID properties of relational databases. We also discussed that **A** of an ACID refers to Atomicity property ensures the entire transaction should either succeed or fail, but in totality. We'll learn to manage the transactions that include – opening a transaction, committing, or failing it.

Structure

In this chapter, we will cover the following topics:

Error handling

Transaction management

Objective

Errors are inevitable, irrespective of how much due care is being taken while writing a program. Even if your program is flawless, errors are possible. For example, a program processing file data may fail if the data are not in the proper format or data type. You don't have any control over the external data. Another example is a network failure, causing a connection to break. There could be many more similar scenarios. Error handling does not really mean to avoid the error, but to handle the error and take the alternate action (or, plan B).

Similarly, transaction management includes defining a transaction, committing, and reversing (or, rollback) it. For example, there are set of DML statements to be executed in sequence. They should execute in totality. If any of these DML statements fails, then the entire transaction should fail. This is possible through transaction management which we'll learn. You may have to rollback a transaction based on various cases, including in case of error. Hence errors and transactions are correlated. We'll also learn to handle the transaction in case of errors.

Error handling

We've already discussed about error handling. Let us understand how to implement it. As part of the error handling feature, SQL Server offers few special keywords, system functions, and global variables. They be used together to handle the errors in a T-SQL program and are given as follows:

TRY ... CATCH

Retrieving error information

ERROR_NUMBER()

ERROR_SEVERITY()

ERROR_STATE()

ERROR_PROCEDURE()

ERROR_LINE()

ERROR_MESSAGE()

@@ERROR

We'll discuss each of these in detail.

TRY ... CATCH

TRY ... CATCH is composed of two different blocks – **TRY** and **CATCH**. Each of these blocks is defined with the help of **BEGIN ... END**. The **TRY** block is defined by **BEGIN TRY ... END**. Similarly, **CATCH** is defined by **BEGIN CATCH ... END**.

A group of T-SQL statements can be enclosed in **TRY** block. **TRY** as the name suggested tries/tests for the error in the given T-SQL statement. If an error occurs in the **TRY** block, control is passed to the **CATCH** block.

Another group of T-SQL statements can be enclosed in **CATCH** block too. **CATCH** as the name suggests catches the error. Remember, **TRY** tests for the error, and **CATCH** define what to do if there is an error. Following is the syntax of **TRY ... CATCH**.

BEGIN TRY

Set of T-SQL statements. You can have N-number of T-SQL statements.

END TRY

BEGIN CATCH

Set of T-SQL statements. You can have N-number of T-SQL statements.

END CATCH

Some of the following few important points to be noted:

A **TRY** block catches all the errors that have severity higher than given it does not close the connection with the database.

A **TRY** block should have an immediate **CATCH** block. You cannot have a **TRY** block without an immediate **CATCH** block, and vice versa.

If there is no error in the T-SQL statements enclosed in a **TRY** block, then control will bypass the **CATCH** block, and the control will pass to the T-SQL statement immediately after the **END CATCH** statement.

If there is an error in the T-SQL statements enclosed in a **TRY** block, then control passes to the first T-SQL statement immediately after the **BEGIN CATCH** statement.

If **END CATCH** statement is the last statement in a T-SQL code, then the T-SQL program execution will end in both the cases – error and no error. In case of error, the T-SQL statements specified within a **CATCH** block will be executed before the execution ends. Whereas in case of no error, **CATCH** block will be bypassed, and execution will end.

A code embedded in **TRY ... CATCH** block will not return an error, unless specifically returned using **RAISEERROR** or **THROW** commands. Errors are generally returned by SQL

Server, but you can also raise the custom errors using these commands.

Read more on **RAISERROR** and **THROW** commands at the following URL of Microsoft official documentation:

<https://docs.microsoft.com/en-us/sql/t-sql/language-elements/raiserror-transact-sql?view=sql-server-ver15>

<https://docs.microsoft.com/en-us/sql/t-sql/language-elements/throw-transact-sql?view=sql-server-ver15>

TRY ... CATCH blocks can be nested. Both **TRY** and **CATCH** blocks can contain nested **TRY ... CATCH** blocks.

You cannot use **GOTO** statement to jump to another **TRY** or **CATCH** block. But you can use **GOTO** statement to jump to a label inside the same **TRY** or **CATCH** block.

You can use **TRY ... CATCH** block in stored procedure and trigger. But you cannot use it in a user-defined function and view. We'll learn about it in [Chapter 13](#).

Retrieving error information

TRY block tries the errors in the embedded T-SQL statement, **CATCH** block catches the error returned by **TRY** block. But *how to retrieve the information about the error?* In this topic, we'll talk about various system functions that can be used to retrieve various information about the error.

Before we talk about these system functions, it is important to know, these functions can be used only in a **CATCH** block. Although you can use it anywhere in a T-SQL code, but they it won't return a value. The **NULL** value would be returned if they are used outside the **CATCH** block. But when they'll be used in a **CATCH** block, you'll get the actual information of the errors, based on the purpose of the function. Each of these functions has a specific purpose discussed as follows. By looking at the name you can relate the purpose of the function, as the names are self-explanatory.

Almost every error has the following properties. Except few such as error message, line, and procedure, you would not find if the other properties make sense. I've not personally used them so far. But they too are important. All these properties are related. Even the Microsoft SQL Server Product team would be using them to debug the errors in the source code of the SQL Server:

Error number: Represents the unique number of the error. Although in case of custom raised error, you have the liberty to assign an error message. All the system error messages are stored in a system table **ERROR_NUMBER()** function returns the error number.

Error severity: Represents the severity of the error. **ERROR_SEVERITY()** function returns the error severity.

Error state: Represents the state number of the error. **ERROR_STATE()** function returns the error state.

Error procedure: Represents the name of the procedure that has returned the error. If the T-SQL code is executed which is not embedded in a procedure then **NULL** would be returned. **ERROR_PROCEDURE()** function returns the error procedure.

Error line: Represents the line number at which the error was returned. **ERROR_LINE()** function returns the error line.

Error message: Represents the exact message of the error. This is something which we would be interested to know. Error message along with line number comes handy while tracing an error. **ERROR_MESSAGE()** function returns the error message.

@@ERROR

@@ERROR is a global variable that returns the values of the type integer. When you see anything in T-SQL with **@@** as a prefix then it is a global variable. There are many global variables offered by SQL Server, for specific purposes. For example, we've seen in [Chapter 10, Temporary Tables, CTE, and MERGE](#) **@@SPID** returns the SPID/session ID. Similarly, there are other global variables too, which we'll talk about in upcoming chapters.

Coming back to It is used to check if the last T-SQL statement executed returned any error. If yes, then **@@ERROR** will return the error number associated with the error. It generally returns the **message_id** column value from the **sys.messages** table, if the error returned is available in this table. If the last T-SQL statement executed did not returned any error then **@@ERROR** will return

@@ERROR is cleared and reset on each statement executed. Hence check it immediately after the statement to be checked for the error.

When a T-SQL code doesn't have the **TRY ... CATCH** block then even after there is an error, the execution will continue. It means all the statements following the statement thrown error would be executed. For example, if there are 10

statements in a T-SQL program. If the error was thrown at the third line the execution will continue, and the remaining 7 lines will also be executed in the sequence they are written.

But, if a T-SQL code is embedded in a **TRY ... CATCH** block then the control will be passed to the **CATCH** block after the error is raised, without executing the further statements. For example, if there are ten statements in a T-SQL program embedded in a **TRY ... CATCH** block. If the error was thrown at the third line, remaining seven lines will not be executed, and the control will be passed to the **CATCH** block.

@@ERROR can be used even without a **TRY ... CATCH** block. However, it is advised to use it in conjunction with the **TRY ... CATCH** block.

The following is an example showing the implementation of error handling in T-SQL. In this example, we'll intentionally raise an error that will be caused by dividing a value by

Before we see the implementation of **TRY ...** let's see what will happen if we'll execute the same statement without embedding it in **TRY ...**. The following is the T-SQL example to intentionally cause an error:

SELECT

When you'll run this statement, the error would be returned as can be seen in the following screenshot:

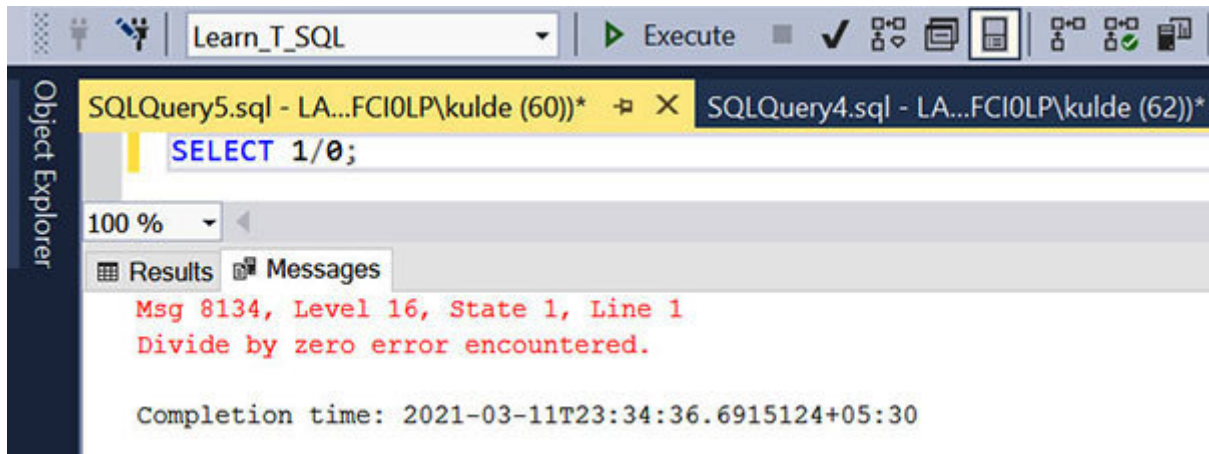


Figure 11.1: Query throwing error

Let us now see how the same statement will behave if we'll embed it in a **TRY ... CATCH** block. In the following example, we've used a table variable to store the error details. You can also log it in a physical table or simply return the error by the way of a **SELECT** statement on the error-related system functions:

```
DECLARE @MyTable TABLE
(
  ErrorNumber    INT
, ErrorState    INT
,
, ErrorLine     INT
,
,
);

BEGIN TRY
```

```
SELECT  
END TRY
```

```
BEGIN CATCH  
INSERT INTO @MyTable  
VALUES  
SELECT * FROM  
END CATCH
```

When you'll run this T-SQL code, the output will be similar as shown in [figure](#)

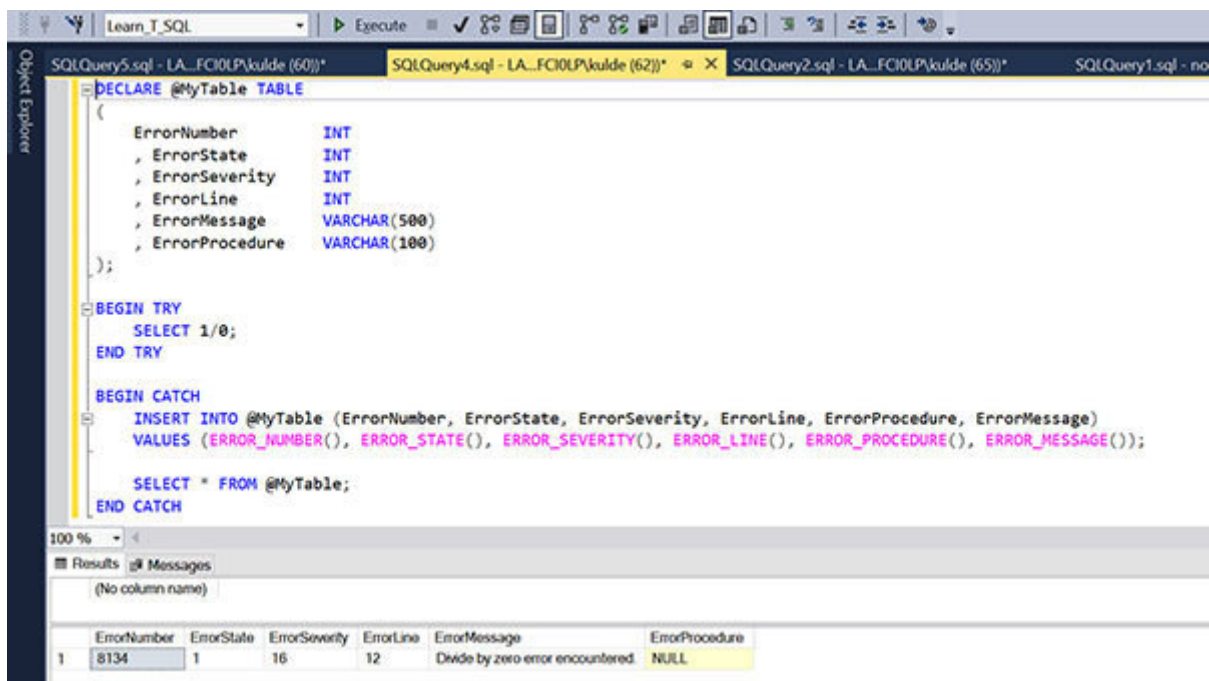


Figure 11.2: Output of query with error handling

Let us see another example as follows, for the same scenario with but without a **TRY ... CATCH** block:

```
SELECT
IF @@ERROR <> 0
BEGIN
PRINT 'Oops, an
END
PRINT 'Line after
```

When you'll run the preceding code, the output will be as can be seen in the following screenshot. As you can see there is an error, but the execution continued. We were able to catch the error with but we could not handle it:

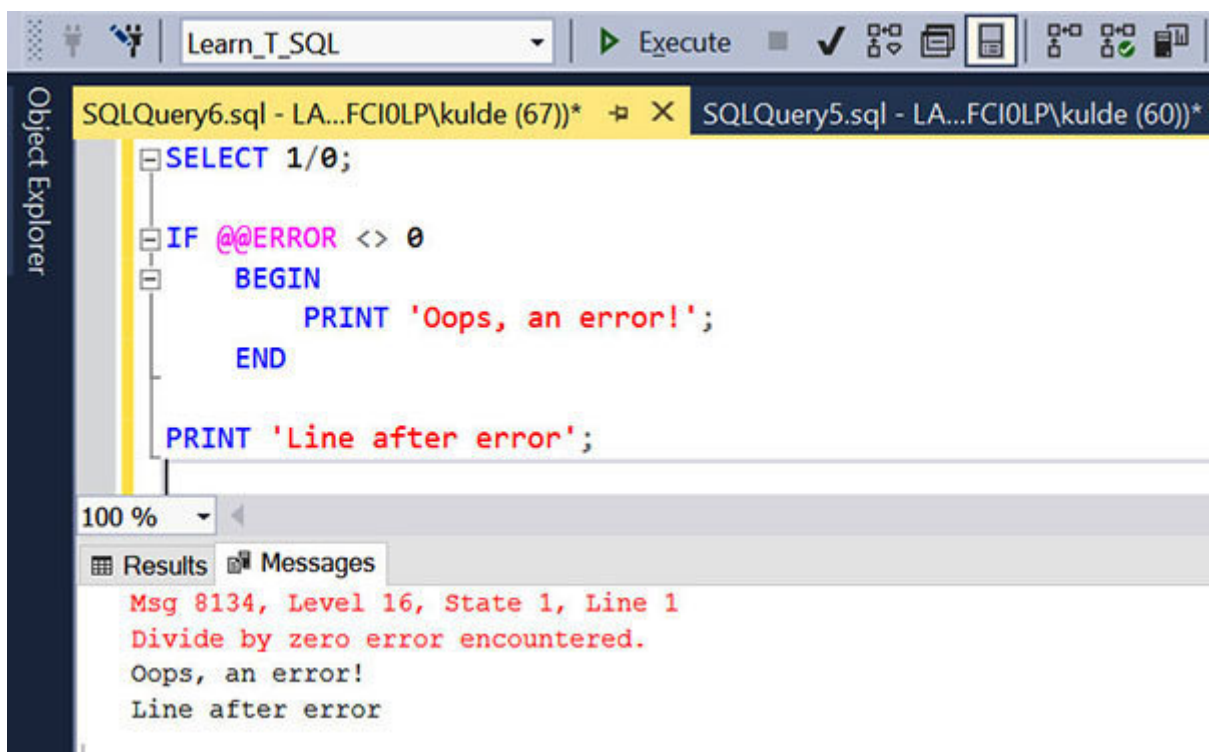


Figure 11.3: Output of the query to catch error using @@ERROR

But when we embed the same T-SQL code in a **TRY ... CATCH** block, the control will be passed to the **CATCH** block, and the statements following the statement thrown error will not be executed. You can try it with the help of the following T-SQL code:

```
BEGIN TRY
SELECT
IF @@ERROR <> 0
BEGIN
PRINT 'Oops, an
END
PRINT 'Line after
END TRY
BEGIN CATCH
PRINT
END CATCH
```

The behavior of the preceding T-SQL code will be similar to as can be seen in the following screenshot:

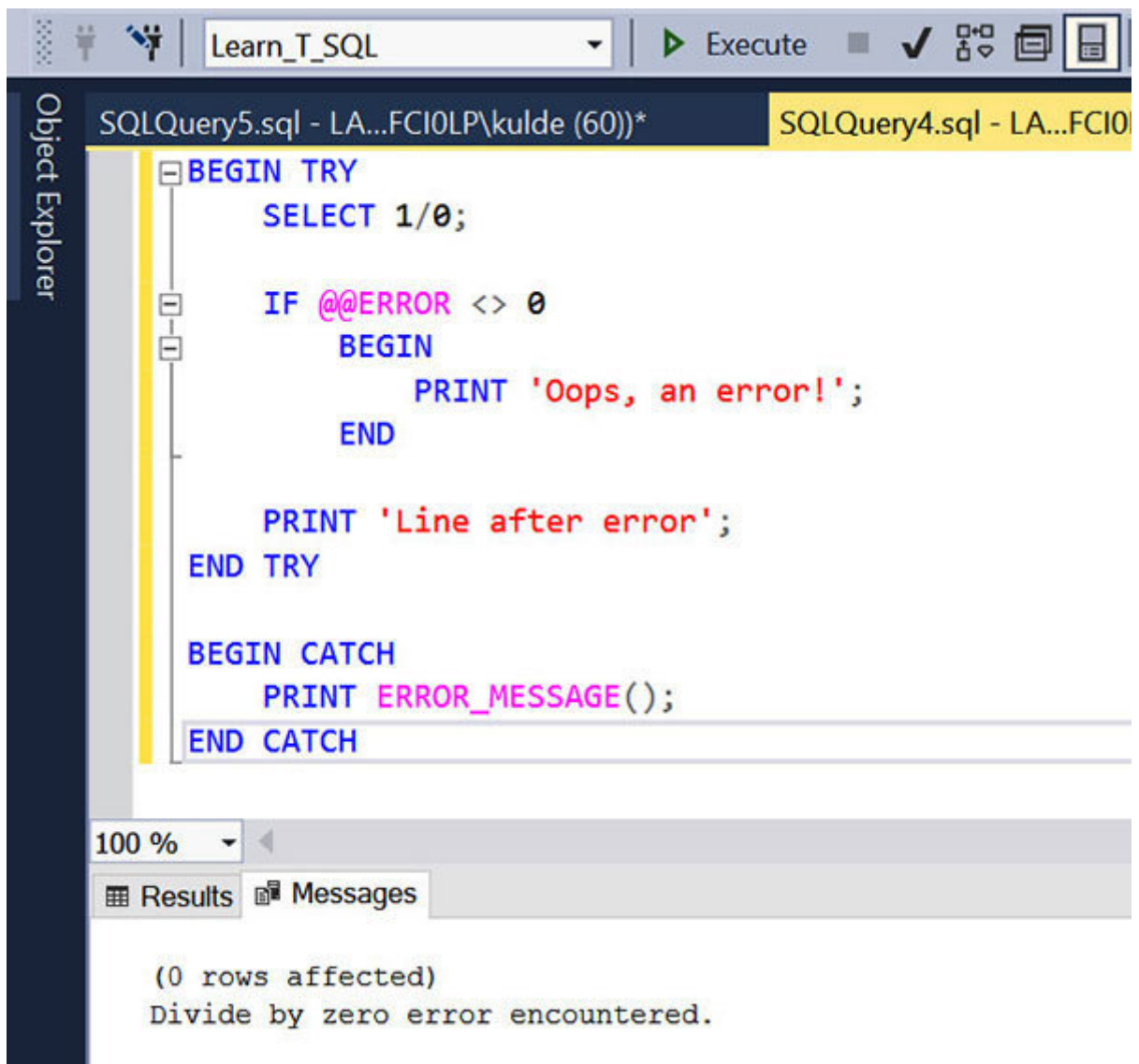


Figure 11.4: Output of query to catch and handle the error

Transaction management

The transaction can be understood as a set of instruction(s), considered as one. In T-SQL, it can be understood as a set of DML statement(s), considered as one. A transaction can either be successful or fail, but not partially. Partially means some instruction (or DML statement) be successful and some fails. This is the atomicity of the ACID property of RDBMS.

Transaction in SQL Server is of four types as follows:

Auto commit transaction: Each individual statement is a transaction.

Implicit transaction: When the prior transaction is completed, a new transaction is implicitly started. A transaction is completed explicitly with a commit or rollback statements. There is no way to implicitly complete a transaction.

Explicit transaction: This type of transaction is explicitly defined (or, started) with **BEGIN TRANSACTION** and explicitly completed with **COMMIT TRANSACTION** or **ROLLBACK TRANSACTION** statements. In this topic, we'll focus on this explicit transaction.

Batch-scoped transaction: Applicable only to multiple active result sets (MARS).

The following statements starts an implicit transaction in SQL Server:

ALTER TABLE

BEGIN TRANSACTION

CREATE

DELETE

DROP

FETCH

GRANT

INSERT

OPEN

REVOKE

SELECT (only the **SELECT** statements referring tables)

TRUNCATE TABLE

UPDATE

When you'll run any of these statements, you'll be starting a transaction, even without starting it explicitly. Mostly, the transaction will be committed automatically too. This is an implicit transaction. Here, you started and committed a transaction without explicitly doing so.

IMPLICIT_TRANSACTIONS setting plays an important role here.

You can read more of it at the following URL of Microsoft official documentation:

<https://docs.microsoft.com/en-us/sql/t-sql/statements/set-implicit-transactions-transact-sql?view=sql-server-ver15>

The Implicit Transactions are internally managed by SQL Server with the **IMPLICIT_TRANSACTIONS** setting, whereas the Explicit Transactions are started and completed explicitly. The Explicit Transactions are started, committed, and rolled back, with the following statements:

BEGIN Starts an explicit transaction. An explicit transaction may or may not have a name. You may or may not specify the name of the transaction. Transaction name is optional. However, it is advisable to specify it. You can have multiple

transactions. You'll find the transactions with names easy to handle.

COMMIT Completes the transaction by committing it.

ROLLBACK Completes the transaction by rolling back it to the last successful point, prior to the **BEGIN**

@@TRANCOUNT

@@TRANCOUNT is a global variable that returns the current active transaction count in a session. Each time a **BEGIN TRANSACTION** statement is executed, the **@@TRANCOUNT** increments by 1. But when a **COMMIT TRANSACTION** statement is executed, the **@@TRANCOUNT** is decremented by 1. When you'll fire a **ROLLBACK** statement, the **@@TRANCOUNT** becomes 0 and all the active transactions in the session are rolled back.

Let's see the following T-SQL example showing how the transaction works:

```
BEGIN TRANSACTION  
SELECT @@TRANCOUNT
```

```
BEGIN TRANSACTION  
SELECT @@TRANCOUNT
```

```
BEGIN TRANSACTION  
SELECT @@TRANCOUNT
```

```
BEGIN TRANSACTION  
SELECT @@TRANCOUNT
```

```
BEGIN TRANSACTION  
SELECT @@TRANCOUNT
```

```
COMMIT TRANSACTION  
SELECT @@TRANCOUNT
```

```
COMMIT TRANSACTION  
SELECT @@TRANCOUNT
```

```
ROLLBACK TRANSACTION  
SELECT @@TRANCOUNT
```

When you'll execute the preceding T-SQL code, the output will look similar to as shown in the following screenshot:

	(No column name)	
1	1	
	(No column name)	
1	2	
	(No column name)	
1	3	
	(No column name)	
1	4	
	(No column name)	
1	5	
	(No column name)	
1	4	
	(No column name)	
1	3	
	(No column name)	
1	0	

Figure 11.5: Output of the query with transaction

XACT_STATE()

XACT_STATE() is the system scalar function that returns the state of the current running request. It indicates if the request has an active user transaction, and if the transaction can be committed.

The return type of this function is **int**. It returns the following values:

1: Represents the current request has an active user transaction.

0: Represents the current request has no active user transaction.

-1: Represents the current request has an active user transaction, but some error has occurred which has made the transaction an uncommittable transaction. The uncommittable transactions cannot be committed or roll backed.

XACT_STATE and **@@TRANCOUNT** can be used together to detect if the current request has an active user transaction.

@@TRANCOUNT can be used to check the transaction count, and **XACT_STATE** can be used to determine if the transaction is committable. You would not want to see an

error, even after implementing the stringent error handling mechanism. **XACT_STATE** can help you to avoid the errors caused due to uncommittable transactions.

Implementing explicit transaction

The following is a sample example depicting the implementation of transaction in T-SQL. In this example, **BEGIN** is used to define the transaction. The DML statements specified after **BEGIN** and before **COMMIT** are treated as one transaction.

The transaction is good in one sense, but otherwise in another sense. If you've a transaction then all the tables on which the read or write is being performed will be locked, until the transaction is committed or roll backed. That is why it is advisable to keep the transactions short as much you can:

BEGIN

DML statement 1

DML statement 2

DML statement 3

IF =

BEGIN

COMMIT

END

ELSE

BEGIN

ROLLBACK

END

It is advisable to use the transaction with **TRY ... CATCH** block. A sample implementation of **TRANSACTION** with **TRY ... CATCH** block will look like shown as follows:

```
BEGIN TRY
```

```
BEGIN
```

```
DML statement 1
```

```
DML statement 2
```

```
DML statement 3
```

```
IF =
```

```
BEGIN
```

```
COMMIT
```

```
END
```

```
END TRY
```

```
BEGIN CATCH
```

```
ROLLBACK
```

```
SELECT
```

```
END CATCH
```

You can read more about transactions on the Microsoft official documentation at the following URL:

<https://docs.microsoft.com/en-us/sql/t-sql/language-elements/transactions-transact-sql?view=sql-server-ver15>

Explicit transactions seem very easy as far as syntax is concerned. But it can be very challenging as far as

implementation is concerned, especially when there are multiple transactions. The situation complicates further if the transactions are nested. You can have nested explicit transactions, but be careful while doing so.

Conclusion

T-SQL is not only writing queries. L of T-SQL is Language. T-SQL is a language. A language has to be comprehensive to offer end to end solutions. T-SQL does that. It allows you write a program that is complete in all the aspect. Error handling and transaction management are some of such features to make the T-SQL complete in the programming aspect.

Transaction management helps you start and complete the transaction (a set of instructions) appropriately, the way you want. Error handling makes your program error free. Error free does not really mean no error. But it enables you to handle the error, and plan the alternative action in case of error. There is no way to make your program error free completely. But yes, you can handle the error and plan the alternative actions to avoid the unintended behavior of your program.

In the next chapter, you will learn about data conversion, cross-database, and cross-server data access.

Points to remember

A **TRY** block should have an immediate **CATCH** block. You cannot have a **TRY** block without an immediate **CATCH** block, and vice versa.

If there is no error in the T-SQL statements enclosed in a **TRY** block, the control will bypass the **CATCH** block, and the control will pass to the T-SQL statement immediately after **END CATCH** statement.

If there is an error in the T-SQL statements enclosed in a **TRY** block, then control passes to the first T-SQL statement immediately after the **BEGIN CATCH** statement.

If the **END CATCH** statement is the last statement in a T-SQL code, then the T-SQL program execution will end in both the cases – error and no error. In case of error, the T-SQL statements specified within a **CATCH** block will be executed before the execution ends. Whereas in case of no error, **CATCH** block will be bypassed, and execution will end.

A code embedded in **TRY ... CATCH** block will not return an error, unless specifically returned using **RAISEERROR** or **THROW** commands. Errors are generally returned by SQL Server, but you can also raise the custom errors using these commands.

You can use **TRY ... CATCH** block in stored procedure and trigger. But you cannot use it in a user-defined function and view.

Implicit transactions are internally managed by SQL Server with **IMPLICIT_TRANSACTIONS** setting.

Explicit transactions are started and completed explicitly started (using **BEGIN** committed (using **COMMIT** and rolled back (using **ROLLBACK** explicitly.

BEGIN TRANSACTION and **BEGIN TRAN** are same.

COMMIT TRANSACTION and **COMMIT TRAN** are same.

ROLLBACK TRANSACTION and **ROLLBACK TRAN** are same.

You can also commit a transaction with just **COMMIT** command, without specifying or **TRAN** word with it.

You can also rollback a transaction with just **ROLLBACK** command, without specifying or **TRAN** word with it.

You can retrieve the transaction count using **@@TRANCOUNT** global variable.

Active transactions can be found in **sys.dm_tran_active_transactions** DMV.

COMMIT TRANSACTION decrements the transaction count by 1. The transaction name assigned with it does not really make sense. In the case of nested transactions, each time **COMMIT TRANSACTION** is executed and if there are more than one active transaction, it will decrement the transaction count by 1 but the transaction stays active.

Once there will be only one transaction remaining, or the **@@TRANCOUNT** becomes 1 in the session (can be retrieved using the actual impact of takes place. The actual impact would be as per the DML statements that are the part of the transaction. The data modifications requested in the transaction become permanent part of the database. The transaction's resources will be freed-up, and **@@TRANCOUNT** is decremented to 0.

ROLLBACK TRANSACTION rolls back all the active transactions in a session. In the case of a nested transaction, you can only specify the name of the parent transaction with **ROLLBACK** transaction. In case you'll try to rollback a child transaction, the statement will fail with the error **Cannot roll back name>. No transaction or savepoint of that name was**

You can use explicit transaction in stored procedures and triggers. But you cannot use it in a user-defined function and view.

Anything starting with @@ as a prefix is a global variable.

Multiple choice question

If there is a parent transactions P₁, and a child transaction P₂. What will happen if you executed ROLLBACK TRANSACTION

There will be an error.

Statement will execute successfully.

Can we have a BEGIN TRY block, without a BEGIN CATCH block?

Yes

No

If there are five transactions – P₁, P₂, P₃, P₄, and P₅. If you have executed COMMIT statement once. What will be the

5

0

4

NULL

If there are five transactions – P₁, P₂, P₃, P₄, and P₅. If you have first executed COMMIT and then ROLLBACK statements, individually once each. What will be the

5

0

4

NULL

You've a T-SQL code in a .sql file. You have opened it and executed it. The query has TRY ... CATCH block. It is simple T-SQL code, and no stored procedure is involved. What will be the value of ERROR_PROCEDURE() function?

Name of .sql file

NULL

Answers

A

B

C

B

B

Questions

When **TRY ... CATCH** block cannot be used?

What should be ideally done in a **CATCH** block?

Why and when you can use

Can we get the error details using functions such as and so on in **TRY** block?

Can we open a transaction in a **CATCH** block?

What is

Can we commit transactions by the transaction name? If yes, consider the **@@TRANCOUNT** is Will the changes of that particular committed transaction take effect immediately after the commit?

Can you commit a transaction by simply using the **COMMIT** statement?

Can you rollback a transaction by simply using the **ROLLBACK** statement?

How to detect if the current request submitted for commit has a committable transaction?

Key terms

BEGIN TRY ... END TRY represents the **TRY** block. It tries/tests the embedded statements for the errors.

BEGIN CATCH ... END CATCH represents the **CATCH** block. It catches the error raised by **TRY** block.

ERROR_NUMBER() function returns the error number.

ERROR_SEVERITY() function returns the error severity.

ERROR_STATE() function returns the error state.

ERROR_PROCEDURE() function returns the error procedure.

ERROR_LINE() function returns the error line.

ERROR_MESSAGE() function returns the error message.

@@ERROR is a global variable that returns the values of type integer.

BEGIN TRANSACTION starts an explicit transaction.

COMMIT TRANSACTION completes the transaction by committing it.

ROLLBACK TRANSACTION completes the transaction by rolling back it to the last successful point, prior to the **BEGIN**

@@TRANCOUNT is a global variable that returns the current active transaction count in a session.

XACT_STATE() is system scalar function that returns the state of current running request.

CHAPTER 12

Data Conversion, Cross-Database, and Cross-Server Data Access

Data are of various categories, such as **Numeric** , **String** , **Date** and **Time** , and **Boolean** . There are various datatypes under each of these categories. You may want to convert a data of one type to another type, such as **Integer** to **String** , or vice versa. This is where data conversion is required. There are various built-in conversion functions, which we'll talk about in this chapter.

You may also have to access the data stored in the other database, on the same server, or on a different server. This chapter will also talk about how to access the data from another database on the same or a different server.

Structure

In this chapter, we will cover the following topics:

Data conversion

CAST and **CONVERT**

PARSE

and **TRY_PARSE**

Cross-database data access

Cross-server data access

Objective

The data conversion functions enable to deal with the variety of the data. You can convert data between two different datatypes. There are functions that you can use to test whether a value (hard-coded, or from a variable, or a column) is of a specific datatype. You will also learn to convert date and time values to various date and time formats.

Cross-database data access means accessing another database from a database, but on the same server. Cross-server data access means accessing another database from a database on the different server. After learning the cross-server and cross-database data access, you will be able to perform the DML actions from one database to another database, on the same, or a different server.

Data conversion

Definition of conversion is *the process of changing or causing something to change from one form to*

When we say data conversion, the definition will become *the process of changing or causing data to change from one datatype to*

I hope the definition given above is sufficient to make you understand what data conversion means. Let us now understand *why and when we need data conversion?*

Consider you are required to concatenate two columns. One of these columns is of but another one is As we all know, concatenation is a string operator, which works only on string values. You can concatenate data only of **String** datatypes. Hence, you need to first explicitly convert the **INT** data to and then you can easily concatenate. If you won't convert the integer to string, then be ready for an unexpected error. Don't believe me, let us see the following example:

We'll replicate the same example as discussed earlier in the form of a T-SQL example as follows:

DECLARE

```
, @INT
```

```
SELECT 'Year'
```

```
, @INT =
```

```
PRINT @String + ' ' + @INT + ' is
```

If you try running the preceding T-SQL code, you will get the conversion error as can be seen in the following screenshot:

```
Msg 245, Level 16, State 1, Line 7  
Conversion failed when converting the varchar value 'Year ' to data type int.
```

Figure 12.1: Before explicit conversion

Let us see how we can avoid this error. We can make use of either **CAST** or **CONVERT** functions to convert **INT** to We'll talk about these functions in detail as we'll discuss them. For now, let's see what happens when we'll use these functions:

```
DECLARE
```

```
, @INT
```

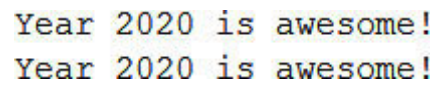
```
SELECT 'Year'
```

```
, @INT =
```

```
PRINT @String + ' ' + AS + ' is
```

```
PRINT @String + ' ' + + ' is
```

The output of the preceding T-SQL code would be similar to as can be seen in the following screenshot:



```
Year 2020 is awesome!  
Year 2020 is awesome!
```

Figure 12.2: *After explicit conversion*

The scenario we discussed is just a sample example. There could be many more such scenarios where you may find data conversion extremely important.

I know you would be excited to know about all these data conversion functions. *Here you go!* We'll talk about the following functions in this chapter. We'll also see how to use the **CONVERT** function, to convert a date and time value, in various date and time formats:

CAST

CONVERT

PARSE

TRY_CAST

TRY_CONVERT

TRY_PARSE

Using **CONVERT** function for converting date and time

We'll talk about these functions. But before we do that let's understand two important topics – **implicit** and **explicit** conversion. Both of these are data conversions. Both will apply only in the case of differing datatypes.

Implicit conversion is handled automatically by SQL Server, whereas explicit conversion has to be done programmatically, with the help of **CAST** or **CONVERT** functions. When you use any of these functions to explicitly convert the data from one datatype to another, the process is called an **explicit**

You would be wondering why we need explicit conversion when we already have implicit conversion. Nobody would want to spend effort on something that is really not required. But that is not the case. Implicit conversion is not to be confused with explicit conversion.

Let's see some examples of implicit conversion. The following T-SQL code shows the examples of implicit conversion:

```
DECLARE
```

```
,
```

```
SELECT 'Year'
```

```
,
```

```
PRINT @String1 + ' ' + @String2 + ' is
```

The followings are the implicit conversions in the preceding T-SQL example. Ideally, the T-SQL code should return an error of conversion, but it won't. Because implicit conversion has done its job:

@String2 is of **CHAR** datatype, which of string type. As we know, the string values should be enclosed within single quotes. But in the following example, the value **2020** is not enclosed in single quotes.

@String1 is of **VARCHAR** datatype, and **@String2** is of **CHAR** datatype. Both are different datatypes, but belongs to string. In the **PRINT** statement, you could see they are concatenated.

The aforesaid T-SQL code will run successfully and would return the output similar to as can be seen in the following screenshot:

A screenshot of a SQL Server query result. The text 'Year 2020 is awesome!' is displayed in a monospaced font, with each word on a new line. The text is color-coded: 'Year' is blue, '2020' is red, 'is' is green, and 'awesome!' is blue.

Figure 12.3: *Implicit conversion*

Generally, when conversion is required between two datatypes of the same category, the implicit conversion plays the role. For example, and and so on are numeric datatype. The conversion between any of these datatypes will cause implicit conversion.

NCHAR and and so on are **String** datatype. The conversion between any of these datatypes will cause implicit conversion.

and so on are date-time datatype. The conversions between any of these datatypes will cause implicit conversion.

Read more on data type conversion at the following URL of Microsoft official documentation:

<https://docs.microsoft.com/en-us/sql/t-sql/data-types/data-type-conversion-database-engine?view=sql-server-ver15>

CAST and CONVERT

CAST and **CONVERT** functions both are used for explicit conversion to convert data from one type to another. Mostly, both are the same, as far as purpose is concerned.

CONVERT can do everything as **CAST** can also do formatting based on the style supplied as an argument. Syntax of both of them is pretty simple as follows:

CAST (expression **AS** data_type [(length)])

CONVERT (data_type [(length)] , expression [, style])

and **length** are common arguments in both **CAST** and **CONVERT** function.

Where:

expression can be an individual value, or a variable, or a column, or a scalar subquery.

data_type is the destination datatype, or the datatype to which the conversion has to be done.

length is an optional integer argument. It could be length or precision and scale, depending upon the **data_type** argument.

Since there are various datatypes such as and so on. which do not accept any length, or precision, and scale. Hence, this argument is optional.

style is an optional integer argument and is only available in the **CONVERT** function. There are various kinds of styles that can be used with **CONVERT** function such as:

Date and **time** styles.

float and **real** styles.

money and **smallmoney** styles.

xml styles.

binary styles.

Out of these styles, we'll only talk about date and time styles, to see what can be done with them using the **CONVERT** function.

The following table represents the possible date and time style values, and their respective output formats. Additionally, there are few more styles, which are not included in the table. They are **126 127 (ISO8601 with time 130 and 131**

Let us see some examples as follows, converting the date and time values in various formats:

```
SELECT AS [100]  
SELECT AS [101]  
SELECT AS [102]  
SELECT AS [103]  
SELECT AS [104]  
SELECT AS [105]  
SELECT AS [106]  
SELECT AS [107]
```

GETDATE() is a built-in function, which returns the current date and time of the server or system, on which SQL Server is installed. You may get different values when you'll execute these queries on your system.

The result of the preceding query will be similar to as can be seen in the following screenshot:

	100	
1	Mar 25 2021 2:02AM	
	101	
1	03/25/2021	
	102	
1	2021.03.25	
	103	
1	25/03/2021	
	104	
1	25.03.2021	
	105	
1	25-03-2021	
	106	
1	25 Mar 202	
	107	
1	Mar 25, 20	

Figure 12.4: Output of query to convert the date-time in various formats

When you supply the length of the **VARCHAR** datatype in the **CONVERT** function for converting the date and time value to various formats, it is important that the length should be sufficient to accommodate the number of characters as per the format. For example, style **101** returns the date and time value in the format `Mar 25 2021 2:02AM`. If you'll count the number of characters then it's **10**. Hence, the length should be greater than or equals to **10**. In case a less value is supplied as length then there would be truncation of the resulting values.

You can also supply just **VARCHAR** without actually supplying the length, as can be seen as follows. It will work, and the resulting output will be similar to [figure](#). However, on a subsequent run, the date and time value will differ:

SELECT

SELECT

SELECT

SELECT

SELECT

SELECT

SELECT

SELECT

PARSE

CAST and **CONVERT** functions convert an expression, which can be of any datatype, to a different datatype. Source and target can be of any datatype. There is no restriction as such.

However, the **PARSE** function converts an expression, which mandatorily has to be a string value, to either a numeric or a date and time datatypes. It also accepts an optional **culture** argument. If the **culture** argument is not provided, then the language of the current session is used. If the **culture** argument is not valid, **PARSE** raises an error.

The following is the syntax of **PARSE** function:

```
PARSE ( string_value AS data_type culture ] )
```

The following is an example with **PARSE** function:

```
SELECT AS MONEY USING
```

Explanation of the preceding example:

€256,56 is the first argument (or, expression of type string).

MONEY is the second argument (or, datatype, or, target datatype) to which the expression is to be converted. **AS** keyword is mandatory to specify before the datatype.

fr-FR is the third argument (or, that means French. **USING** keyword is mandatory to specify before

The value **€256,56** can be converted with **PARSE** function, using the cultures pertaining to the European countries using the currency euro

The result of the preceding query will be

Let us try the same conversion with some other cultures such as It is to be noted, *Norway* is part of the but has its own currency called *Norwegian*

SELECT AS MONEY USING

An error will be thrown, similar to as can be seen as follows, upon running the preceding query. A similar errors will be thrown for other incompatible cultures:

```
Msg 9819, Level 16, State 1, Line 1
Error converting string value '€256,56' into data type money using culture 'nn-NO'.
```

Figure 12.5: Error thrown due to incompatible culture

If you will try to convert the same value **€256,56** with **CAST** and the resulting output will be messed up, as can be seen in the following screenshot:

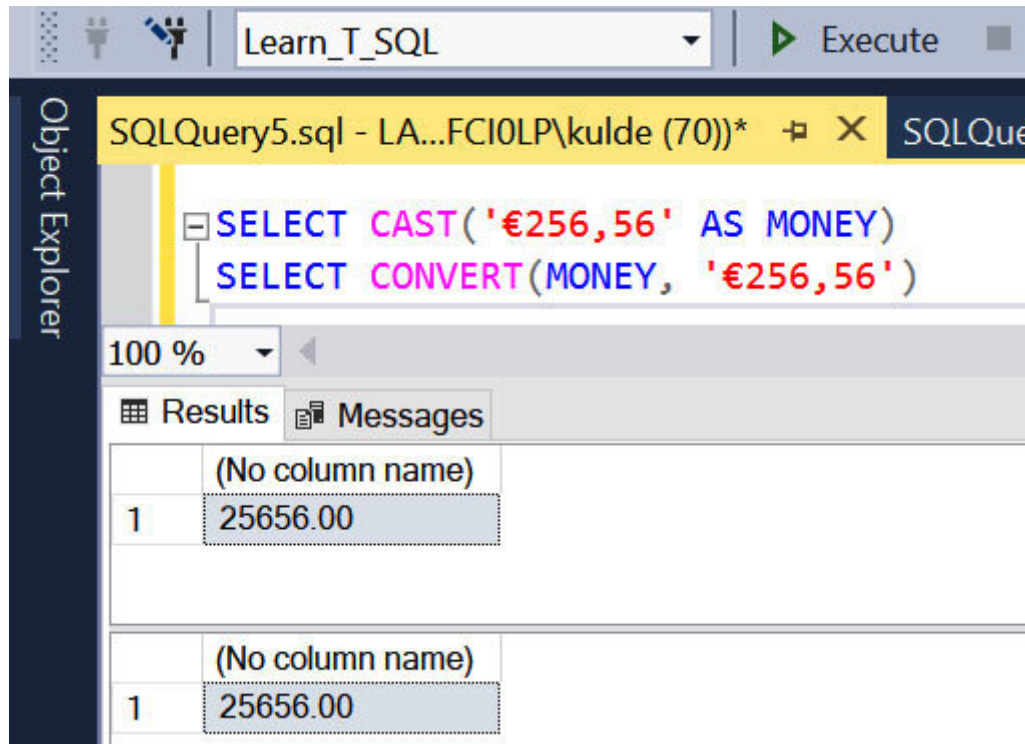


Figure 12.6: Data getting messed up with CAST and CONVERT functions

TRY_CAST, TRY_CONVERT, and TRY_PARSE

These are lovely functions, which do the exactly same thing as that of and They also do something that and **PARSE** does not.

and **PARSE** functions return an error if conversion fails due to incompatible datatypes, style, or culture. But and **TRY_PARSE** do not return any error, if conversion fails due to incompatible datatypes, style, or culture. Instead, they return

Don't go by my words. *Let us try it!*

The following examples of and which has made to intentionally raise errors:

```
SELECT AS MONEY USING  
SELECT AS  
SELECT
```

An error will be thrown similar to as can be seen in the following screenshot, upon running the preceding queries:

```
Msg 9819, Level 16, State 1, Line 1  
Error converting string value '@256,56' into data type money using culture 'nn-NO'.  
Msg 245, Level 16, State 1, Line 2  
Conversion failed when converting the varchar value '@256,56' to data type int.
```


Figure 12.7: Conversion error with *PARSE*, *CAST*, and *CONVERT* functions

Let's try the same conversion with the help of and An example is mentioned as follows:

```
SELECT AS MONEY USING  
SELECT AS  
SELECT
```

Magic! The query ran successfully. Although, the NULL's have been returned, as shown in the following screenshot:

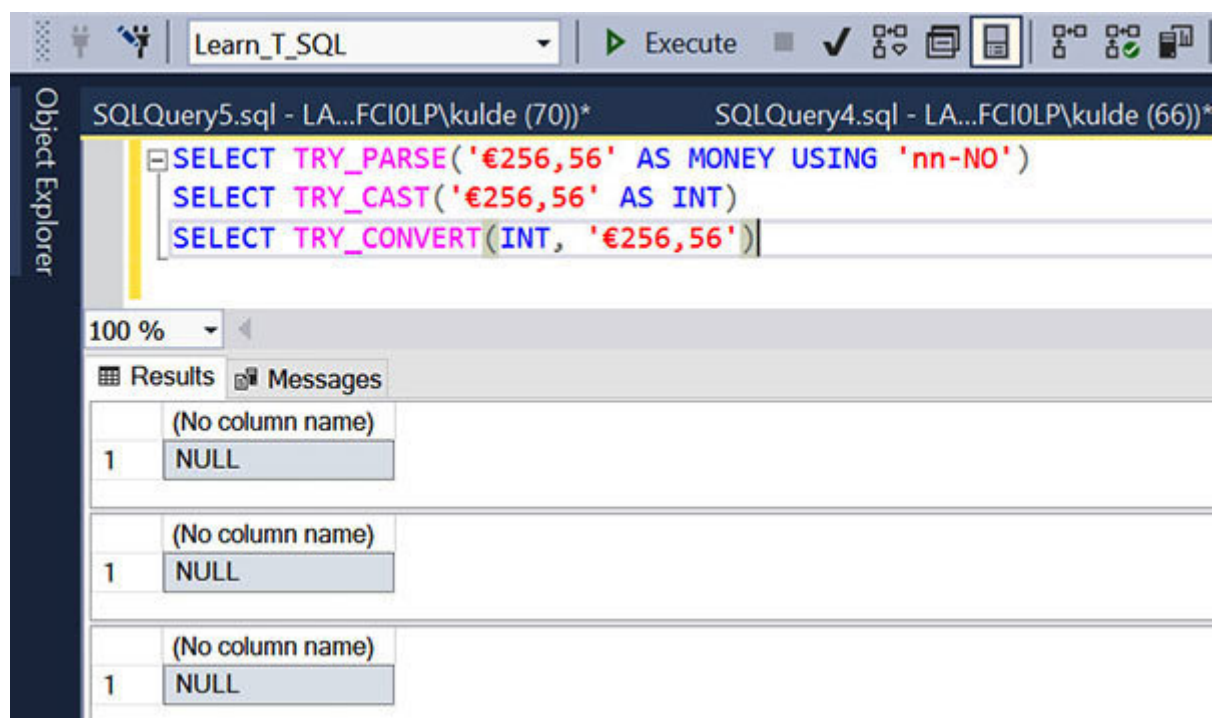


Figure 12.8: Handling conversion errors with *TRY_PARSE*, *TRY_CAST*, and *TRY_CONVERT* functions

Errors are the most annoying things in any programming. I found these functions quite helpful in tackling the errors caused due to conversion failure.

You can read more about conversion, and about all these functions we discussed in this chapter, at the following URL of Microsoft official documentation:

<https://docs.microsoft.com/en-us/sql/t-sql/functions/conversion-functions-transact-sql?view=sql-server-ver15>

Cross-database data access

Cross-database data access is a mechanism to read the data from, or write to, another database, from a database. There are two or more databases involved in cross-database data access, and all of them are on the same server.

Cross-database data access requires three-part naming, represented by the following syntax:

..

Default name> is current database and default name> is dbo. If these options are not specified then SQL Server will treat the query is to be executed on dbo schema on the current database.

Instead of just object name, also specify the database and schema name, and it becomes three-part naming. Specify the name of the object such as a table, in this format, in order to read data from, or write to, another database, from a database, on the same server.

In order to use this feature, the SQL Server user should have the access to the participating databases, with the necessary rights, and permissions to execute DDL and DML commands.

Cross-server data access

Cross-server data access is a mechanism to read the data from, or write to, another database on a different server, from a database (on a different server). There are two or more databases involved in cross-server data access, and each of them is on a different server.

Cross-server data access requires four-part naming represented by the following syntax:

...

Servers can be linked with the help of linked servers. You need to supply the name of the linked server in place of **server_name** of the four-part naming.

For example, there are two servers – server **S1** and Server **S1** has database Server **S2** has database If you want to access database **D2** from database **D1** of server then you will have to create a linked server on server **S1** for server

Read more about linked servers at the following URL of Microsoft official documentation:

<https://docs.microsoft.com/en-us/sql/relational-databases/linked-servers/linked-servers-database-engine?view=sql-server-ver15>

Conclusion

Data conversion functions make our life easy by giving us the flexibility to convert the data between various types. In the absence of these functions, the situation would have been very difficult. The best part is you can even test a given value for a datatype before you actually convert. This gives a great control to a developer to handle the conversion in a much better way, which in turn helps avoid the conversion errors. You will definitely not want to see errors in your program, *right!*

Cross-database and cross-server data access too is a great feature. It makes the T-SQL querying much easier. You can treat any other database like a table in the same database. It does not matter how many databases you have. You can query any database from any database. Even multiple databases from a database. Linked servers also enable to query even a remote database of a different technology such as Oracle.

In the next chapter, you will learn programmability.

Points to remember

GETDATE() is a built-in function, which returns the current date and time of the server or system, on which SQL Server is installed.

Default **name>** is current database and default **name>** is If these options are not specified then SQL Server will treat the query is to be executed on **dbo** schema on the current database.

and **PARSE** functions return an error if conversion fails due to incompatible datatypes, style, or culture. But and **TRY_PARSE** do not return any error, if conversion fails due to incompatible datatypes, style, or culture. Instead, they return NULL.

Multiple choice questions

Which function amongst the following can be used to get the date value in the dd-mm-yyyy format?

CAST

TRY_CAST

CONVERT

PARSE

Can we have used the PARSE function to convert a string to BIT datatype?

Yes

No

Answers

C

B

Questions

What is the difference between **CAST** and

What is the difference between **CONVERT** and

What is the difference between **PARSE** and

What is the difference between **CAST** and

What is the difference between **CAST** and

What is the difference between **PARSE** and

Key terms

Three-part naming is represented by:

..

Four-part naming is represented by:

...

CHAPTER 13

Programmability

This chapter is going to be fully loaded. We'll talk about various features and functionalities of SQL Server, which is ancillary to the T-SQL programming. You'll also learn how to bundle the T-SQL statements into various programmability objects such as user-defined functions, views, triggers, and stored procedures. In the process, we'll understand more about each of these programmability objects, their purpose, and finally, the methods to use them in T-SQL, including how to create, modify, drop, and execute them.

Structure

In this chapter, we will cover the following topics:

Dealing with NULL values

Dealing with IDENTITY value

Dynamic SQL

SET NOCOUNT {ON | OFF}

IS and IS NOT

EXISTS and NOT EXISTS

Programmability objects

Views

User-defined functions

Stored procedures

Triggers

Objective

A program has to be reusable. *Replicating the same code, again and again, is not reusability?* Reusability means writing once and using again and again. Programmability objects enable reusability in SQL Server. You can bundle a set of T-SQL codes in a programmability object, and use it as and when needed. You do not need to execute the entire set of T-SQL statements, instead just execute the programmability object and you're done.

You can generate and execute the dynamic T-SQL code using Dynamic SQL. Suppose you have hundreds of tables. You need to generate and execute a select command based on the table name supplied. Only Dynamic SQL can make it possible. In the absence of the Dynamic SQL feature, you'll have to write the individual select statement for each table.

You can use the **ISNULL** function to test the null ability of the expression and return the desired value instead.

COALESCE returns the first not null value from the supplied expressions. You can supply *N* number of expressions to a **COALESCE** function. **NULLIF** tests an expression for a specific value, and if it's a match, then **NULL** is returned.

Dealing with NULL values

Various datatypes can be assigned to a column or variable. *What if you don't have a value for it?* SQL Server treats it as **NULL** has uniform meaning across all the datatypes and that is and **NULLIF** functions give the control to us to deal with **NULL** values.

Let us understand each of these functions in detail.

ISNULL

ISNULL as the name itself suggests tests if the supplied expression is a null and returns another value (or, expression). It accepts two arguments, both of which are expressions. The first argument is the expression to be tested for null, and the second argument is the expression to be returned, if the first expression is a null.

Following is the syntax of ISNULL function:

```
ISNULL ( expression_to_test_for_null ,  
expression_to_be_returned )
```

Where both the expressions **expression_to_test_for_null** and **expression_to_be_returned** can be any of the followings:

Static value/constant

Variable

Column

Scalar valued function

Scalar subquery

It is to be noted that the datatype of **expression_to_be_returned** argument should be compatible with the **expression_to_test_for_null** argument, else conversion error will be thrown. The return type of the function will be that of the

The following are the sample examples depicting T-SQL script with **ISNULL** function:

```
SELECT o);
```

```
DECLARE  
SELECT
```

```
SELECT 1 + NULL), o);
```

```
SELECT 1 + NULL), 1 + 1));
```

The first **SELECT** statement is an example of **ISNULL** function with a static literal. The output of this statement will be o.

The second **SELECT** statement is an example of **ISNULL** function with a variable. The output of this statement will be an empty string

The third and fourth **SELECT** statements are the examples of **ISNULL** function with the scalar subquery. You can actually have any **SELECT** query that should return a scalar value. The output of the third **SELECT** statement will be The output of the fourth **SELECT** statement will be

COALESCE

COALESCE function accepts *N* number of arguments and returns the first non-null value from the given expressions. It is to be noted that the datatypes of all these expressions should be compatible, else conversion error shall be thrown.

The following is the syntax of **COALESCE** function:

COALESCE (expression_1 , expression_2, ..., expression_N)

Where all these expressions can be any of the followings:

Static value/constant

Variable

Column

Scalar valued function

Scalar subquery

For example, if **COALESCE** function has supplied the values as then **1** would be returned. If **3** is supplied to the function

then **1** would be returned. If **2** is supplied to the function then **1** would be returned. In all these examples, **1** is the first non-null values in all the scenarios discussed.

COALESCE can be used instead of but vice-versa is not possible.

The following are the sample examples depicting T-SQL script with **COALESCE** function:

```
SELECT
```

```
DECLARE  
SELECT
```

```
SELECT 1 + NULL),
```

```
SELECT COALESCE 1 + NULL), 1 +
```

The first **SELECT** statement is an example of **COALESCE** function with a static literal. The output of this statement will be 0.

The second **SELECT** statement is an example of **COALESCE** function with a variable. The output of this statement will be an empty string

The third and fourth **SELECT** statements are the examples of **COALESCE** function with the scalar subquery. You can actually have any **SELECT** query that should return a scalar value. The output of the third **SELECT** statement will be The output of the fourth **SELECT** statement will be

NULLIF

NULLIF function returns a **NULL** value if the expressions supplied are equal, else the value from the first arguments (expression) is returned. It accepts two arguments, and both are expressions.

For example, if you want to return **NULL** if a specific expression is You can use **NULLIF** function for this purpose. Suppose the expression **1** is a variable of type integer, and expression **2** is an integer literal/constant with a value of If the value of the variable (expression) will be anything other than **10** then the same value will be returned, but if the value of the variable will be **10** then a **NULL** value will be returned.

The following is the syntax of **NULLIF** function:

NULLIF (expression_1 , expression_2)

Where all these expressions can be any of the following:

Static value/constant

Variable

Column

Scalar valued function

Scalar subquery

The following are the sample examples depicting T-SQL script with **NULLIF** function. This script will return a **NULL** value. However, if the variable value will be changed to any other value or the second argument will be changed to anything other than an empty string then the function will return the value of the expression from the first argument:

DECLARE

SET @Value =

SELECT

Dealing with IDENTITY value

IDENTITY is a column property. The column has to be of numeric datatype in order to use this property. If enabled, it will auto-generate the numeric value for the column. You don't need to specify this column in the **INSERT** statement.

It requires two arguments – **SEED** and **INCREMENT** is the starting value, and **INCREMENT** is the increment value by which each row is to be incremented.

For example, **IDENTITY (1, 1)** will start with **1 (SEED)** and each row will be incremented by **1**. The first row will be the second row will be the third row will be and so on.

IDENTITY (100, 1) will start with **100** and each row will be incremented by **1**. The first row will be the second row will be the third row will be and so on.

IDENTITY (100, 5) will start with **100** and each row will be incremented by **5**. The first row will be the second row will be the third row will be and so on..

The following is a sample T-SQL depicting the implementation and working of **IDENTITY** property. We've created a temporary

table, but the same syntax can be used with physical tables and table variables too:

```
CREATE TABLE #MyTable  
(  
[ID] INT NOT NULL  
,  
);
```

```
INSERT INTO #MyTable VALUES
```

```
SELECT * FROM
```

The output of the aforesaid T-SQL code will be similar to as can be seen in the following screenshot:

	ID	Name
1	1	A
2	2	B

Figure 13.1: Output of the temporary table with an identity column

As we discussed, the column with identity property need not to be supplied in the **INSERT** statement, and we should not. You cannot perform **INSERT** and **UPDATE** on the identity column. There is a way to perform **INSERT** on the identity column, but there is no way to update an identity value.

Let us try this with the following T-SQL example. Make sure you've created the temporary table before you run the following **INSERT** statement:

```
INSERT INTO #MyTable VALUES
```

When you'll run the preceding T-SQL code, you'll get an error as can be seen in the following screenshot. Even if you'll try to insert value in an identity column of a physical table, the similar error will be returned:

```
Msg 544, Level 16, State 1, Line 10  
Cannot insert explicit value for identity column in table '#MyTable' when IDENTITY_INSERT is set to OFF.
```

Figure 13.2: Error when assigning explicit value to an identity column of a temporary table when `IDENTITY_INSERT` is set to `OFF`

If you'll try to insert a value in an identity column of a table variable, you'll get an error similar to as can be seen in the following screenshot:

```
Msg 1077, Level 16, State 1, Line 7  
INSERT into an identity column not allowed on table variables.  
Msg 1077, Level 16, State 1, Line 7  
INSERT into an identity column not allowed on table variables.
```

Figure 13.3: Error when assigning explicit value to an identity column of a table variable when `IDENTITY_INSERT` is set to `OFF`

If you wish to insert a value in an identity column then you need to first run the following T-SQL statement, before you run the **INSERT** statement. When you'll run this statement, automatic generation of identity values in the specified table will be stopped in all the sessions:

SET IDENTITY_INSERT

When **IDENTITY_INSERT** is you need to explicitly specify the value for the identity column, else an error will be thrown as can be seen in the following screenshot:

```
Msg 545, Level 16, State 1, Line 8
Explicit value must be specified for identity column in table '#MyTable'
either when IDENTITY_INSERT is set to ON or when a replication user is
inserting into a NOT FOR REPLICATION identity column.
```

Figure 13.4: Error if explicit value is not assigned to an identity column of a temporary table when **IDENTITY_INSERT** is set to **ON**

IDENTITY_INSERT is a global setting. Make it **OFF** the moment you're done. Refer to the following syntax:

SET IDENTITY_INSERT

Let us try this with the help of the following T-SQL code:

```
SET IDENTITY_INSERT #MyTable
INSERT INTO #MyTable VALUES
SET IDENTITY_INSERT #MyTable
```

`SELECT * FROM`

The output of the aforesaid T-SQL code will be similar to as can be seen in the following screenshot:

	ID	Name
1	1	A
2	2	B
3	3	C
4	4	D

Figure 13.5: *Output of the temporary table when explicit value assigned to identity column*

I hope you've understood what **IDENTITY** property is. Let's now learn few important functions to retrieve the last identity value. One way to retrieve is by running a **SELECT** statement on a table and sorting the result in the descending order based on the identity column value. But *isn't it a lengthy process? Certainly!*

SQL Server came up with interesting yet useful functions to retrieve the last identity value. This will work only if you are doing insert one by one. If you are inserting multiple records at once then I'm not sure if these functions will really help.

@@IDENTITY

Returns the last inserted identity value, across all the tables, and across all the sessions.

Example:

```
SELECT
```

IDENT_CURRENT()

Returns the last inserted identity value of the supplied table, across all the sessions. It accepts the table name as an argument.

Example:

```
SELECT IDENT_CURRENT
```


SCOPE_IDENTITY()

Returns the last inserted identity value of the previous insert statement, in a session. This one you'll mostly use.

Example:

SELECT

Dynamic SQL

Dynamic SQL as the name itself suggests is a feature to generate and execute dynamic T-SQL statements. For example, if you've a set of tables, and you want to select the records from the table selected. You've two options as follows:

Write select statements for the individual tables and embed them in the **IF** statement. Respective select statements will be triggered based on the condition.

Write a Dynamic SQL.

You already know how to write the select statement for a table, so we'll not discuss this part. Let us understand how you can write a Dynamic SQL to select the records dynamically from the selected table. Let us assume we'll get the selection of the table in a variable. So, the table name that is to be selected will be stored in a variable. The following is an example of Dynamic Inline SQL. It is also called **Inline**

DECLARE

,

SELECT @SchemaName = 'dbo'

, @TableName = 'Customer'

EXECUTE * FROM ' + @SchemaName + '.' + @TableName +

The output of the aforesaid T-SQL code will be similar to as can be seen in the following screenshot:

	CustomerID	CustomerName	CustomerAddress	CustomerMobile
1	1	Customer 1	Customer 1 Address	1111111111
2	2	Customer 2	Customer 2 Address	2222222222
3	3	Customer 3	Customer 3 Address	3333333333
4	4	Customer 4	Customer 4 Address	4444444444
5	5	Customer 5	Customer 5 Address	5555555555
6	6	Customer 6	Customer 6 Address	6666666666
7	7	Customer 7	Customer 7 Address	7777777777
8	8	Customer 8	Customer 8 Address	8888888888
9	9	Customer 9	Customer 9 Address	9999999999
10	10	Customer 10	Customer 9 Address	1010101010
11	11	Customer 11	Customer 11 Address	1111111111
12	12	Customer 12	Customer 12 Address	1212121212
13	13	Customer 13	Customer 13 Address	1313131313
14	14	Customer 14	Customer 14 Address	1414141414
15	15	Customer 15	Customer 15 Address	1515151515
16	16	Customer 16	Customer 16 Address	1616161616
17	17	Customer 17	Customer 17 Address	1717171717
18	18	Customer 18	Customer 18 Address	1818181818
19	19	Customer 19	Customer 19 Address	1919191919
20	20	Customer 20	Customer 20 Address	2020202020

Figure 13.6: Output of dynamic inline T-SQL script

You can run the Dynamic SQL for any table by simply changing the schema and table name. This is a wonderful feature, but a dangerous one. This way of Dynamic SQL is prone to **SQL** SQL Injection is a type of attack by an intruder in which the DML statement is manipulated. The intruder can even drop your table, or the database with SQL Injection. Any kind of T-SQL command can be executed with **EXECUTE**

command. Hence, if someone is able to manipulate your request, think about what all he can do. That is why there is another feature in which the SQL is parameterized, instead of the concatenation of SQL string. It is called **Parameterized**

You can read more about SQL Injection at the following URL of Microsoft official documentation:

<https://docs.microsoft.com/en-us/sql/relational-databases/security/sql-injection?view=sql-server-ver15>

Using parameterized SQL, you cannot parametrize the object names such as schema, table, and column name, and so on. If you want to do so then you may have to use the Inline SQL. If you are doing so, take due care of SQL Injection.

The scenario we discussed is not only the possible use case of Dynamic SQL. There are many more. Here is another example. Suppose you want to select from **PurchaseOrder** table for a specific **ProductID** and **CustomerID** as selected. The parameterized SQL will look like as mentioned as follows.

If you are using multiple parameters then you will have to specify all of them in both **@SQLParameters** variable and also in **EXECUTE sp_executesql** command, in the sequence of their definition:

```
DECLARE @ProductID INT  
, @CustomerID
```

```

SELECT 1
, @CustomerID =

DECLARE @SQL

,

SET N'SELECT *
FROM PurchaseOrder
WHERE ProductID = @ProductID
AND CustomerID = @CustomerID;'
SET N'@ProductID INT
, @CustomerID INT'

EXEC sp_executesql

@SQL
, @SQLParameters
, @ProductID
,

```

The output of the aforesaid T-SQL code will be similar to as can be seen in the following screenshot:

	OrderID	TransactionDate	CustomerID	ProductID	Quantity	Rate	Amount
1	1	2019-01-01	1	1	1	150000.00	150000.00

Figure 13.7: Output of dynamic parameterized T-SQL script

The preceding T-SQL code can be written as follows. The output of both these T-SQL scrips will be exactly the same:

```
DECLARE @ProductID INT  
, @CustomerID
```

```
SELECT 1  
, @CustomerID =
```

```
SELECT *  
FROM PurchaseOrder  
WHERE @ProductID  
AND CustomerID =
```

You would be wondering why to use parameterized SQL when we can do the same stuff with normal T-SQL. Yes, you can. But *what if you need a dynamic solution?* By blending Dynamic SQL with parameterized SQL, you can do the great things. You just need to be cautious to make sure your solution is not prone to SQL Injection.

For example, before actually executing the Dynamic SQL, you can check if the values being concatenated to form Dynamic SQL is tempered or *not*? You can do this by searching for any DDL or DML commands in it. One way is to develop a generic scalar function that may accept the value to be tested for any adverse command, and return if the value is safe to be concatenated.

SQL Injection won't be caused when you execute a Dynamic SQL from SSMS. It is possible when the Dynamic SQL is generated and triggered from the application source code. Suppose you've few text fields that have been used in the concatenation to form Dynamic SQL. If the textbox value won't be validated and sanitized, it is prone to vulnerability. It is possible that someone with malicious intention may put a or **DELETE** statement in it. It is like inviting a hacker to hack your solution.

That is why most of the applications do not allow to the input of special characters and commands that may be detrimental. SQL Injection is generally handled at the application level. But being a database developer, you should also take due care by having a second level of defense (wherever possible).

Another way to parameterize the T-SQL is using the Stored Procedure which we'll discuss in the proceeding topics of this chapter. But it is not logical to create a stored procedure for every individual T-SQL statement. Hence, dynamic parameterized SQL plays a vital role to parameterize the query, even without a stored procedure.

SET NOCOUNT {ON | OFF}

When you run a DML statement, SQL Server prints a message showing the count of number of rows affected by the DML statement. This message is shown for each DML statement. SET **NOCOUNT** is an important command that controls whether to show or not to show the message.

Default is

If you make it **ON** then the message will not be shown, but if it is made **OFF** then the message will be shown. **SET NOCOUNT** is specific to a session (or,

The message with the count of number of rows affected, will not be returned for the DML commands executed after **SET NOCOUNT**. However, the message will be returned for the DML commands executed before **SET NOCOUNT** and after **SET NOCOUNT**. If you specify **SET NOCOUNT ON** in a query window on the top then no message will be displayed for any of the DML statements.

In the result sections, there is a tab called These messages are populated here. The following screenshot shows the default behavior (or, **SET NOCOUNT**). You can see the number of rows affected by the last DML statement, which is a **SELECT** statement. If there would have been multiple DML

statements, then the similar message would have been populated for each DML statement:

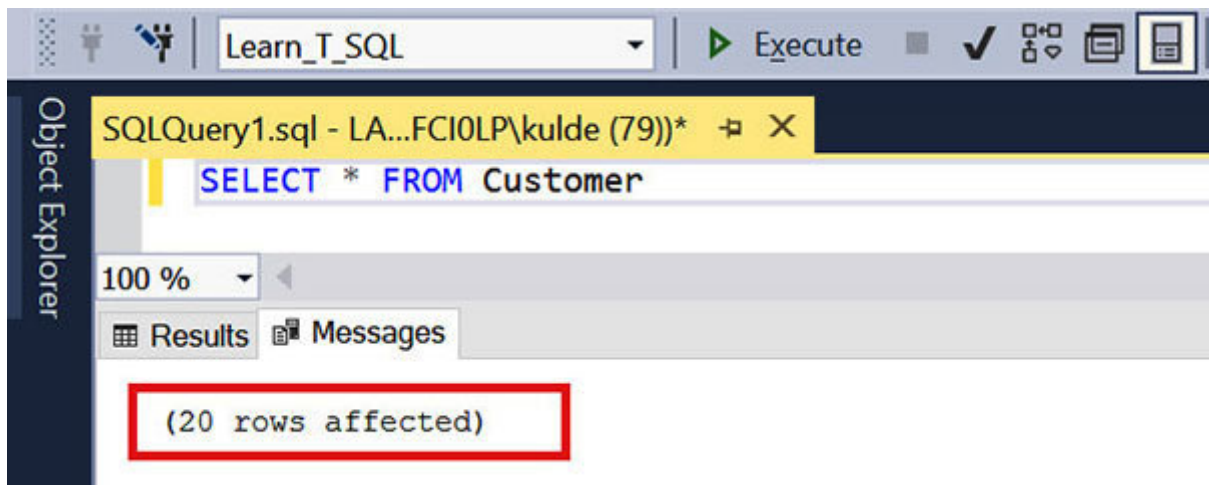


Figure 13.8: Message tab showing the number of rows affected

The following screenshot shows the behavior with **SET NOCOUNT ON**. No such message being populated showing the number of rows affected. Irrespective of whatever number of DML statements you've in the query being executed, you will only see a single message **Commands completed**

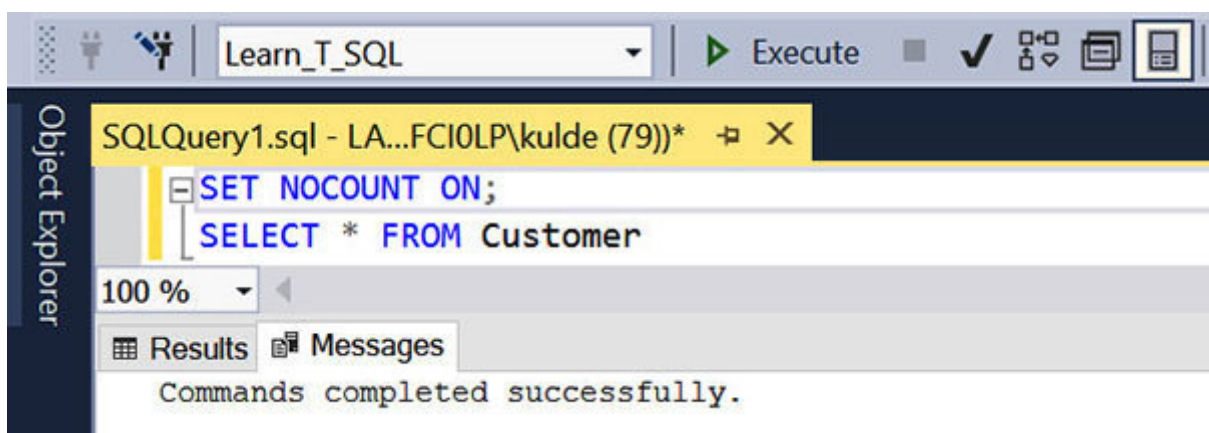


Figure 13.9: *Number of rows affected is not presented after SET NOCOUNT ON*

IS and IS NOT

These are the comparison operators. They are useful in comparing the **NULL** values. Use **IS** operator to compare with the **NULL** as an equality. Use **IS NOT** operator to compare with the **NULL** as an inequality.

If you will try to compare the **NULL** with equality or inequality or then you would be surprised to see it does not work, the way you want. **NULL** is a complex structure and is driven by various parameters (or, settings) such as **ANSI NULLS Enabled** and **ANSI NULL**. These are database-level settings, and by default both of them are. These can be modified at the database level and at the individual query level too.

Database level property can be found and can be changed from the option, as can be seen in the following screenshot:

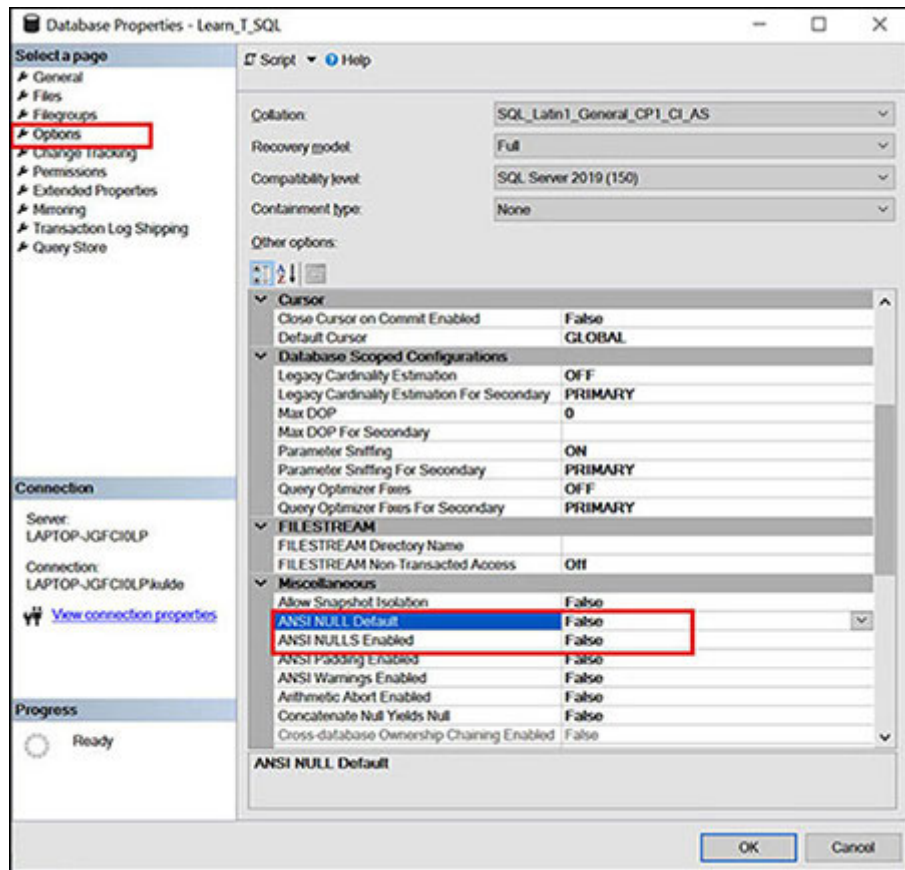


Figure 13.10: ANSI NULL settings

At the individual query level, these settings can be changed using the **SET** command.

You can read more about these settings at the Microsoft official documentation using the following URLs:

<https://docs.microsoft.com/en-us/sql/t-sql/statements/set-ansi-nulls-transact-sql?view=sql-server-ver15>

<https://docs.microsoft.com/en-us/sql/t-sql/statements/set-ansi-null-dflt-off-transact-sql?view=sql-server-ver15>

<https://docs.microsoft.com/en-us/sql/t-sql/statements/set-ansi-null-dflt-on-transact-sql?view=sql-server-ver15>

EXISTS and NOT EXISTS

These are the built-in functions that accept the expression as the T-SQL **SELECT** query and return the value of type bit such as 1 or 0. The supplied **SELECT** query can be a simple **SELECT** statement, or the complex **SELECT** statement, or a subquery. Refer to [Chapter 6, Join, Apply, and Subquery](#) to brush up your knowledge on joins and subqueries.

EXISTS

If the supplied **SELECT** query returns any row or value, the **EXISTS** function returns But if the supplied **SELECT** query doesn't return any row or value, the **EXISTS** function returns

EXISTS function can be used in the control flow statements such as **IF** and **CASE** statements. It can be also used in the **WHERE** clause. You cannot directly use it in the **SELECT** statement, but it can be used in **SELECT** statement within a **CASE** statement.

The following is the sample example of **EXISTS** function:

```
-- Count of customers with CustomerID less than 100. 20
such customers exist.
SELECT FROM Customer WHERE CustomerID <
-- Count of customers with CustomerID greater than 100. No
such customer exists.
SELECT FROM Customer WHERE CustomerID >
```

```
IF EXISTS 1 FROM Customer WHERE CustomerID =
BEGIN
SELECT 'CustomerID
ELSE
BEGIN
SELECT 'CustomerID not
```

SELECT FROM Customer

WHERE EXISTS 1 FROM Customer WHERE CustomerID =

The result of the query will look like as can be seen in the following screenshot, considering you have similar data:

Results		Messages
	(No column name)	
1	20	
	(No column name)	
1	0	
	(No column name)	
1	CustomerID not found.	
	(No column name)	
1	0	

Figure 13.11: Output of the query with EXISTS function

NOT EXISTS

NOT EXISTS function behaves exactly opposite of the **EXISTS** function.

If the supplied **SELECT** query returns any row or value, the **NOT EXISTS** function returns But if the supplied **SELECT** query doesn't return any row or value, the **NOT EXISTS** function returns

NOT EXISTS function can be used in the control flow statements such as **IF** and **CASE** statements. It can be also used in the **WHERE** clause. You cannot directly use it in the **SELECT** statement. But it can be used in **SELECT** statement within a **CASE** statement.

The following is the sample example of the **NOT EXISTS** function:

-- Count of customers with CustomerID less than 100. 20 such customers exist.

SELECT FROM Customer **WHERE** CustomerID <

-- Count of customers with CustomerID greater than 100. No such customer exists.

SELECT FROM Customer **WHERE** CustomerID >

```

IF NOT EXISTS 1 FROM Customer WHERE CustomerID =
BEGIN
SELECT 'CustomerID not
ELSE
BEGIN

SELECT 'CustomerID

SELECT FROM Customer
WHERE NOT EXISTS 1 FROM Customer WHERE CustomerID
=

```

The result of the query will look like as can be seen in the following screenshot, considering you have similar data:

Results		Messages
	(No column name)	
1	20	
	(No column name)	
1	0	
	(No column name)	
1	CustomerID not found.	
	(No column name)	
1	20	

Figure 13.12: Output of the query with NOT EXISTS function

Programmability objects

When we talk about a program, the first thing that comes to our mind is the number of lines of codes, and *how to execute it?* Are we going to execute all these lines of code again and again, or *is there any shortcut?*

Relate these questions from T-SQL perspective, and suppose you've a T-SQL program with thousands of lines of code. This program is required to be executed quite frequently. The first thing you may think is to save it in a .sql file, and open it in SSMS, and run it whenever needed.

But *what if it is required to be triggered from a remote application such as a web, or a windows application?* Obviously, you can call all these lines of code individually from the remote application, but *does it really make sense to repeat the same lines of code, everywhere it is required to be executed?* If you ask me, then it doesn't make sense.

SQL Server offers various programmability objects, each for a specific purpose. Let us understand each of them in detail.

Views

The view can only contain a single **SELECT** statement. When we say single **SELECT** statement, it can be a complex **SELECT** query with multiple CTE's, joins, and subqueries, and so on. It can also have multiple **SELECT** statements combined to form a single **SELECT** statement using and **EXCEPT** operators.

So, if you have a **SELECT** query that is often used, it is advised to embed it in the form of a view. View as the name suggest is just the view being returned from the embedded query. You cannot perform and **DELETE** operations on view.

You can use a view, scalar user-defined function, and table-valued user-defined function inside a view. But you cannot invoke a stored procedure inside a view. Nesting of view is also possible. It means you can call the same view inside the body of the view. But there is a nesting limit of When this limit is reached, SQL Server will throw an error.

Nesting means you used the same view, function, or procedure inside the body. When you'll invoke it. Let says it's **N1** will again invoke itself and become **N2** will again invoke itself and become It will go on until it reaches The moment it will happen, an error will be thrown. Remember all these invocations from **N1** to **N32** are of the same programmability object such as view, function, stored procedure, or trigger.

You can run or execute the view with the help of the **SELECT** command, the same way does for the tables.

Syntax to create a

```
[< schema_name name_of_the_view > ]
```

AS

Example to create a view. The following view will return all the rows of **Customer** table:

```
[dbo].[vw_Customer]
```

AS

Syntax to alter or modify a

```
[< schema_name name_of_the_view > ]
```

AS

Example to alter/modify a view:

```
[dbo].[vw_Customer]
```

AS

Syntax to invoke or execute a

[< schema_name name_of_the_view >]

Example to invoke or execute a view:

[dbo].[vw_Customer]

Since there is no filter in the **SELECT** statement embedded in the view, it will return all the rows from the **Customer** table, as can be seen in the following screenshot:

	CustomerID	CustomerName	CustomerAddress	CustomerMobile
1	1	Customer 1	Customer 1 Address	1111111111
2	2	Customer 2	Customer 2 Address	2222222222
3	3	Customer 3	Customer 3 Address	3333333333
4	4	Customer 4	Customer 4 Address	4444444444
5	5	Customer 5	Customer 5 Address	5555555555
6	6	Customer 6	Customer 6 Address	6666666666
7	7	Customer 7	Customer 7 Address	7777777777
8	8	Customer 8	Customer 8 Address	8888888888
9	9	Customer 9	Customer 9 Address	9999999999
10	10	Customer 10	Customer 9 Address	1010101010
11	11	Customer 11	Customer 11 Address	1111111111
12	12	Customer 12	Customer 12 Address	1212121212
13	13	Customer 13	Customer 13 Address	1313131313
14	14	Customer 14	Customer 14 Address	1414141414
15	15	Customer 15	Customer 15 Address	1515151515
16	16	Customer 16	Customer 16 Address	1616161616
17	17	Customer 17	Customer 17 Address	1717171717
18	18	Customer 18	Customer 18 Address	1818181818
19	19	Customer 19	Customer 19 Address	1919191919
20	20	Customer 20	Customer 20 Address	2020202020

Figure 13.13: Output of the view

Syntax to drop a

DROP VIEW [< schema_name name_of_the_view >]

Example to execute a view:

DROP VIEW [dbo].[vw_Customer]

User-defined functions

Functions always return a value. That's the basic thing taught in every programming language. *So does in T-SQL too!* User-defined functions as the name implies can be created by us and should mandatorily return a value.

Similar to view, the function can only have a **SELECT** statement(s). It means you can only perform read not write. You cannot perform and **DELETE** on physical and temporary tables inside a user-defined function. Although, you can declare a table variable, and perform and **DELETE** operations on it.

You cannot create use DDL statements such as and **DROP** inside a user-defined function. Also, you cannot create a temporary table inside a user-defined function.

Functions have a difference as compared to views. You can have an argument in the functions, which you can use to filter your data within functions. Whereas, you cannot have arguments in view, as a result of which you cannot dynamically filter data within view. Although, you can have static filters in the **SELECT** query embedded in the view by the way of static values.

User-defined functions come in two different flavors:

The scalar user-defined function.

Table-valued user-defined function.

Let us understand each of these types of user-defined functions.

Scalar user-defined functions

The scalar user-defined function returns a scalar value of type as defined in the return type. You need to specify what type of value the function will return.

You can use a view, scalar user-defined function, and table-valued user-defined function inside a scalar user-defined function. But you cannot invoke a stored procedure inside a scalar user-defined function. Nesting of scalar user-defined function is also possible. It means you can call the same scalar user-defined function inside the body of the function. But there is a nesting limit of 10. When this limit is reached, SQL Server will throw an error.

Syntax to create a scalar user-defined

```
[< schema_name name_of_the_function >]
(
  @Param_1
  , @Param_2
  , ...
  , ...
  , @Param_N
)
RETURNS
AS
```

BEGIN

Do your stuff here

RETURN

END

Example to create a scalar user-defined function:

The following scalar function will return the **CustomerName** against the supplied **CustomerID** from the **Customer** table:

```
(  
@CustomerID    INT  
)  
AS  
BEGIN  
DECLARE  
  
SELECT @CustomerName = [CustomerName]  
FROM Customer  
WHERE [CustomerID] =  
  
RETURN  
END
```

Syntax to alter or modify a scalar user-defined

```
[< schema_name name_of_the_function >]  
(  
@Param_1
```

```
, @Param_2  
, ...  
, ...  
, @Param_N  
)
```

```
RETURNS
```

```
AS
```

```
BEGIN
```

```
Do your stuff here
```

```
RETURN
```

```
END
```

Example to alter or modify a scalar user-defined function:

```
(  
@CustomerID INT  
)
```

```
AS
```

```
BEGIN
```

```
DECLARE
```

```
SELECT @CustomerName = [CustomerName]
```

```
FROM Customer
```

```
WHERE [CustomerID] =
```

```
RETURN
```

```
END
```

Syntax to invoke or execute a scalar user-defined

```
SELECT [< schema_name name_of_the_function >] @Param_2,  
...,
```

Example to invoke or execute a scalar user-defined function:

```
SELECT
```

If you've data similar to what I've, you'll get the output similar to as can be seen in the following screenshot:

	(No column name)
1	Customer 1

Figure 13.14: *Output of the scalar user-defined function*

Syntax to drop a scalar user-defined

```
DROP FUNCTION
```

Example to drop a scalar user-defined function:

```
DROP FUNCTION
```

Table-valued user-defined functions

The table-valued user-defined function always returns a table. There are two types of table-valued user-defined function:

Inline table-valued function

Multiline table-valued function

You can use a view, scalar user-defined function, and table-valued user-defined function inside a table-valued user-defined function, but you cannot invoke a stored procedure inside a table-valued user-defined function. Nesting of table-valued user-defined function is also possible. It means you can call the same table-valued user-defined function inside the body of the function. But there is a nesting limit of 10. When this limit is reached, SQL Server will throw an error.

Inline table-valued functions

The inline table-valued user-defined function returns a table. Here, you are not required to specify the structure of the table such as column name and datatype of the columns to be returned as a part of the function output.

You just need to specify the return type as a table in the function definition. The function body should include the first statement as the return command. The return command should have a **SELECT** statements. The **SELECT** statement specified can have multiple **SELECT** statement combined using and **INTERCEPT** operators. You can have a simple **SELECT** statement, and also complex ones with multiple joins, subqueries, and so on.

The function cannot have any other command or statement, except a **SELECT** statement. For example, you cannot create a variable. You cannot use a **SET** command, and so on. You cannot also use **BEGIN ... END** blocks in the inline table-valued function.

Syntax to create an inline table-valued

```
[< schema_name name_of_the_function >]
(
  @Param_1
, @Param_2
, ...
, ...
, @Param_N
)
AS
RETURN
(
  Write your select statement here.
);
```

Example to create an inline table-valued function:

The following function will accept the **CustomerID** as the argument and return the table that will include the **Customer** attributes such as

```
(
@CustomerID    INT
)

AS
RETURN
(
SELECT [CustomerID]
, [CustomerName]
, [CustomerAddress]
, [CustomerMobile]
FROM Customer
WHERE [CustomerID] = @CustomerID
);
```

Syntax to alter or modify an inline table-valued

```
[< schema_name name_of_the_function >]
(
@Param_1
, @Param_2
, ...
, ...
```



```

, @Param_N
)
AS
RETURN
(
Write your select statement here.
);

```

Example to alter or modify an inline table-valued function:

```

(
@CustomerID INT
)
AS
RETURN
(
SELECT [CustomerID]
, [CustomerName]
, [CustomerAddress]
, [CustomerMobile]
FROM Customer
WHERE [CustomerID] = @CustomerID
);

```

Syntax to invoke or execute an inline table-valued

[< schema_name name_of_the_function >] @Param_2, ...,

Example to invoke or execute an inline table-valued function:

If you've data similar to what I've, you'll get the output similar to as can be seen in the following screenshot:

	CustomerID	CustomerName	CustomerAddress	CustomerMobile
1	1	Customer 1	Customer 1 Address	111111111

Figure 13.15: Output of the inline table-valued user-defined function

Syntax to drop an inline table-valued

DROP FUNCTION

Example to drop an inline table-valued function:

DROP FUNCTION

Multiline table-valued functions

The multiline table-valued user-defined function also returns a table in the structure defined in function definition. Here, you are required to specify the structure of the table such as column name and datatype of the columns to be returned as part of the function output.

The function has two sections:

Function definition

Function body

Function definition contains the name of the function, and the structure of the table output to be returned. The return type is generally a table variable. You fill the data in this table variable in the function body, and return the table variable at the end of the function body.

You cannot perform write operations on view and functions. Hence, here as well you cannot run any DDL or DML statement to modify the structure of the database of any physical or temporary tables. Although, you can declare variables, including table variables. You can run the **INSERT** and **DELETE** statements on the table variable. You can have multiple statements in multiline table-valued function. You can also use the **SET** command here.

The function body of the multiline table-valued function should start with **BEGIN** and end with **END** commands of the **BEGIN ... END** block. The last statement immediately before the **END** command should be the **RETURN** command:

Syntax to create a multiline table-valued

[< schema_name name_of_the_function >]

```
(  
  @Param_1  
  , @Param_2  
  , ...  
  , ...  
  , @Param_N  
)  
RETURNS TABLE  
(  
  Col_1  
  , Col_2  
  , ...  
  , ...  
  , Col_N  
)  
AS  
BEGIN  
Write logic to fill the table variable specified in the function  
definition  
END
```

Example to create a multiline table-valued function:

The following function will accept the **CustomerID** as the argument and return the table that will include the **Customer** attributes such as and

```
(  
  @CustomerID    INT  
)
```

```

RETURNS @Customer TABLE
(
, [CustomerName]
, [CustomerAddress]
, [CustomerMobile]
)
AS
BEGIN
@Customer
(
[CustomerID]
, [CustomerName]
, [CustomerAddress]
, [CustomerMobile]
)
SELECT [CustomerID]
, [CustomerName]

, [CustomerAddress]
, [CustomerMobile]
FROM Customer
WHERE [CustomerID] = @CustomerID

END

```

Syntax to alter or modify a multiline table-valued

```

[< schema_name name_of_the_function >]
(
@Param_1
, @Param_2

```

```
, ...  
, ...  
, @Param_N  
)  
RETURNS TABLE  
(  
Col_1  
, Col_2  
, ...  
, ...  
, Col_N  
)  
AS  
BEGIN
```

Write logic to fill the table variable specified in the function definition

```
END
```

Example to alter or modify a multiline table-valued function:

```
(  
@CustomerID INT  
)  
RETURNS @Customer TABLE  
(  
, [CustomerName]  
, [CustomerAddress]  
, [CustomerMobile]  
)  
AS
```

```

BEGIN
@Customer
(
[CustomerID]
, [CustomerName]
, [CustomerAddress]
, [CustomerMobile]
)
SELECT [CustomerID]
, [CustomerName]
, [CustomerAddress]
, [CustomerMobile]

FROM Customer
WHERE [CustomerID] = @CustomerID

END

```

Syntax to invoke or execute a multiline table-valued

[< schema_name name_of_the_function >] @Param_2, ...,

Example to invoke or execute a multiline table-valued function:

If you've data similar to what I've, you'll get the output similar to as can be seen in the following screenshot:

	CustomerID	CustomerName	CustomerAddress	CustomerMobile
1	1	Customer 1	Customer 1 Address	1111111111

Figure 13.16: *Output of the multiline table-valued user-defined function*

Syntax to drop a multiline table-valued

DROP FUNCTION

Example to drop a multiline table-valued function:

DROP FUNCTION

Stored procedures

Now it comes to the most interesting part of the SQL Server. One thing you would often hear people are talking about SQL Server is the stored procedures or simply procedures. Both of these names are used interchangeably.

Stored Procedures are superb! You can do almost anything inside the stored procedure body. You can run all the DDL and DML commands. You can do everything here that you can and cannot do with other programmability objects. Refer to the following list:

You can run **DROP** commands.

You can execute **REVOKE** commands.

You can perform and **DELETE** on the physical tables.

You can create a temporary table.

You can declare variables and table variables.

You can assign values to the variables.

You can use **SET** commands.

You can use views, scalar functions, and table-valued functions (both inline as well as multiline) inside a procedure.

Nesting of the procedure is also possible. But there is a limit of It means you can invoke the same procedure inside the body of the procedure. But if the recursion reaches to the limit of up to 32 levels, an error will be thrown by the SQL Server.

You can return a scalar value or a table. You can even simply choose not to return anything.

You have the flexibility to define the output parameters in the stored procedure definition. These output parameters can be manipulated in the stored procedure body, and the final value can be returned as the output parameter of the stored procedure.

You can return both – the output parameters and the procedure output, at the same time.

You can include the transaction management and error handling inside a procedure.

You can cover almost the entire T-SQL within a stored procedure.

They have few limitations though such as, you cannot use **GO** command inside a stored procedure body, and so on. Technically, you do not need a **GO** command inside a procedure. *Isn't it?*

Stored procedures with output parameters

Output parameter, *isn't it a new topic we discussed? Indeed!*

Output parameters are also an argument to a procedure, but they are not meant for supplying the input to the procedure. Instead to return the output from the procedures. Let us understand *how to define and retrieve it?*

Defining output parameter is optional and is no rocket science. Output parameters are defined the very same way the input parameters are defined. It's just you need to specify the **OUTPUT** keyword against each such output parameter.

Any parameter can be used as the variable to hold the temporary values. Generally, the input parameters are not changed, but output parameters do. Because that is why they are used to return the value. Unless you'll assign a value to the output parameter, *how will it return the value?* If the value is not assigned and returned using the output parameter then *what's the point of using it?*

In order to retrieve the values of the output parameters, you need to declare the corresponding equal number of variables of the same datatypes as the output parameters. You need to assign these variables to the respective output parameters

during procedure invocation. You also need to explicitly specify the **OUTPUT** keyword against each of such output parameters.

After invocation of the procedures, the variables assigned to the output parameters will hold the respective values of the output parameters. As we already discussed, output parameters are optional. You may or may not have them in your procedure. Let us understand both of these cases of procedures – with and without output parameters.

Stored procedures with the default parameters

You can also have default values to the parameters in the procedure definition. It is optional. When you do so, even if you'll not assign any value to such parameters during invocation, the procedure will run successfully. If no value is assigned then during invocation, the default value will be used by the T-SQL code embedded within the procedure body. But if the value is supplied to a parameter, even though a default value is assigned to it, the default value will be overridden with the value supplied.

All the parameters without a default value should be supplied during invocation, else an error will be thrown.

Getting hands dirty with stored procedures

Let us now learn the various syntax and examples to create, alter, drop, and invoke the stored procedure.

Syntax to create a stored procedure is as

The following procedure is written to select the records from the **Customer** table for the supplied It also returns the count of records returned in the result-set as an output parameter. But you can do anything. You can perform a combination of and You can combine multiple DDL and DML operations, and finally may or may not return some result-set:

```
[< schema_name name_of_the_procedure >]
(
  @Param_1    {= } {
, @Param_2    {= } {
, ...
, ...
, @Param_N    {= } {
)
AS
BEGIN
```

Do your stuff here.

Make sure to assign value to an output parameter (if used).

If you'll not assign the value, you'll not get it after procedure

invocation.

END

Example to create a stored procedure:

```
(  
  , @NumberOfCustomers  
)
```

AS

BEGIN

FROM Customer

WHERE [CustomerID] =

SET @NumberOfCustomers

END

Syntax to alter or modify a stored

[< schema_name name_of_the_procedure >]

```
(  
  @Param_1    {= } {  
  , @Param_2    {= } {  
  , ...  
  , ...  
  , @Param_N    {= } {  
)
```

AS

BEGIN

Do your stuff here.

Make sure to assign value to an output parameter (if used).
If you'll not assign the value, you'll not get it after procedure invocation.

END

Example to alter or modify a stored procedure:

```
(  
    , @NumberOfCustomers  
)
```

AS

BEGIN

FROM Customer

WHERE [CustomerID] =

SET @NumberOfCustomers

END

Syntax to invoke a stored

A stored procedure can be invoked using **EXECUTE** or **EXEC** command. Both are the same and can be used interchangeably.

If there are multiple output parameters then you need to declare an equal number of variables for each output parameter.

You need to assign a variable, and explicitly specify the **OUTPUT** keyword against each output parameter. After invocation you'll have the output parameter values, you can simply **SELECT** it, or use it the way you want.

```
DECLARE @  
EXECUTE [< schema_name name_of_the_procedure >]  
@Param_1    = {} {  
, @Param_2    = {} {  
, ...  
, ...  
, @Param_2    = {} {
```

Example to invoke a stored procedure:

The following is an example to invoke a procedure with output parameters:

```
DECLARE @NumberOfCustomers  
  
EXECUTE  
@CustomerID    = 1  
, @NumberOfCustomers = @NumberOfCustomers  
SELECT
```

Suppose the procedure doesn't have any output parameter. It just has one input parameter The invocation will look like as can be seen as follows:

EXECUTE

@CustomerID =

If you've data similar to what I've, you'll get the output similar to as can be seen in the following screenshot:

	CustomerID	CustomerName	CustomerAddress	CustomerMobile
1	1	Customer 1	Customer 1 Address	1111111111
	(No column name)			
1	1			

Figure 13.17: Output of the stored procedure

Syntax to drop a stored

DROP PROCEDURE [< schema_name name_of_the_procedure
>]

Example to drop a stored procedure:

DROP PROCEDURE

Triggers

Stored procedures are invoked by the user (by us), but triggers are automatically invoked upon certain DDL or DML actions, as specified in the trigger definition. You can almost do everything in triggers that you can do in procedures. It's just you cannot have the parameters in the triggers, and also you cannot invoke the triggers.

You can use a view, scalar user-defined function, and table-valued user-defined function and procedure inside a trigger. Nesting of trigger is also possible. But there is a nesting limit of 32. When this limit is reached, SQL Server will throw an error.

Triggers are typically of two types – **DDL** and **DML** triggers. There is one more type of trigger called **CLR**. A CLR trigger can be DDL or DML trigger. The following diagram depicts the various types of triggers in SQL Server:

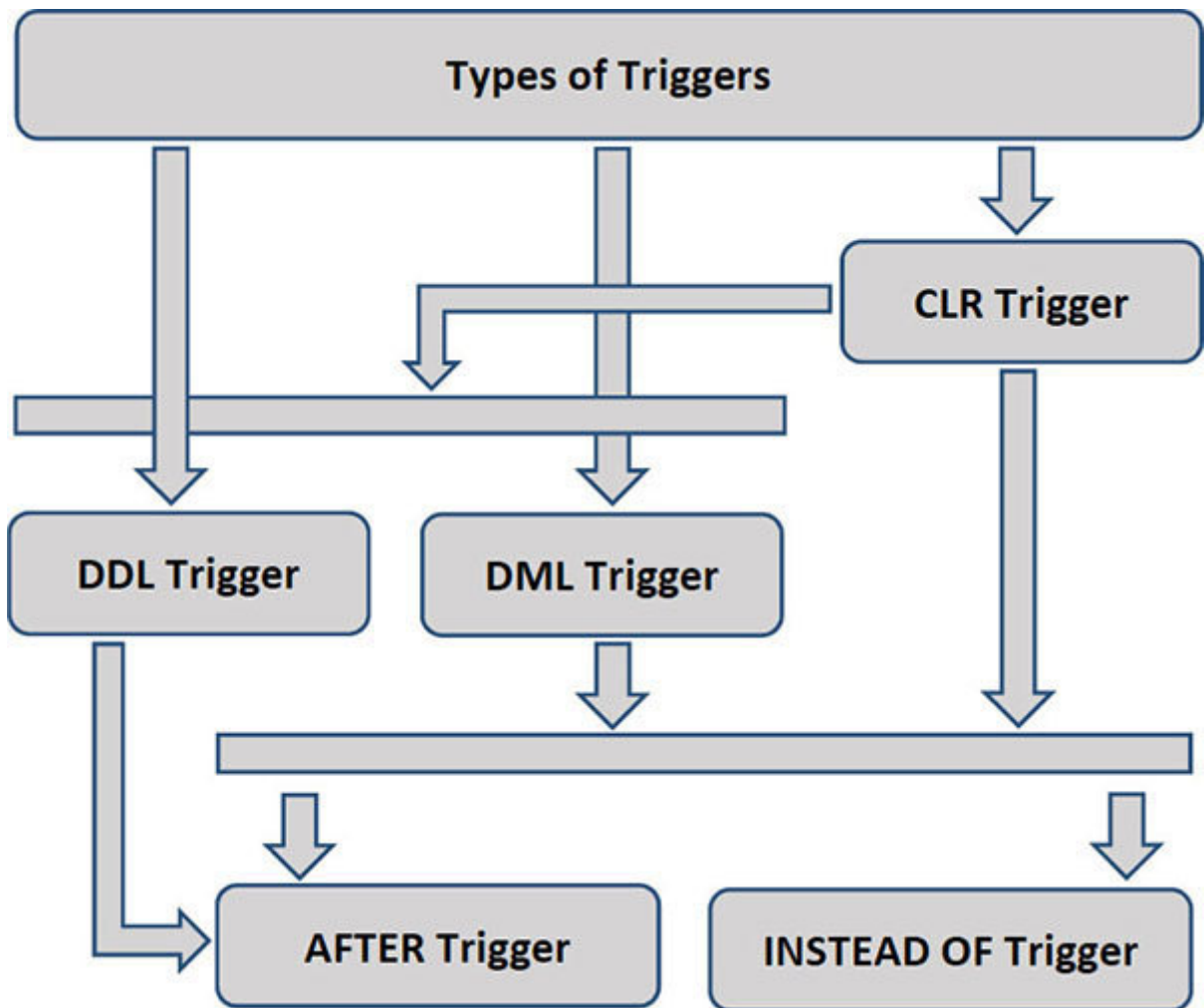


Figure 13.18: *Types of triggers*

As can be seen in the aforesaid diagram, there are two types of triggers – **AFTER** and **INSTEAD OF** at the leaf level in the trigger hierarchy. DML and CLR can be **AFTER** or **INSTEAD OF** trigger, but DDL can only be AFTER trigger. Let us understand at a high level *what do these triggers means?*

As we discussed earlier, a trigger is automatically invoked. But even to invoke itself automatically, it needs an event. AFTER and INSTEAD OF are a kind of events for the trigger.

Triggers are directly linked with the T-SQL query (DDL or DML):

AFTER trigger: These kinds of triggers are invoked only after all the operations specified in the triggering T-SQL query (DDL or DML) have been completed successfully. For example, if an AFTER trigger is created on the **Customer** table for DML operations such as **INSERT** and/or **DELETE** then the trigger will fire (or invoked) only after the triggering T-SQL query completes successfully.

INSTEAD OF trigger: These kinds of triggers are invoked by the DML statements, and the triggering T-SQL statements action is overridden by the action been specified within the trigger definition. These types of triggers are helpful if you want to bypass a particular DML action and instead do something else.

For example, you have two tables – **Customer_Major** and **Customer_Minor**. You also have one more table **Customer_Staging**. You actually want all the 18+ customers to be moved to **Customer_Major** table, and others in the **Customer_Minor** table. You can create an **INSTEAD OF** trigger on **Customer_Staging** table for **INSERT** that will have the required T-SQL code to route the data to the respective tables. Each time when a row will be inserted in the **Customer_Staging** table, the trigger will fire and route the data accordingly to the **Customer_Major** and **Customer_Minor** tables. No row will be inserted in the **Customer_Staging** table, even though the triggering **INSERT** statement will be successful.

DDL triggers

DDL triggers are created on the database and are defined to trigger upon certain or combination of DDL events such as and You can have an individual trigger for each DDL event or a single trigger for all the DDL events. DDL triggers are useful in tracking the database schema changes. With the help of DDL triggers, you can track who changed what, and when in the database.

Read more about DDL triggers at the following URL of Microsoft official documentation:

<https://docs.microsoft.com/en-us/sql/relational-databases/triggers/ddl-triggers?view=sql-server-ver15>

DML triggers

DML triggers are created on the tables and defined to trigger upon certain or combination of DML events such as and You can have individual trigger for each DML event, or a single trigger for all the DML events.

We talked about the magic tables – inserted and deleted tables in [Chapter 4](#). They are also called **virtual**. The inserted and deleted tables can be accessed through the DML triggers. These tables can be referred in the trigger to know what has been inserted, deleted, or modified by the triggering DML statement, and implement the action accordingly.

For example, if there is a table **Customer** which has three columns – and The respective inserted and deleted table will also have three columns similar to the **Customer** table. The rows in the inserted and deleted table will be the same as what has been inserted, deleted, or updated through the triggering DML statement. If one row is inserted into the **Customer** table then one row will be available in the inserted table, and you can actually access the data being inserted from the trigger.

AFTER (DML) trigger can be only created on the tables.
Whereas INSTEAD OF (DML) trigger can be created on the tables as well as views.

Let us see an example of DML trigger. We'll replicate the example in T-SQL for the scenario discussed in the aforesaid topic **INSTEAD OF** trigger.

The following is the T-SQL script for the three tables – and

```
CREATE TABLE Customer_Staging
(
    CustomerID INT
    ,
    ,
)
```

```
CREATE TABLE Customer_Major
(
    CustomerID INT
    ,
    ,
)
```

```
CREATE TABLE Customer_Minor
(
    CustomerID INT
    ,
    ,
)
```

The following is the T-SQL script for the **INSTEAD OF** trigger to route the data to **Customer_Minor** and **Customer_Major** tables:

```
CREATE TRIGGER TR_Customer_Staging
ON Customer_Staging
INSTEAD OF INSERT
AS
BEGIN
SET NOCOUNT ON
```

```
INSERT INTO Customer_Major
(
CustomerID
, CustomerName
, DOB

)
SELECT CustomerID
, CustomerName
, DOB
FROM inserted
WHERE > 18
```

```
INSERT INTO Customer_Minor
(
CustomerID
, CustomerName
, DOB
)
SELECT CustomerID
```

```
, CustomerName  
, DOB  
FROM inserted  
WHERE <= 18  
END
```

We'll now add two rows to the **Customer_Staging** table. One of them is major (18+) and another is minor (< 18). According to the example we discussed, rows pertaining to the major clients should go to the **Customer_Major** table, and rows pertaining to the minor clients should go to the **Customer_Minor** table. No row should go to the **Customer_Staging** table.

The following T-SQL script includes an insert statement with two rows. Immediately after the **INSERT** statement, there is a **SELECT** statement on all the three tables:

```
INSERT INTO Customer_Staging  
  
VALUES  
,  
SELECT * FROM Customer_Staging  
SELECT * FROM Customer_Minor  
SELECT * FROM Customer_Major
```

As can be seen in the following screenshot, all the rows inserted in the **Customer_Staging** table are routed to the **Customer_Minor** and **Customer_Major** tables. **Customer_Staging** table does not hold any row:

	CustomerID	CustomerName	DOB
1	1	Shashank	2009-07-23
1	2	Santosh	1986-01-10

Figure 13.19: *Output of the trigger*

DML triggers are useful in auditing which means keeping track of data changes in a table, but it is not limited to auditing, you can write your own solution. You can do something else upon a DML action on a table.

Data auditing through trigger is a legacy way, and nowadays not preferred due to performance issues. SQL Server has introduced various features for maintaining the audit log/data change log. These features are **change change data** and **temporal**

Read more about DML triggers at the following URL of Microsoft official documentation:

<https://docs.microsoft.com/en-us/sql/relational-databases/triggers/dml-triggers?view=sql-server-ver15>

Conclusion

and **NULLIF** enable to deal with Nulls effectively. Dynamic SQL makes it possible to generate and execute the T-SQL dynamically. **EXISTS** function tests the existence of the record in the query as an argument to it.

Programmability objects enable to achieve the reusability in T-SQL. You can bundle set of T-SQL statements in a programmability object such as stored procedure, User-defined function, view, or trigger. Trigger is a programmability object similar to the stored procedure, user-defined function, and view. But you cannot execute it manually. As the name suggests they have triggered automatically upon a specific DDL or DML action.

Triggers are of two types – DDL and DML triggers. DML triggers are created on the table, for a specific DML action such as and/or You can also create a trigger for multiple actions together. Similarly, the DDL triggers are created at database level for a specific DDL action such as and/or

With this chapter, we can conclude that we have covered all the crucial aspects of T-SQL development. In the next chapter, we'll learn to deploy a SQL Server database and the T-SQL objects.

Points to remember

COALESCE can be used instead of but vice-versa is not possible.

If you will try to compare the **NULL** with equality or inequality or then you would be surprised to see it does not work, the way you want. **NULL** is a complex structure and is driven by various parameters (or, settings) such as **ANSI NULLS Enabled** and **ANSI NULL**. These are database-level settings, and by default both of them are. These can be modified at the database level, and at the individual query level too.

Views and inline table-valued user-defined functions can only contain a single **SELECT** statement. When we say single **SELECT** statement, it can be a complex **SELECT** query with multiple CTE's, joins and subqueries, and so on. It can also have multiple **SELECT** statements combined to form a single **SELECT** statement using and **EXCEPT** operators.

Views cannot have parameters.

User-defined functions are of two types – scalar and table-valued user-defined functions.

There are two types of table-valued user-defined functions – inline and multiline table-valued functions.

Views and user-defined functions cannot invoke stored procedures.

View and table-valued functions can be used in the joins.

Scalar functions can be used as the column in the **SELECT** statement in the column list, join conditions, and in the filter conditions.

Stored procedures can have optional output parameters. Output parameters are also an argument to a procedure, but they are not meant for supplying the input to the procedure. Instead to return the output from the procedures.

Stored procedures parameters can have optional default values. Such parameters are called default parameters. You can assign default values to the parameters in the procedure definition.

Triggers cannot be invoked manually. Triggers are automatically invoked upon certain actions.

There are two types of triggers: DDL and DML triggers.

DDL triggers are created on the database and defined to trigger upon certain or combinations of DDL events such as and You can have individual trigger for each DDL event or a single trigger for all the DDL events.

DDL triggers are useful in tracking the database schema changes.

DML triggers are created on the tables, and defined to trigger upon certain or combinations of DML events such as and

You can have an individual trigger for each DML event or a single trigger for all the DML events.

DML triggers are useful in auditing which means keeping track of data changes in a table. But it is not limited to auditing, you can write your own solution. You can do something else upon a DML action on a table.

DDL triggers can be only **AFTER** trigger, whereas DML triggers can be **AFTER** or **INSTEAD OF** trigger.

INSTEAD OF trigger can be created on tables and views.

Multiple choice questions

You've a procedure that inserts a record in the Product table. One record is inserted at a time from the procedure. The procedure can be invoked from the multiple session in parallel. You are told to modify the procedure to return the last inserted ProductID from the procedure as an output parameter. Which of the following function would you use?

IDENT_CURRENT

SCOPE_IDENTITY

@@IDENTITY

You've a procedure that has three parameters – A, B, and C. B has a default value assigned as 0. The procedure simply returns the values of the supplied parameters. During invocation you supplied A = 1, B = 2, and C = 3. What would be the output of the procedure?

1, 0, 2

1, 0, 0

1, 0, 3

1, 2, 3

Answers

B

D

Questions

What is the difference between views and functions?

When not to use view?

What cannot be done in a multiline table-valued user-defined function?

What is the primary difference between triggers and stored procedures?

Which type of trigger can be used to log the changes made to table structure?

Write a procedure that should **INSERT** fresh record, and modify the existing record in the **Customer** table. The procedure should have two output parameters – **IsValidationSuccess** and The procedure should validate if the parameter value supplied to the procedure contains any of the following DDL and DML commands – and If any of these commands are found in the parameter value, then the output parameter should return **IsValidationSuccess** output parameter as and the **ValidationMessage** output parameter as a message showing which commands were found. If no such commands

are found then return **IsValidationSuccess** output parameter as
and the **ValidationMessage** output parameter as

Key terms

ISNULL tests if the supplied expression is a null.

COALESCE function accepts *N* number of arguments and returns the first non-null value from the given expressions.

NULLIF function returns a **NULL** value if the expressions supplied are equal, else the value from the first arguments (expression) is returned. It accepts two arguments, and both are expressions.

IDENTITY is a column property. The column has to be of numeric datatype in order to use this property. If enabled, it will auto-generate the numeric value for the column.

@@IDENTITY returns the last inserted identity value, across all the tables, and across all the sessions.

IDENT_CURRENT returns the last inserted identity value of the supplied tables, across all the sessions. It accepts the table name as an argument.

SCOPE_IDENTITY returns the last inserted identity value of the previous insert statement, in a session.

SET NOCOUNT {ON | OFF} If you make it **ON** then the message showing the count of number of rows affected by the DML statements will not be shown, but if it is made **OFF** then the message will be shown. **SET NOCOUNT** is specific to a session (or,

IS operator is used to perform equality comparison with the **NULL** value.

IS NOT is used to perform inequality comparison with the **NULL** value.

EXISTS returns **1** if the supplied **SELECT** query returns any row or value, else **0** is returned.

NOT EXISTS returns **0** if the supplied **SELECT** query returns any row or value, else **1** is returned. **NOT EXISTS** is exactly the opposite of

CREATE VIEW command is used to create a view.

ALTER VIEW command is used to alter or modify a view.

DROP VIEW command is used to drop a view.

CREATE FUNCTION command is used to create a user-defined functions.

ALTER FUNCTION command is used to alter or modify a user-defined functions.

DROP FUNCTION command is used to drop a user-defined functions.

CREATE PROCEDURE command is used to create a stored procedure.

ALTER PROCEDURE command is used to alter or modify a stored procedure.

DROP PROCEDURE command is used to drop a stored procedure.

command is used to invoke a stored procedure.

SELECT command is used to invoke views and all kinds of user-defined functions.

CREATE TRIGGER command is used to create a trigger.

ALTER TRIGGER command is used to alter or modify a trigger.

DROP TRIGGER command is used to drop a trigger.

CHAPTER 14

Deployment

The end objective of any development is the delivery for testing, and finally deploying it on production. So far in the earlier chapters, we learnt to develop and write T-SQL queries and programmability objects. In this chapter, we'll talk about different ways of deployment of the developed T-SQL objects.

Structure

In this chapter, we will cover the following topics:

Deployment

Different methods of deployment

Generating and executing the script

Backup and restore

Objective

Deployment is one of the most important aspects of any software development process. Any miss-step in this process and you could have your both hands full of multiple disasters and escalations with a very minimal time on hand for remedy. The objective of this chapter is thus to show the different methods of deployment and also to show which method should ideally be used in which situations.

Deployment

The ultimate goal of developing a software program (including T-SQL) is to make it live. **Software Development Life Cycle** have various stages including development, testing, UAT, and production. Each of these stages is probably performed on different systems. The program is packaged in the form of release, and released for further environments such as QA, UAT, and production. The release is then deployed on the respective environments.

You develop the program in your development machine, package it, and send the release to the **Quality Assurance** team for testing. QA team performs testing, and if testing is successful without any error, the release is further moved to UAT for user acceptance testing. But if the QA team finds an error, it is reported back to the development team. The development team fixes the bugs and sends release to the QA team again for testing. The cycle goes on until the release is bug free.

Similarly, when UAT passed successfully without any error, the release is moved to the production. But if any bug is observed in UAT, it is reported to the development team. The development team fixes the bugs and sends release to the QA team for testing, and after successful QA testing release

is moved to UAT. The cycle goes on until the release is bug free.

Refer to the following diagram showing the different stages of the SDLC we talked about. This will help you understand how the SDLC iteration works:

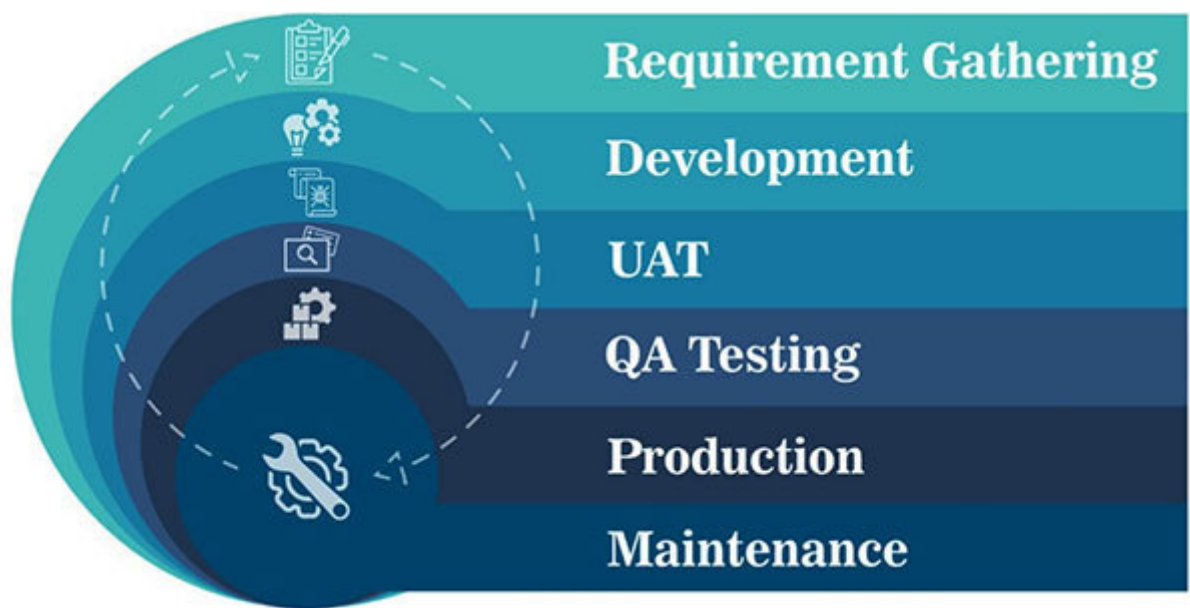


Figure 14.1: *Software Development Life Cycle (SDLC)*

The end objective of anything we build is to deliver it to the end customer. You can relate it to any real-world example. Something is built in one place and delivered to another place. The entity receiving the delivery then deploys by following the deployment instructions received therewith.

I do not wish to talk about SDLC on this topic, because it can't be covered in a topic or a chapter. It's a whole subject in itself. The purpose of talking about SDLC here was to

make you understand what the deployment is, and why it is needed.

Different methods of deployment

Deployment is associated with the releases. You can't deploy unless you'll get the release. *Can you deploy an OS update or an Android update, if it is not released by the provider?* The answer is In order to deploy a release, you must have the release.

Releases in the case of SQL Server are of the following two types:

Full: This is generally sent the very first time and includes the entire database schema including tables and all other programmability objects part of the database. The full release may also include seed data. Seed data are generally part of some master tables, which always remain the same. For example, countries are hardly added or deleted. The full release is helpful for the fresh database setups.

Incremental: This includes the specific objects such as tables, programmability objects, and seed data, which were added, deleted, or modified after the full release.

As we already discussed, deployment is tightly linked with the release. You will only deploy what you'll get. So, the deployment also is of two types – **Full** and Let us understand

the different methods of the release and deployment in SQL Server.

Generating and executing script

SQL Server is all about T-SQL script. Everything in SQL Server is created with T-SQL script. You can generate the T-SQL script for the individual object, and also for the entire database. You can also choose to generate the script only for schema, or data, or both. So, technically, you can generate the script for anything that is part of your SQL Server database.

Scripts become an important mode of release. This is a widely used solution to package the SQL Server objects in the form of release, and send it for the deployment. It is very simple. You developed your SQL Server database, generated the script, saved it in the form of **.sql** file, and sent it for the deployment. Deployment is very simple. Just open the **.sql** file and execute it on the target database. The scripts need to be executed on the target in the same sequence, they are released.

A separate **.sql** file can be used for each object, or everything can be embedded in a single **.sql** file. Whether you create a single **.sql** file, or a separate **.sql** file for each object, the sequence of the objects is very important. When you create anything, the parents come first, and the child next. But when you drop or delete anything, the child comes first, and the parent next. Always remember this.

SQL Server offers a built-in feature to generate the script. It has got a comprehensive GUI where you can choose your parameters, which are self-explanatory, and generate the script. Let us understand how to generate the script using **Generate Scripts wizard** of SSMS.

Generating Scripts

In order to open the Generate Script wizard:

You need to go to the SSMS, and right click on the database, go-to and then choose **Generate** A screen similar to as can be seen in the following screenshot will be populated:

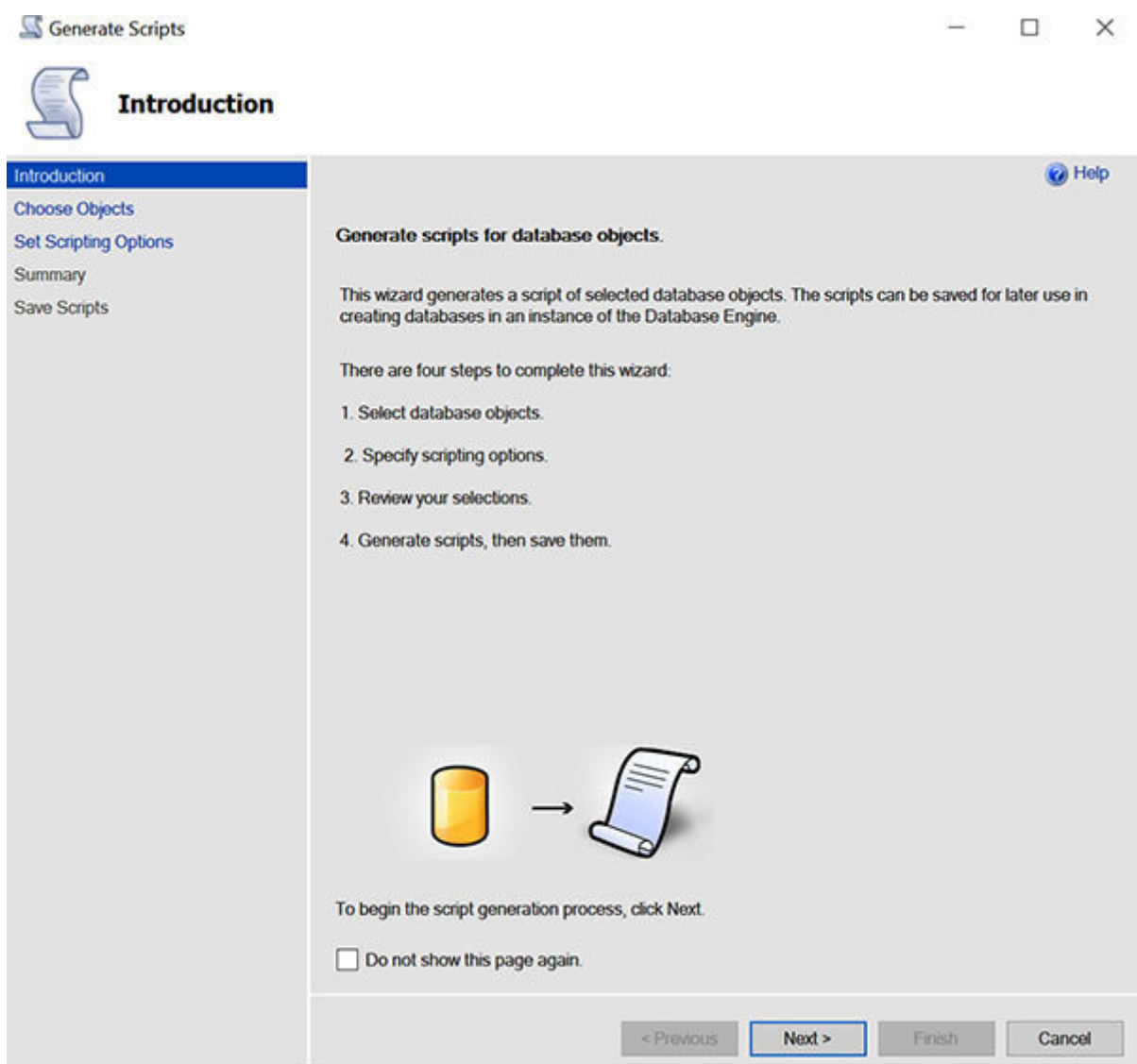


Figure 14.2: *Generate Scripts wizard - Introduction screen*

Click on the **Next >** button, and the next screen will look like as can be seen in the following screenshot. Here, you can choose whether you want to script the entire database or specific objects:

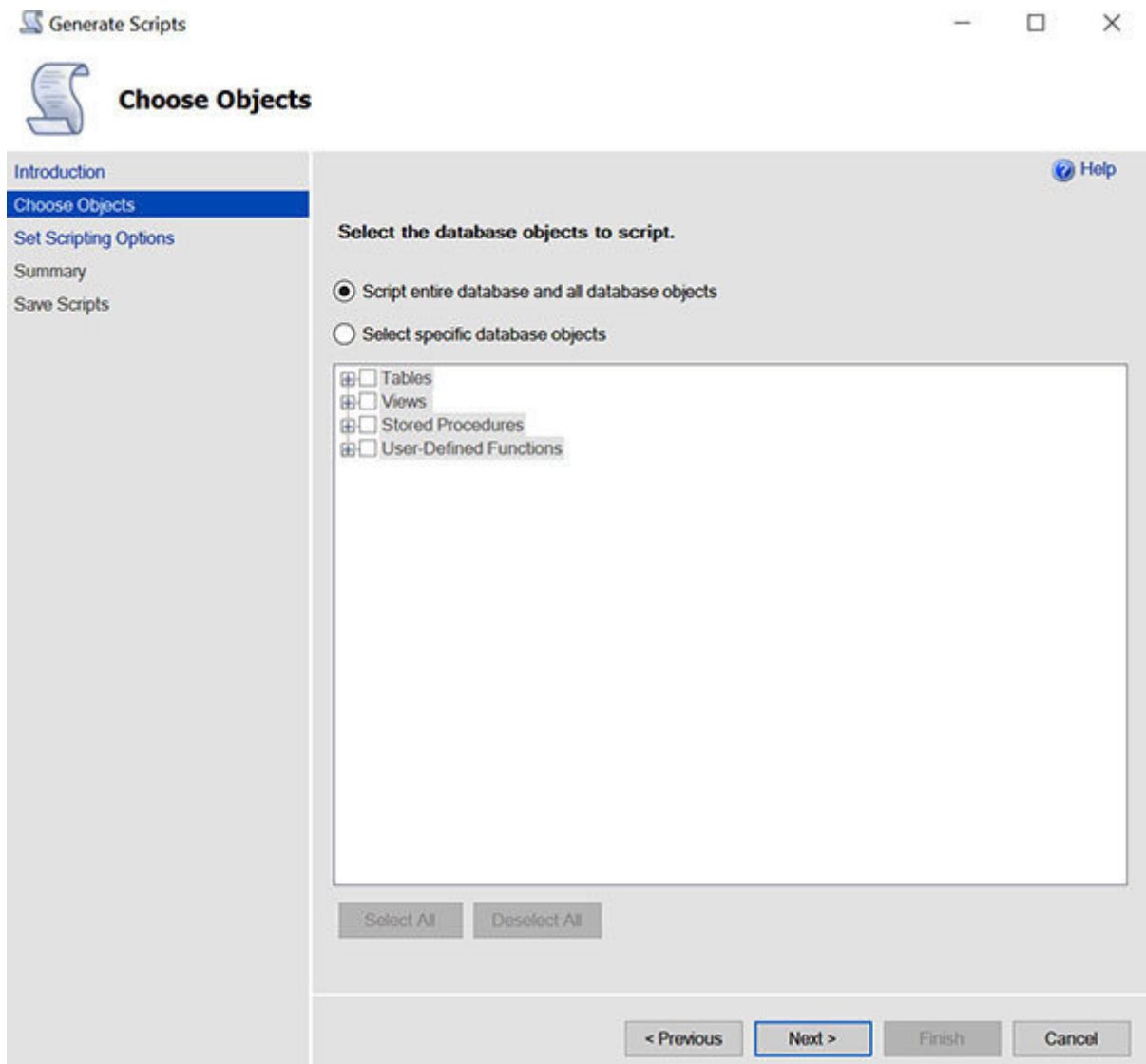


Figure 14.3: *Generate Scripts wizard – Choose Objects screen*

Once you've chosen the required option, again click on the **Next >** button, the screen similar to as can be seen in the following screenshot will be populated. Here, you can choose the desired option to save the generated scripts. There are four options, and each of them is self-explanatory.

Can you see the Advanced button in the following screenshot?
This is the most important part of the Generate Scripts utility:

The screenshot shows the 'Generate Scripts' utility window with the 'Set Scripting Options' tab selected. The left sidebar contains a navigation pane with the following items: Introduction, Choose Objects, Set Scripting Options (highlighted), Summary, and Save Scripts. The main area is titled 'Specify how scripts should be saved.' and contains the following options:

- ☐ Save as notebook (with an 'Advanced' button next to it)
- ☐ Save as script file
 - Files to generate: ☒ Single script file, ☐ One script file per object
 - File name: C:\Users\kulde\Documents\script.sql
 - ☒ Overwrite existing file
 - Save as: ☒ Unicode text, ☐ ANSI text
- ☐ Save to clipboard
- ☐ Open in new query window

At the bottom of the dialog are four buttons: '< Previous', 'Next >', 'Finish', and 'Cancel'.

Figure 14.4: *Generate Scripts wizard – Set Scripting Options screen*

Click on the **Advanced** button, and the screen similar to the following screenshot will be populated. It will contain multiple parameters. Due to limited screen size, all the parameters cannot be shown in the single screenshot. There are two types of options – **General** and **Table/View** Settings under the **General** option apply to all kinds of objects in the SQL Server. Whereas settings under **Table/View Options** apply only to the tables and views.

Following screenshot shows the parameter available in the **General** option of the **Advanced** button:

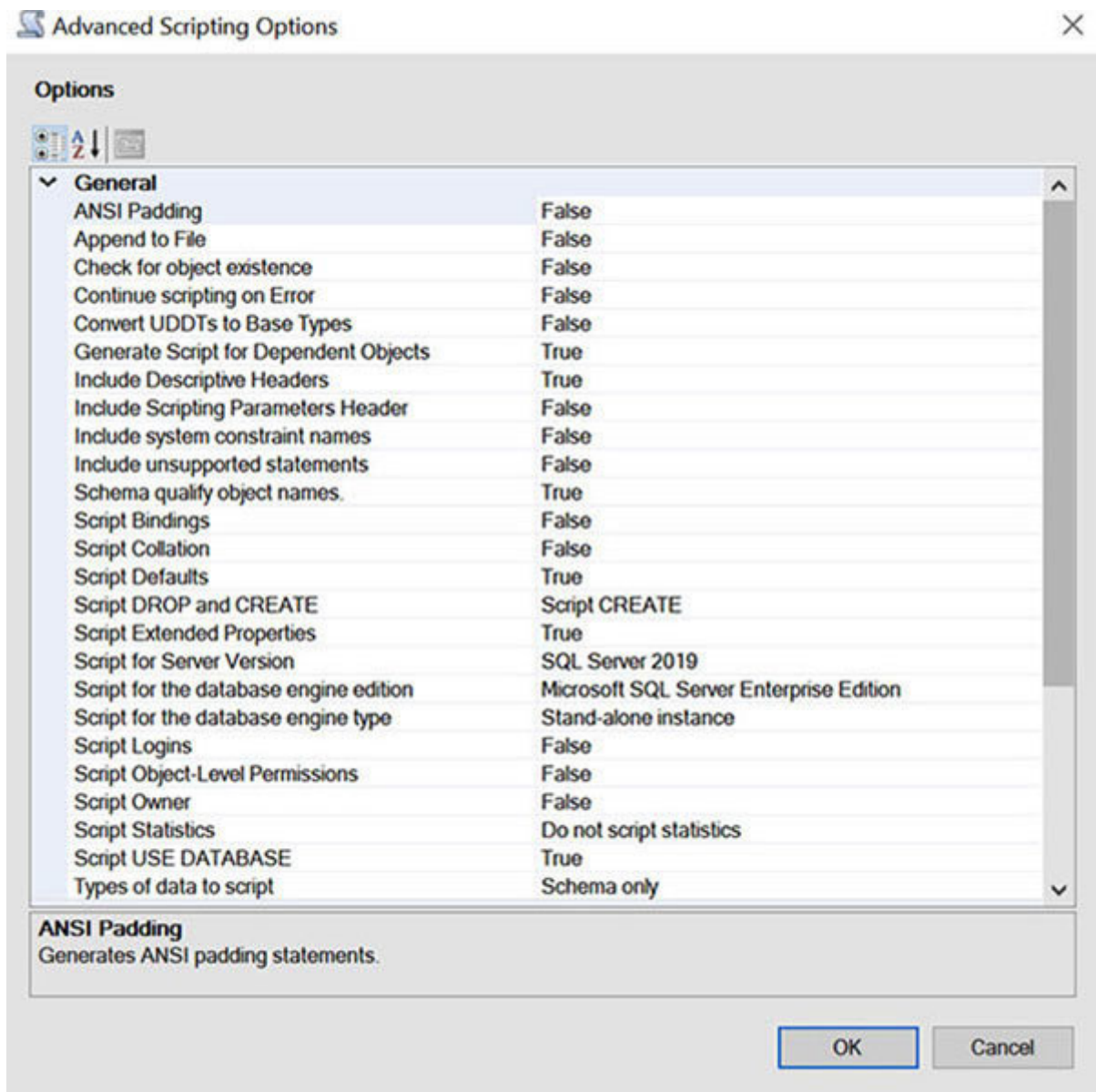


Figure 14.5: *Generate Scripts wizard – Advanced Scripting Options screen*

The following screenshot shows the parameters available in the **Table/View Options** of the **Advanced** button:

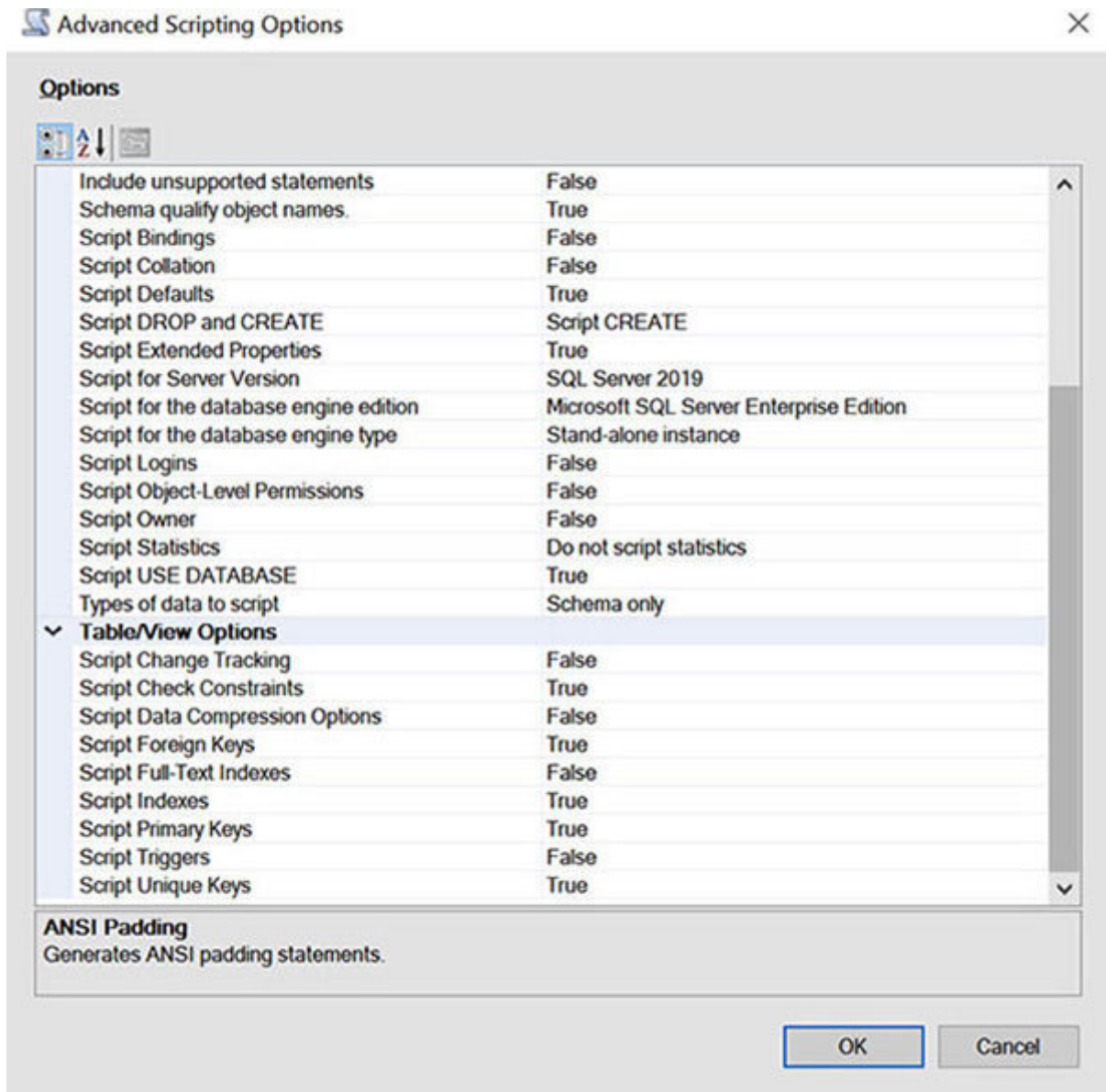


Figure 14.6: *Generate Scripts wizard – Advanced Scripting Options screen continued*

You can choose the desired options from the settings available under the **General** and **Table/View Options** from **Advanced Scripting** as you can see in [figures 14.5](#) and The resulting scripts will be generated accordingly.

Once you are done with the selection of the desired settings from **Advanced Scripting** click on the **OK** button. The **Advanced Scripting Options** screen will be closed and the original screen can be seen. You need to click on the **Next** button on this screen again.

A screen similar to as can be seen in the following screenshot will be populated. This screen shows all the options you've selected in the previous steps. You need to validate it. If everything looks good, click on the **Next** button again:

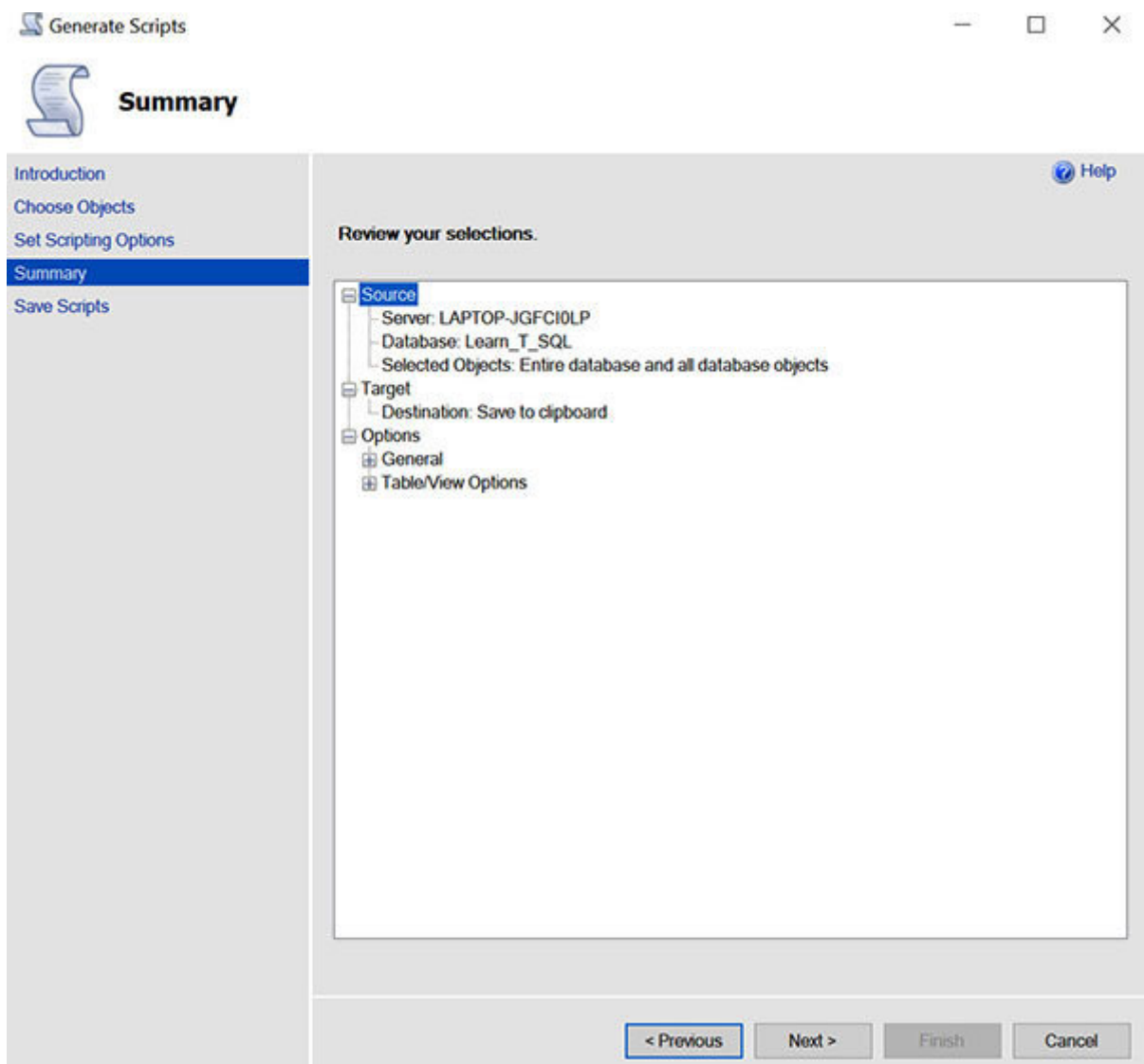


Figure 14.7: Generate Scripts wizard – Summary screen

Click on the **Next** button again, and script generation will start. A new screen similar to the following screenshot will be shown, which will show the progress of the script generation of the individual objects. Once the script generation of all the objects as chosen will be completed, the **Finish** button will be enabled. You need to click on it, and *you are all set!*

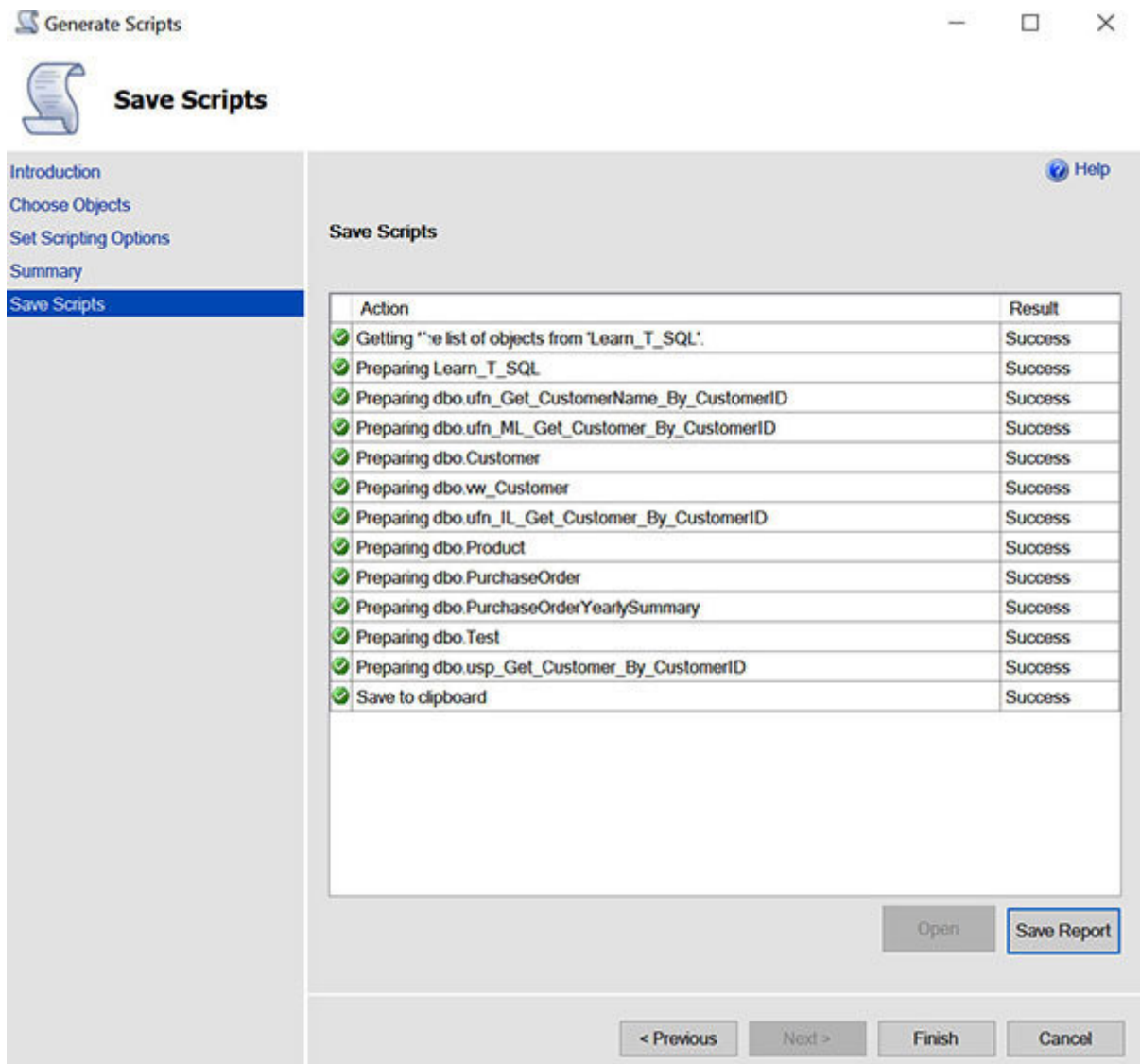


Figure 14.8: *Generate Scripts wizard – Save Scripts screen*

The script will be available as per the script saving option you selected in the earlier screen. If you chose to save it in the **.sql** file, go to the folder selected, and you can see the **.sql** file with the name you provided. If you chose to save the script in the clipboard, it will be available in the clipboard, and you can paste it anywhere you wish. Clipboard has a limited storage limit, so if the data is huge, chose to save it in the **.sql** file. If you chose to save the script in the new query window, then new query windows will be automatically opened with the generated scripts.

Backup and Restore

Backup and Restore is another method of deployment. But is to be used carefully. This method is hardly used. This can be used in the case of very first release. The backup can be generated from the development database and restored on further instances. If the target system already has the database, think twice before you restore.

Once you'll restore, you'll lose the existing contents of the database. When you restore a database, the earlier version of the database (both schema and data) will be completely replaced with the new backup copy as chosen during the restore. In order to restore a database, you need a backup.

It is to be noted, the *target* version should be the same or higher than that of the in order to restore a backup successfully.

Where

Source is the SQL Server instance on which the database backup is taken.

Target is the SQL Server instance on which the backup (taken on the source) would be restored.

Version is the SQL Server version.

If the source SQL Server version is higher than the target SQL Server versions, the restore will fail. For example, if the source version is 2016 and the target version is 2019, the restore shall be successful. It means the backup taken on SQL Server 2016 can be restored on the SQL Server 2019. Whereas, vice versa is not possible, and the restoration shall fail.

It is good to have the same version of SQL Server on both source and target.

Backup of SQL Server database generates a file with extensions mentioned as follows. You need to choose the desired backup file in order to restore it:

Full database backup: ***.bak**

Differential backup: ***.dif**

Transaction log backup: ***.trn**

Database backup is generally preferred in the production environment. You would not want to lose the production data, in case of any disaster such as system crash, flood,

earthquake, and so on. *Isn't it?* Backups can be extremely helpful for disaster recovery.

There are different backup strategies followed by different organizations. Backups are of three types – and **Transaction** log backups. A full backup includes the entire database. Differential's backup generates the backup from point of the last successful full or differentials backup. Log backup generates the backup of the transaction logs.

It is not wise to run the daily full backup. If you do so, think about the amount of disk space you would require. All organizations have a different backup strategies, but I've observed the following strategy used widely:

Weekly full backup.

Daily differential backups.

Transaction log backups in every few minutes.

Suppose weekly full back is scheduled every *Sunday* at 10 daily differential backup is scheduled daily at 10 transaction log backup is scheduled in every 10 *minutes*. If there is a disaster on *Thursday* at 11 then you'll have to restore the last successful full backup, then all the differential backups, then all the transaction log backups after the last successful differential backup. This chain is called the **backup**

So, when you've to recover from any disaster, you will have to restore the complete backup chain in the sequence. Full back-up is restored first, then the differentials, and finally the log backup, all in the sequence. It has to follow the FIFO order. The backup which was taken first has to be restored first.

Even though we have a backup option, still there is the possibility of data loss. Suppose the log backup is taken in every 10 minutes. Suppose the system crashed after 5 minutes of the last successful log backup. You'll lose the data of 5 minutes in this case. If you have a proper database backup chain and have chosen the correct recovery model, you would be also able to restore a database to a specific point called **Point in Time**. The specific point means a specific date and time.

Read more on High Availability (HA) and Disaster Recovery (DR) at the following URL of the Microsoft official documentation:

<https://docs.microsoft.com/en-us/sql/database-engine/sql-server-business-continuity-dr?view=sql-server-ver15>

Database Backup and Restore are independent processes and can be done through T-SQL script or using GUI (SSMS wizard).

You can read more about SQL Server Database Backup and Restore, including the steps to do it, at the following URL of the Microsoft official documentation:

<https://docs.microsoft.com/en-us/sql/relational-databases/backup-restore/back-up-and-restore-of-sql-server-databases?view=sql-server-ver15>

Conclusion

Deployment is directly linked with the release. You can only deploy what you get as a part of the release.

SQL Server Databases and their objects can be released in two ways – full and incremental. The full release includes the entire database, whereas incremental release includes only specific changes made after full release.

Backup and restore can be used for the very first release (full release), but due caution is to be taken while doing so. Restoring a backup replaces the entire existing database with new contents. So, all your existing contents will be lost.

Backup is of three types – Full, Differential, and Transaction log backup. Collectively they form backup chain. You need to restore the backup chain in FIFO order to restore a database. With the correct database backup strategies, Point in Time Restore is also possible.

The SQL Server Database release is widely done using scripts. Generate Scripts wizard of the SSMS can be used for generating the script. It is a comprehensive solution to generate the script out of your SQL Server Database. You have a better control of your release through this approach. The deployment of such releases is quite simple, and the

person deploying the scripts has a great control with it. Unless the script itself is incorrect. In order to deploy released scripts, just open the individual **.sql** file, and run it in the sequence as instructed.

Points to remember

Point in Time Recovery is a recovery option in which you can restore a database to a specific point (or, a specific date and time).

There are three types of backups – full, differential, and transaction log backups.

The SQL Server version of both source and target has to be the same, in order to restore a database backup.

Database restore replaces the existing database with contents of the backup copy. All its existing contents are lost.

Release and deployment through scripts is an ideal deployment solution.

Multiple choice questions

You have developed a fresh database on a SQL Server version SQL Server 2016. You want to setup it for QA testing. The SQL Server version of the QA machine is 2012. You took the database backup from the development machine. Will you be able to restore it on the QA machine?

Yes

No

Can you generate individual scripts for the individual objects?

Yes

No

Can you generate the script for the entire database?

Yes

No

Answers

B

A

A

Questions

You have developed a fresh database. You want to setup it for QA testing. The database is not yet created on the QA machine. This is the first time it will be released. How you will do it and why?

When do you think Point in Time Recovery can be useful, and how you can achieve it?

Key terms

Backup: Refers to backup of the database.

Restore: Refers to the restoration of the database backup file.

Symbols

@@ERROR [317](#)

example

@@IDENTITY [352](#)

@@TRANCOUNT [324](#)

A

ABS () function

using [167](#)

ACID [9](#)

Atomicity (A) [10](#)

Consistency (C) [10](#)

Durability (D) [10](#)

Isolation (I) [10](#)

transaction [10](#)

AFTER trigger [382](#)

aggregate functions [134](#)

AVG () [149](#)

COUNT () [146](#)

GROUP BY [135](#)

HAVING [135](#)

MAX () [151](#)

MIN () [150](#)

SUM ()

syntax [141](#)

using, with INSERT statement

aggregate windowing functions [220](#)

example

T-SQL example [226](#)

Algebrizer [25](#)

ALTER TABLE command

American National Standards Institute [11](#)

analytical functions [231](#)

syntax [232](#)

T-SQL example [233](#)

APPLY clause [209](#)

CROSS APPLY

OUTER APPLY

using [210](#)

approximate floating point numeric data types [44](#)

atomicity [313](#)

AVG () function

using [149](#)

B

Backup and Restore [402](#)

backup chain [402](#)

backups

differential [402](#)

full [402](#)

transaction [402](#)

BEGIN ... END keyword [274](#)

binary data type [42](#)

bookmark lookup [86](#)

key lookup [86](#)

RID lookup [86](#)

Boolean data type [45](#)

BREAK keyword

using [291](#)

built-in functions

aggregate functions [134](#)

date functions [171](#)

numeric functions [166](#)

string functions [154](#)

windowing functions [224](#)

C

cardinality [83](#)

good cardinality [83](#)

poor cardinality [83](#)

cascading [71](#)

CASE statement

using

CAST and CONVERT functions

arguments [336](#)

examples [338](#)

CAST keyword

using [290](#)

CEILING () function

using [169](#)

CHARINDEX () function

using [164](#)

CHECK constraint [68](#)
clustered columnstore index [92](#)
syntax [92](#)
clustered index
intermediate nodes [81](#)
leaf nodes [81](#)

qualities [83](#)
root node [81](#)
syntax [84](#)
clustered index key
COALESCE function [348](#)
columnstore indexes [92](#)
clustered columnstore index [92](#)
filtered columnstore index [94](#)
non-clustered columnstore index [93](#)
Common Table Expression (CTE)
T-SQL example [309](#)
Composite Primary Key [56](#)
constraints [8](#)
CONTINUE keyword
using [292](#)
conversion [332](#)
COUNT () function
COUNT (*) [147](#)
COUNT (name>) [146](#)
COUNT (DISTINCT name>) [148](#)
using [146](#)
covering index [88](#)
syntax [89](#)
CREATE DATABASE command
CREATE SCHEMA command [40](#)

CREATE TABLE command [46](#)

CROSS APPLY

using

cross-database data access

cross-server data access [342](#)

D

data

semi-structured data [237](#)

structured data [237](#)

unstructured data [237](#)

database [3](#)

column [4](#)

rows [4](#)

table [3](#)

Database Engine

binding [25](#)

parsing [25](#)

query execution

query optimization [25](#)

Database Management Systems (DBMSs) [4](#)

Data Control Language (DCL) [12](#)

data conversion [333](#)

CAST function [335](#)

CONVERT function [335](#)

explicit conversion [334](#)

implicit conversion [334](#)

PARSE function [340](#)

TRY_CAST [341](#)

TRY_CONVERT [341](#)

TRY_PARSE [341](#)

Data Definition Language (DDL) [11](#)

data filtering [106](#)

Data Manipulation Language (DML) [12](#)

data pages [79](#)

data types [40](#)

approximate floating point numeric data types [44](#)

binary [42](#)

boolean [45](#)

date [45](#)

numeric (decimal values) [44](#)

numeric (non-decimal values) [43](#)

string [42](#)

time [45](#)

DATEADD () function

using

DATE data type [45](#)

DATEDIFF () function

using [179](#)

date functions

DATEADD ()

DATEDIFF () [179](#)

DATEPART ()

GETDATE () [172](#)

ISDATE () [173](#)

using [171](#)

DATEPART () function

using

DDL triggers [382](#)

DECLARE command [270](#)

DEFAULT constraint [67](#)

DELETE statement [124](#)

syntax [125](#)

de-normalization [5](#)

de-normalized data [5](#)

DENSE_RANK () function [229](#)

deployment [392](#)

different methods [393](#)

full [393](#)

incremental [393](#)

DML (Data Manipulation Language) [101](#)

DML statements

DELETE statement [124](#)

INSERT statement

SELECT statement [104](#)

TRUNCATE statement [126](#)

UPDATE statement [124](#)

DML triggers [383](#)

example

DROP INDEX [94](#)

DROP TABLE command [54](#)

Dynamic SQL

E

Entity Relationship (ER) [8](#)

error handling [314](#)

@@ERROR

error information, retrieving [317](#)

implementing [314](#)

TRY-CATCH

errors

error line [317](#)

error message [317](#)

error number [316](#)

error procedure [317](#)

error severity [317](#)

error state [317](#)

information, retrieving [316](#)

execution time [298](#)

EXISTS function [360](#)

explicit conversion [334](#)

explicit transactions [323](#)

implementing [326](#)

F

filtered columnstore index [94](#)

syntax [94](#)

filtered index [89](#)

syntax [90](#)

filtering, SELECT statement

range search [115](#)

wildcard search

Float [44](#)

FLOOR () function

using [169](#)

Foreign Key [69](#)

cascading [71](#)

Foreign Key constraint [70](#)

FULL OUTER JOIN

G

Generate Script wizard
scripts, generating with
GETDATE () function

using [172](#)

global temporary tables [300](#)

example [302](#)

GOTO keyword [276](#)

Graphical User Interface (GUI) [33](#)

GROUP BY function [135](#)

H

HAVING command [135](#)

sample data, populating
heap RID (key) [87](#)

I

IDENT_CURRENT () [352](#)

IDENTITY value

dealing with

DENT_CURRENT () [352](#)

@@IDENTITY [352](#)

SCOPE_IDENTITY () [353](#)

IF statement

- using [284](#)
- implicit conversion [334](#)
- implicit transactions [322](#)
- index
 - columnstore indexes [91](#)
 - in practical
 - rowstore indexes [80](#)
- index key [83](#)
- index scan [83](#)
- index seek [83](#)
- inline table-valued user-defined function [368](#)

- creating [369](#)
- dropping [371](#)
- executing [370](#)
- modifying [370](#)
- inline view subquery [215](#)
- INNER JOIN [192](#)
- many-to-many relationship
- one-to-many relationship [194](#)
- one-to-one relationship [193](#)
- INSERT statement [103](#)
- aggregate functions, using with
 - example [104](#)
 - syntax [104](#)
- INSTEAD OF trigger [382](#)
- Integrated Development Environment (IDE) [1](#)
- International Organization for Standardization [11](#)
- IS and IS NOT [359](#)
- ISDATE () function
 - using [173](#)
- ISNULL function [346](#)

syntax [347](#).

ISNUMERIC () function

using [167](#).

J

JOIN clause

example [185](#)

INNER JOIN [192](#)

OUTER JOIN [199](#).

versus VLOOKUP in Excel

JSON

URL [254](#).

JSON data

dealing with [255](#)

OPENJSON

table to JSON [264](#).

JSON functions

ISJSON() function [260](#)

JSON_MODIFY() function [261](#)

JSON_QUERY() function [263](#)

JSON_VALUE() function [260](#)

K

key lookup [87](#).

keywords, T-SQL statement [274](#).

BEGIN ... END [274](#).

GOTO [276](#)

RETURN [275](#)

semicolon (;) [274](#)

WAITFOR [277](#)

L

LEFT () function

using [155](#)

LEFT OUTER JOIN

LEN () function

using [158](#)

local temporary tables [300](#)

example

loop [285](#)

LOWER () function

using [160](#)

LTRIM () function

using [159](#)

M

magic tables [127](#)

many-to-many relationships

MAX () function

using [151](#)

MERGE command [309](#)

MIN () function

using [150](#)

multiline table-valued user-defined function [371](#)

creating [372](#)

dropping [375](#)

executing [375](#)
modifying [374](#)

N

nested subquery [215](#)
non-clustered columnstore index [93](#)
syntax [93](#)
non-clustered index [85](#)
syntax [87](#)
intermediate nodes [85](#)
leaf nodes [86](#)
root node [85](#)
NON EXISTS function [361](#)
normalization [6](#)
Not Null constraint [63](#)
NTILE () function [229](#)

NULL [55](#)
NULLIF function [349](#)
NULL values
COALESCE function [348](#)
dealing with [346](#)
ISNULL [347](#)
NULLIF [349](#)
numeric data type
decimal values [44](#)
non-decimal values [43](#)
numeric functions
ABS () [166](#)
CEILING () [169](#)

FLOOR () [169](#),
ISNUMERIC () [167](#),
POWER () [170](#),
RAND () [171](#),
ROUND () [168](#),
using [166](#)

O

one-to-many relationship [194](#),
one to one relationship [193](#),
Online Analytical Processing (OLAP) [79](#),
Online Transaction Processing (OLTP) [79](#),
OPENJSON
OPENXML
table to XML
OUTER APPLY
using

OUTER JOIN [199](#),
FULL OUTER JOIN
LEFT OUTER JOIN
RIGHT OUTER JOIN
OUTPUT clause in DML statements
OUTPUT clause with DELETE statement [129](#),
OUTPUT clause with INSERT statement [128](#),
OUTPUT clause with UPDATE statement [129](#),

P

paging [119](#),

- Parameterized SQL [354](#)
- PARSE function [339](#)
 - examples [339](#)
- partition [220](#)
- Plan Cache [25](#)
- Point in Time Recovery [402](#)
- POWER () function
 - using [170](#)
- Primary Key [56](#)
 - methods
 - versus, UNIQUE constraint [66](#)
- Primary Key constraint [56](#)
- programmability objects [362](#)
- stored procedures [376](#)
- triggers [382](#)
- user-defined functions [365](#)
- views

Q

- Quality Assurance (QA) team [392](#)

- QUOTENAME () function
 - using [165](#)

R

- RAND () function
 - using [171](#)
- RANK () function
 - using [228](#)

ranking functions

DENSE_RANK () [229](#)

NTILE () [229](#)

RANK () [228](#)

ROW_NUMBER() [228](#)

T-SQL example [230](#)

using [227](#)

Recursive CTE [309](#)

Relational Database Management System (RDBMS) [5](#)

constraints [8](#)

de-normalization [6](#)

Entity Relationship (ER) [8](#)

Foreign Key [9](#)

normalization

Primary Key [8](#)

trigger [9](#)

relations [5](#)

REPLACE () function

using [157](#)

REPLICATE () function

using [162](#)

RETURN keyword [275](#)

REVERSE () function

using [163](#)

RID lookup [87](#)

RIGHT () function

using [155](#)

RIGHT OUTER JOIN

root node [238](#)

ROUND () function

using [168](#)

ROW_NUMBER() function

using [228](#)

rowstore indexes [80](#)

clustered indexes

covering index [89](#)

filtered index [90](#)

non-clustered indexes

unique index [91](#)

RTRIM () function

using [159](#)

S

scalar subquery [216](#)

correlated [216](#)

simple [216](#)

scalar user-defined functions [365](#)

creating [366](#)

dropping [368](#)

invoking [367](#)

modifying [367](#)

SCOPE_IDENTITY () [353](#)

SELECT command [273](#)

SELECT statement [104](#)

filtering

paging

SELECT-specific columns [106](#)

SELECT * statement [104](#)

sorting

TOP [121](#)

semicolon keyword [274](#)
SET command [271](#)
SET NOCOUNT {ON | OFF} [357](#)
Software Development Life Cycle (SDLC) [392](#)
sorting [117](#)
example [118](#)
SPACE () function
using [161](#)
SPID [299](#)
SQL Injection [354](#)
SQL Server
downloading [16](#)
installing [16](#)
version and editions [16](#)
SQL Server Management Studio (SSMS) [16](#)
comment out selected lines [29](#)
executing [24](#)
indent, decreasing [30](#)
indent, increasing [29](#)
Object Explorer [21](#)
Query Editor [23](#)

query, parsing [23](#)
Results to File [28](#)
Results to Grid [27](#)
Results to Text [28](#)
uncomment out selected lines [29](#)
working with
SQL Server physical architecture [12](#)
database [14](#)
data file [15](#)
extent [13](#)

- file group [15](#)
- log file [15](#)
- mixed extent [13](#)
- pages [13](#)
- partition [14](#)
- schema [14](#)
- system database [14](#)
- table [14](#)
- uniform extent [14](#)
- user database [15](#)
- SQL Server Query Optimizer [25](#)
- stored procedures [37.5](#)
 - advantages [37.6](#)
 - creating [37.8](#)
 - dropping [38.1](#)
 - invoking [38.0](#)
 - modifying [37.9](#)
 - with default parameters [37.7](#)
 - with output parameters [37.7](#)

- string data type
 - fixed length data type [41](#)
 - variable length data type [42](#)
- string functions
 - CHARINDEX () [164](#)
 - LEFT () [154](#)
 - LEN () [158](#)
 - LOWER () [160](#)
 - LTRIM () [159](#)
 - QUOTENAME () [165](#)
 - REPLACE () [157](#)
 - REPLICATE () [162](#)

REVERSE () [163](#)
RIGHT () [155](#)
RTRIM () [159](#)
SPACE () [161](#)
STRING_SPLIT () [166](#)
SUBSTRING () [156](#)
UPPER () [161](#)
using [154](#)
STRING_SPLIT () function [165](#)
using [166](#)
Structured Query Language (SQL) [11](#)
subqueries, types
inline view [215](#)
nested [215](#)
scalar subquery [216](#)
subquery [214](#)
types [215](#)

SUBSTRING () function
using [156](#)
SUM () function
using
super admin [19](#)

T

table-valued user-defined functions [368](#)
inline table-valued functions [368](#)
multiline table-valued user-defined function [371](#)
table variable [272](#)
tempdb database [300](#)

- temporary tables [299](#)
 - creating [301](#)
 - global temporary tables [300](#)
 - local temporary tables [300](#)
 - three-part naming
- TIME data type [45](#)
- TOP [123](#)
- syntax [122](#)
- Transaction Control Language (TCL) [12](#)
- transaction management [321](#)
 - auto commit transaction [321](#)
 - batch-scoped transaction [322](#)
 - explicit transaction [322](#)
 - explicit transaction, implementing [326](#)
 - implicit transaction [321](#)
 - implicit transaction example [322](#)
- @@TRANCOUNT [324](#)
- XACT_STATE() [325](#)

- triggers [381](#)
 - AFTER trigger [382](#)
 - CLR triggers [381](#)
 - DDL triggers [382](#)
 - DML triggers [383](#)
 - INSTEAD OF trigger [382](#)
 - types [381](#)
- TRUNCATE statement [126](#)
 - example [127](#)
 - syntax [127](#)
- TRY_CAST [340](#)
- TRY ... CATCH
- TRY_CONVERT [340](#)

TRY_PARSE [340](#)

T-SQL [11](#)

SQL [11](#)

T-SQL commands [11](#)

Data Control Language (DCL) [12](#)

Data Definition Language (DDL) [11](#)

Data Manipulation Language (DML) [12](#)

Transaction Control Language (TCL) [12](#)

T-SQL script

executing [393](#)

generating

T-SQL statement

keywords [274](#)

U

UNIQUE constraint

unique index [91](#)

syntax [91](#)

UPDATE statement [123](#)

example [124](#)

syntax [123](#)

UPPER () function

using [161](#)

USE DATABASE command

user-defined functions [365](#)

scalar user-defined functions

table-valued user-defined function [368](#)

V

- variables [270](#)
- declaring [271](#)
- values, assigning
- values, reading [273](#)
- views [362](#)
- executing [363](#)
- output [364](#)
- VLOOKUP in Excel
- versus JOIN

W

- WAITFOR command [277](#)
- WHILE loop
- BREAK keyword [291](#)
- CAST keyword [290](#)
- CONTINUE keyword [292](#)
- using
- windowing functions [220](#)
- aggregate functions

- analytical functions
- ranking functions [227](#)

X

- XACT_STATE() function [325](#)
- XML data
- dealing with [239](#)

filtering, with XQuery [245](#)

OPENXML

querying, with XQuery [240](#)

XML Header [238](#)

XQuery [240](#)

data, filtering with [245](#)

T-SQL example [241](#)

T-SQL example for XML data

XQuery methods

exists() method [247](#).

modify() method [247](#).

query() method [247](#).