



Intelligent Data Management with SQL Server

Intelligent Data Management with SQL Server

© 2020 Aptech Limited

All rights reserved.

No part of this book may be reproduced or copied in any form or by any means – graphic, electronic or mechanical, including photocopying, recording, taping, or storing in information retrieval system or sent or transferred without the prior written permission of copyright owner Aptech Limited.

All trademarks acknowledged.

APTECH LIMITED

Contact E-mail: ov-support@onlinevarsity.com

Edition 1 - 2020



Onlinevarsity



ANYTIME | ANYWHERE

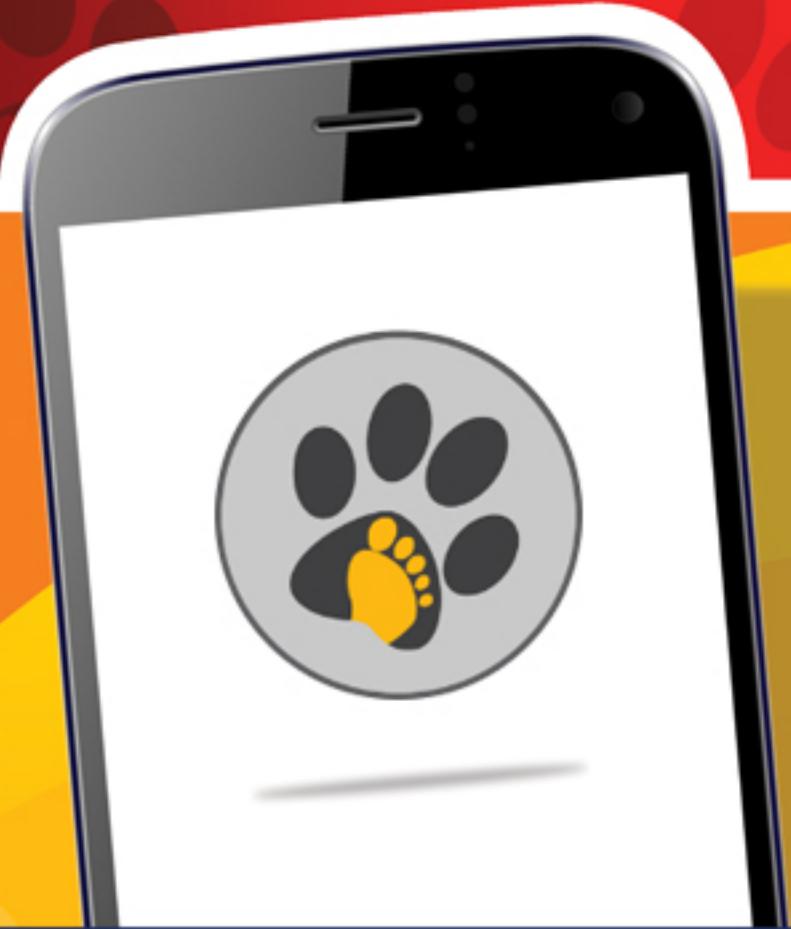
PREFACE

SQL Server is a client-server based Relational Database Management System (RDBMS) from Microsoft. It provides an enterprise-level data management platform for an organization. SQL Server includes numerous features and tools that make it an outstanding database and data analysis platform. It is also targeted for large-scale Online Transaction Processing (OLTP), data warehousing, and e-commerce applications. One of the key features of SQL Server is that it is now available on the cloud too.

The book begins with an introduction to RDBMS concepts and moves on to introduce SQL Server 2019. The book then covers various SQL Server topics such as data types, usage of Transact-SQL, and database objects such as indexes, stored procedures, functions, and so on. The book also introduces Azure SQL and cloud databases. The book describes transactions, programming elements with Transact-SQL, and finally troubleshooting errors with error handling techniques.

The book also explores SQL Server 2019 new features and enhancements. These include features such as Big Data clusters, PolyBase, Query Store, Stretch Database, and In-Memory enhancements. Besides these, you will also learn about the improved Performance Tools and Transact-SQL enhancements.

The knowledge and information in this book is the result of the concentrated effort of the Design Team, which is continuously striving to bring to you the latest, the best and the most relevant subject matter in Information Technology. As a part of Aptech's quality drive, this team does intensive research and curriculum enrichment to keep it in line with industry trends and learner requirements.



Onlinevarsity App for Android devices

Download from Google Play Store

Onlinevarsity

INDUSTRY BEST PRACTICES

SYNC WITH THE INDUSTRY



Contents

1. RDBMS Concepts
2. Entity-Relationship (E-R) Model and Normalization
3. Introduction to SQL Server 2019
4. Transact-SQL
5. Creating and Managing Databases
6. Creating Tables
7. Azure SQL
8. Accessing Data
9. Advanced Queries and Joins
10. Views, Stored Procedures, and Querying Metadata
11. Indexes
12. Triggers
13. Programming Transact-SQL
14. Transactions
15. Error Handling
16. Enhancements in SQL Server 2019
17. PolyBase, Query Store, and Stretch Database

**LEARN
@
YOUR PACE
YOUR
DEVICE**

Onlinevarsity

Session - 1

RDBMS Concepts

Welcome to the Session, **RDBMS Concepts**.

This session deals with the concepts related to databases and database management systems, explores various database models, and introduces the concept of an RDBMS.

In this session, you will learn to:

- Explain the concept of data and database
- Describe the approaches to data management
- Define a Database Management System (DBMS) and list its benefits
- Explain the different database models
- Define and explain RDBMS
- Describe entities and tables and list the characteristics of tables
- List the differences between a DBMS and an RDBMS

1.1 Introduction

Organizations often maintain large amounts of data, which are generated as a result of day-to-day operations. A database is an organized form of such data. It may consist of one or more related data items called records. Think of a database as a data collection to which different questions can be asked. For example, 'What are the phone numbers and addresses of the five nearest post offices?' or 'Do we have any books in our library that deal with health food? If so, on which shelves are they located?' or 'Show me the personnel records and sales figures of five best-performing sales people for the current quarter, but their address details are not required to be shown'.

1.2 Data and Database

Data means information and it is the most important component in any work that is done. In the day-to-day activity, either existing data is used or more data is generated. When this data is gathered and analyzed, it yields information. It can be any information such as information about the vehicle, sports, airways, and so on. For example, a sport magazine journalist (who is a soccer enthusiast) gathers the score (data) of Germany's performance in 10 world cup matches. These scores constitute data. When this data is compared with the data of 10 world cup matches played by Brazil, the journalist can obtain information as to which country has a team that plays better soccer.

Information helps to foresee and plan events. Intelligent interpretation of data yields information. In the world of business, to be able to predict an event and plan for it could save time and money. Consider an example, where a car manufacturing company is planning its annual purchase of certain parts of the car, which has to be imported since it is not locally available. If data of the purchase of these parts for the last

five years is available, the company heads can actually compile information about the total amount of parts imported. Based on these findings, a production plan can be prepared. Therefore, information is a key-planning factor.

A database is a collection of data. Some like to think of a database as an organized mechanism that has the capability of storing information. This information can be retrieved by the user in an effective and efficient manner.

A phone book is a database. The data contained consists of individuals' names, addresses, and telephone numbers. These listings are in alphabetical order or indexed. This allows the user to reference a particular local resident with ease. Ultimately, this data is stored in a database somewhere on a computer. As people move to different cities or states, entries may have to be added or removed from the phone book. Likewise, entries will have to be modified for people changing names, addresses, or telephone numbers, and so on.

Figure 1.1 illustrates the concept of a database.

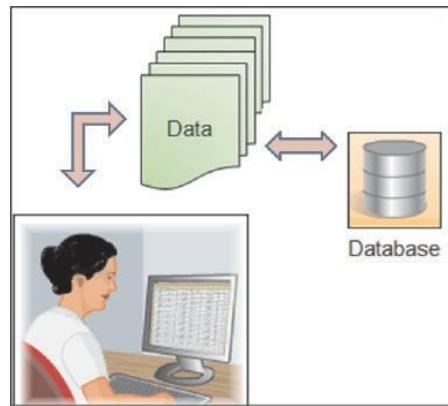


Figure 1.1: Database

Thus, a database is a collection of data that is organized such that its contents can be easily accessed, managed, and updated.

1.3 Data Management

Data management deals with managing large amount of information, which involves both the storage of information and the provision of mechanisms for the manipulation of information. In addition, the system should also provide the safety of the information stored under various circumstances, such as multiple user access and so on.

The two different approaches of managing data are file-based systems and database systems.

1.3.1 File-based Systems

Storage of large amounts of data has always been a matter of huge concern. In early days, file-based systems were used. In this system, data was stored in discrete files and a collection of such files was stored on a computer. These could be accessed by a computer operator. Files of archived data were called tables because they looked like tables used in traditional file keeping. Rows in the table were called records and columns were called fields.

Conventionally, before the database systems evolved, data in software systems was stored in flat files.

An example of the file-based system is illustrated in table 1.1.

First Name	Last Name	Address	Phone
Eric	David	ericd@eff.org	213-456-0987
Selena	Sol	selena@eff.org	987-765-4321
Jordan	Lim	nadroj@otherdomain.com	222-3456-123

Table 1.1: File-based System

➤ Disadvantages of File-based Systems

In a file-based system, different programs in the same application may be interacting with different private data files. There is no system enforcing any standardized control on the organization and structure of these data files.

- **Data redundancy and inconsistency**

Since data resides in different private data files, there are chances of redundancy and resulting inconsistency. For example, a customer can have a savings account as well as a mortgage loan. Here, the customer details may be duplicated since the programs for the two functions store their corresponding data in two different data files. This gives rise to redundancy in the customer's data. Since the same data is stored in two files, inconsistency arises if a change made in the data of one file is not reflected in the other.

- **Unanticipated queries**

In a file-based system, handling sudden/ad-hoc queries can be difficult, since it requires changes in the existing programs. For example, the bank officer must generate a list of all the customers who have an account balance of \$20,000 or more. The bank officer has two choices: either obtain the list of all customers and have the required information extracted manually, or hire a system programmer to design the necessary application program. Both alternatives are obviously unsatisfactory. Suppose that such a program is written, and several days later, the officer must trim that list to include only those customers who have opened their account one year ago. As the program to generate such a list does not exist, it leads to a difficulty in accessing the data.

- **Data isolation**

Data are scattered in various files, and files may be in a different format. Though data used by different programs in the application may be related, they reside as isolated data files.

- **Concurrent access anomalies**

In large multi-user systems, the same file or record may have to be accessed by multiple users simultaneously. Handling this in a file-based system is difficult.

- **Security problems**

In data-intensive applications, security of data is a major concern. Users should be given access only to required data and not to the whole database.

For example, in a banking system, payroll personnel may have to view only that part of the database that has information about various bank employees. They do not require access to information about customer accounts. Since application programs are added to the system in an ad-hoc manner, it is difficult to enforce such security constraints. In a file-based system, this can be handled only by additional programming in each application.

- **Integrity problems**

In any application, there will be certain data integrity rules, which must be maintained. These could be in the form of certain conditions/constraints on the elements of the data records. In the savings bank application, one such integrity rule could be 'Customer ID, which is the unique identifier for a customer record, should not be empty'. There can be several such integrity rules. In a file-based system, all these rules must be explicitly programmed in the application program.

Though all these are common issues of concern to any data-intensive application, each application had to handle all these problems on its own. The application programmer must not only be concerned about implementing the application business rules but also, about handling these common issues.

1.3.2 Database Systems

Database Systems evolved in the late 1960s to address common issues in applications handling large volumes of data, which are also data intensive. Some of these issues could be traced back to the disadvantages of File-based systems.

Databases are used to store data in an efficient and organized manner. A database allows quick and easy management of data. For example, a company may maintain details of its employees in various databases. At any point of time, data can be retrieved from the database, new data can be added into the databases and data can be searched based on some criteria in these databases.

Data storage can be achieved even using simple manual files. For instance, a college has to maintain information about teachers, students, subjects, and examinations.

Details of the teachers can be maintained in a Staff Register and details of the students could be entered in a Student Register and so forth. However, data stored in this form is not permanent. Records in such manual files can only be maintained for a few months or few years. The registers or files are bulky, consume a lot of space, and hence, cannot be kept for many years.

Instead of this, if the same data was stored using database system, it could be more permanent and long-lasting.

➤ **Advantages of database systems**

Information or data can be permanently stored in the form of computerized databases. A database system is advantageous because it provides a centralized control over the data.

Some of the benefits of using such a centralized database system are as follows:

- **The amount of redundancy in the stored data can be reduced**

In an organization, several departments often store the same data. Maintaining a centralized database helps the same data to be accessed by many departments. Thus, duplication of data or 'data

redundancy' can be reduced.

- **No more inconsistencies in data**

When data is duplicated across several departments, any modifications to the data have to be reflected across all departments. Sometimes, this can lead to inconsistency in the data. As there is a central database, it is possible for one person to take up the task of updating the data on a regular basis. Consider that Mr. Larry Finner, an employee of an organization is promoted as a Senior Manager from Manager.

In such a case, there is just one record in the database that must be changed. As a result, data inconsistency is reduced.

- **The stored data can be shared**

A central database can be located on a server, which can be shared by several users. In this way, all users can access the common and updated information all the time.

- **Standards can be set and followed**

A central control ensures that a certain standard in the representation of data can be set and followed. For example, the name of an employee has to be represented as 'Mr. Larry Finner'. This representation can be broken down into the following components:

- A title (Mr.)
- First name (Larry)
- Last name (Finner)

It is certain that all the names stored in the database will follow the same format if the standards are set in this manner.

- **Data Integrity can be maintained**

Data integrity refers to the accuracy of data in the database. For example, when an employee resigns and leaves the organization, consider that the Accounts department has updated its database and the HR department has not updated its records. The data in the company's records is hence, inaccurate.

Centralized control of the database helps in avoiding these errors. It is certain that if a record is deleted from one table, its linked record in the other table is also deleted.

- **Security of data can be implemented**

In a central database system, the privilege of modifying the database is not given to everyone. This right is given only to one person who has full control over the database. This person is called as Database Administrator or DBA. The DBA can implement security by placing restrictions on the data. Based on the permissions granted to them, the users can add, modify, or query data.

1.4 Database Management System (DBMS)

A DBMS can be defined as a collection of related records and a set of programs that access and manipulate these records. A DBMS enables the user to enter, store, and manage data. The main problem with the earlier DBMS packages was that the data was stored in the flat file format. So, the information about different objects was maintained separately in different physical files. Hence, the relations between these objects, if any, had to be maintained in a separate physical file. Thus, a single package would consist of too many files and vast functionalities to integrate them into a single system.

A solution to these problems came in the form of a centralized database system. In a centralized database system, the database is stored in the central location. Everybody can have access to the data stored in a central location from their machine. For example, a large central database system would contain all the data pertaining to the employees. The Accounts and the HR department would access the data required using suitable programs. These programs or the entire application would reside on individual computer terminals.

A Database is a collection of interrelated data, and a DBMS is a set of programs used to add or modify this data. Thus, a DBMS is a set of software programs that allow databases to be defined, constructed, and manipulated.

A DBMS provides an environment that is both convenient and efficient to use when there is a large volume of data and many transactions to be processed. Different categories of DBMS can be used, ranging from small systems that run on personal computers to huge systems that run on mainframes.

Examples of database applications include the following:

- Computerized library systems
- Automated teller machines
- Flight reservation systems
- Computerized parts inventory systems

From a technical standpoint, DBMS products can differ widely. Different DBMS support different query languages, although there is a semi-standardized query language called Structured Query Language (SQL).

Sophisticated languages for managing database systems are called Fourth Generation Language (4GLs). The information from a database can be presented in a variety of formats. Most DBMS include a report writer program that enables the user to output data in the form of a report. Many DBMSs also include a graphics component that enables the user to output information in the form of graphs and charts.

It is not necessary to use general-purpose DBMS for implementing a computerized database. The users can write their own set of programs to create and maintain the database, in effect creating their own special-purpose DBMS software. The database and the software together are called a database system.

The end user accesses the database system through application programs and queries. The DBMS software enables the user to process the queries and programs placed by the end user. The software accesses the data from the database.

Figure 1.2 illustrates a database system.

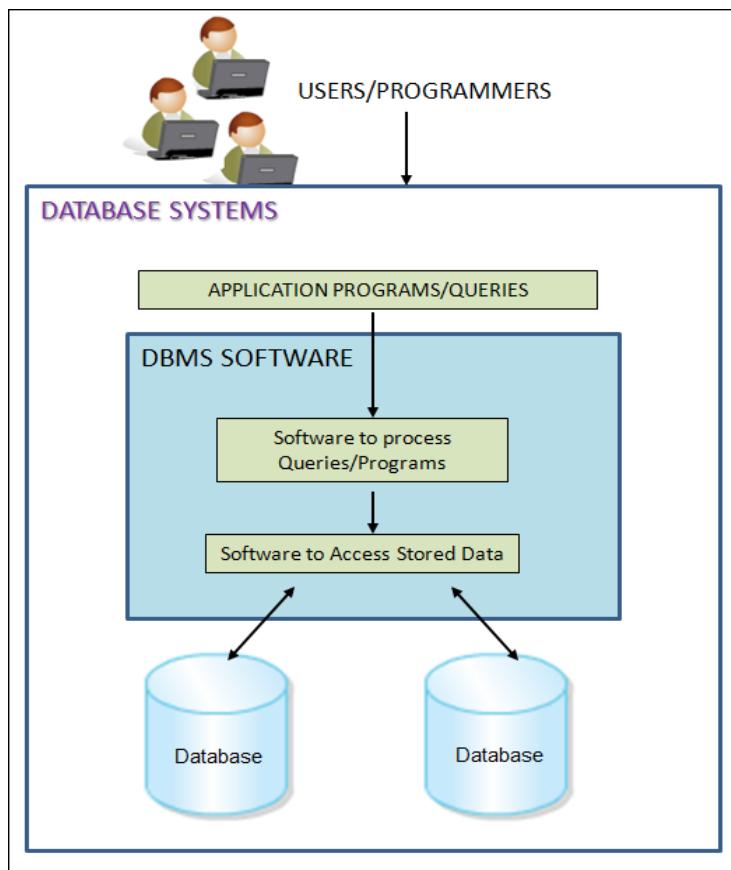


Figure 1.2: A Simplified Database System Environment

1.4.1 Benefits of DBMS

A DBMS is responsible for processing data and converting it into information. For this purpose, the database has to be manipulated, which includes querying the database to retrieve specific data, updating the database, and finally, generating reports.

These reports are the source of information, which is, processed data. A DBMS is also responsible for data security and integrity.

The benefits of a typical DBMS are as follows:

➤ **Data storage**

The programs required for physically storing data, handled by a DBMS, is done by creating complex data structures, and the process is called data storage management.

➤ **Data definition**

A DBMS provides functions to define the structure of the data in the application. These include defining and modifying the record structure, the type and size of fields, and various constraints/conditions to be satisfied by the data in each field.

➤ **Data manipulation**

Once the data structure is defined, data must be inserted, modified, or deleted. The functions, which perform these operations, are also part of a DBMS. These functions can handle planned and unplanned data manipulation requirements. Planned queries are those, which form part of the application.

Unplanned queries are ad-hoc queries, which are performed as and when required.

➤ **Data security and integrity**

Data security is of utmost importance when there are multiple users accessing the database. It is required for keeping a check over data access by users. The security rules specify, which user has access to the database, what data elements the user has access to, and the data operations that the user can perform.

Data in the database should contain as few errors as possible. For example, the employee number for adding a new employee should not be left blank. Telephone number should contain only numbers. Such checks are taken care of by a DBMS.

Thus, the DBMS contains functions, which handle the security and integrity of data in the application. These can be easily invoked by the application and hence, the application programmer does not have to code these functions in the programs.

➤ **Data recovery and concurrency**

Recovery of data after a system failure and concurrent access of records by multiple users are also handled by a DBMS.

➤ **Performance**

Optimizing the performance of the queries is one of the important functions of a DBMS. Hence, the DBMS has a set of programs forming the Query Optimizer, which evaluates the different implementations of a query and chooses the best among them.

➤ **Multi-user access control**

At any point of time, more than one user can access the same data. A DBMS takes care of the sharing of data among multiple users, and maintains data integrity.

➤ **Database access languages and Application Programming Interfaces (APIs)**

The query language of a DBMS implements data access. SQL is the most commonly used query language. A query language is a non-procedural language, where the user must request what is required and does not have to specify how it is to be done. Some procedural languages such as C, Visual Basic, Pascal, and others provide data access to programmers.

1.5 Database Models

Databases can be differentiated based on functions and model of the data. A data model describes a container for storing data, and the process of storing and retrieving data from that container. The analysis and design of data models has been the basis of the evolution of databases. Each model has evolved from the previous one.

1.5.1 Flat File Data Model

In this model, the database consists of only one table or file. This model is used for simple databases - for example, to store the roll numbers, names, subjects, and marks of a group of students. This model cannot handle very complex data. It can cause redundancy when data is repeated more than once.

Table 1.2 depicts the structure of a flat file database.

Roll Number	FirstName	LastName	Subject	Marks
45	Jones	Bill	Maths	84
45	Jones	Bill	Science	75
50	Mary	Mathew	Science	80

Table 1.2: Structure of Flat File Data Model

1.5.2 Hierarchical Data Model

In the Hierarchical Model, different records are inter-related through hierarchical or tree-like structures. In this model, relationships are thought of in terms of children and parents. A parent record can have several children, but a child can have only one parent. To find data stored in this model, the user must know the structure of the tree.

The Windows Registry is an example of a hierarchical database storing configuration settings and options on Microsoft Windows operating systems.

Figure 1.3 illustrates an example of a hierarchical representation.

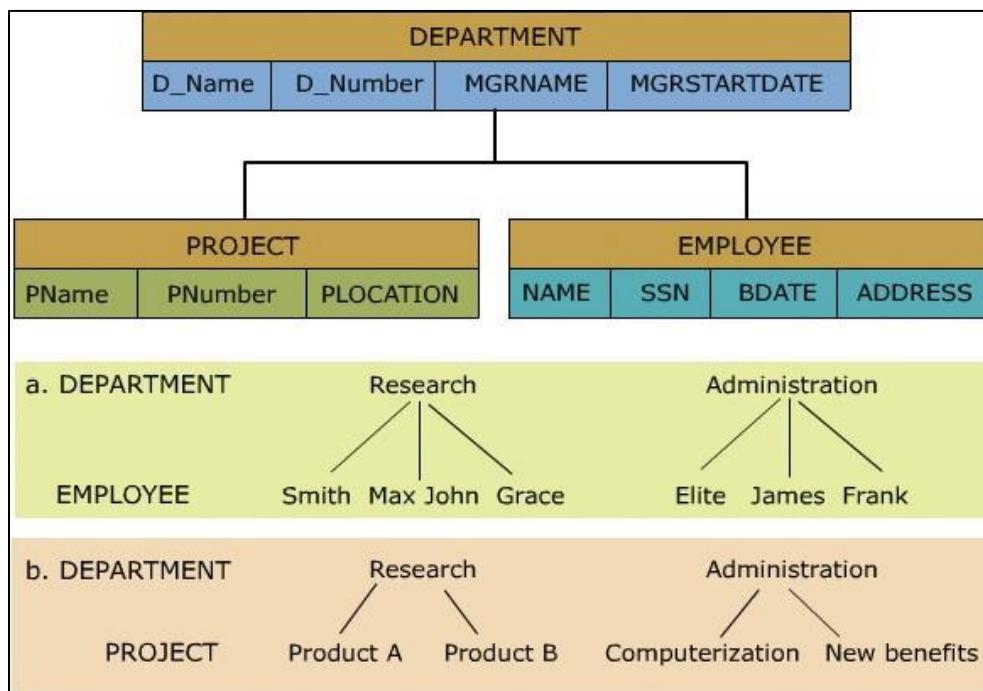


Figure 1.3: Example of a Hierarchical Model

Within the hierarchical model, Department is perceived as the parent of the segment. The tables, Project and Employee, are children. A path that traces the parent segments beginning from the left, defines the tree. This ordered sequencing of segments tracing the hierarchical structure is called the hierarchical path.

It is clear from the figure that in a single department, there can be many employees and a department can have many projects.

➤ **Advantages of the hierarchical model**

The advantages of a hierarchical model are as follows:

- Data is held in a common database so data sharing becomes easier, and security is provided and enforced by a DBMS.
- Data independence is provided by a DBMS, which reduces the effort and costs in maintaining the program.

This model is very efficient when a database contains a large volume of data. For example, a bank's customer account system fits the hierarchical model well because each customer's account is subject to a number of transactions.

1.5.3 Network Data Model

This model is similar to the Hierarchical Data Model. The hierarchical model is actually a subset of the network model. However, instead of using a single-parent tree hierarchy, the network model uses set theory to provide a tree-like hierarchy with the exception that child tables were allowed to have more than one parent.

In the network model, data is stored in sets, instead of the hierarchical tree format. This solves the problem of data redundancy. The set theory of the network model does not use a single-parent tree hierarchy. It allows a child to have more than one parent. Thus, the records are physically linked through linked-lists. Integrated Database Management System (IDMS) from Computer Associates International Inc. and Raima Database Manager (RDM) Server by Raima Inc. are examples of a Network DBMS.

The network model together with the hierarchical data model was a major data model for implementing numerous commercial DBMS. The network model structures and language constructs were defined by Conference on Data Systems Language (CODASYL).

For every database, a definition of the database name, record type for each record, and the components that make up those records is stored. This is called its network schema. A portion of the database as seen by the application's programs that actually produce the desired information from the data contained in the database is called sub-schema. It allows application programs to access the required data from the database.

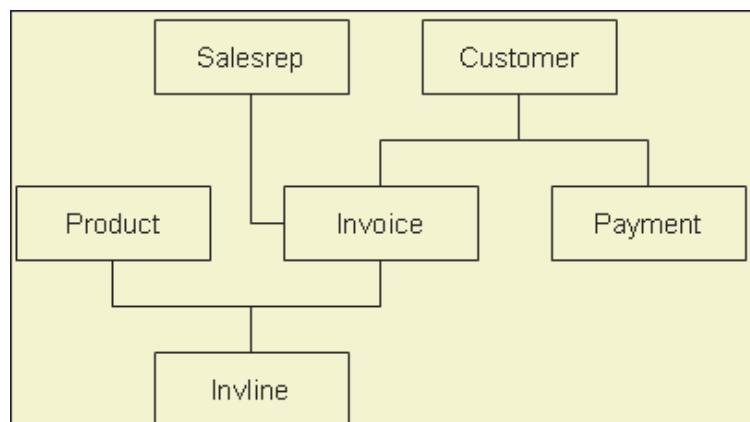


Figure 1.4: Network Model

The network model shown in figure 1.4 illustrates a series of one-to-many relationships, as follows:

1. A sales representative might have written many Invoice tickets, but each Invoice is written by a single Sales representative (Salesrep).
2. A Customer might have made purchases on different occasions. A Customer may have many Invoice tickets, but each Invoice belongs only to a single customer.
3. An Invoice ticket may have many Invoice lines (Invline), but each Invline is found on a single Invoice ticket.
4. A Product may appear in several different Invline, but each Invline contains only a single Product.

Components of the language used with network models are as follows:

Advantages of network model

Advantages of such a structure are specified as follows:

The relationships are easier to implement in the network database model than in the hierarchical model.

This model enforces database integrity.

This model achieves sufficient data independence.

Disadvantages of network model

Disadvantages are specified as follows:

The databases in this model are difficult to design.

The programmer has to be very familiar with the internal structures to access the database.

The model provides a navigational data access environment. Hence, to move from A to E in the sequence A-B-C-D-E, the user has to move through B, C, and D to get to E.

This model is difficult to implement and maintain. Computer programmers, rather than end users, utilize this model.

1.5.4 Relational Data Model

As demand for information grew and more sophisticated databases and applications were required, database design, management, and use became too cumbersome. The lack of query facility took a lot of time of the programmers to produce even the simplest reports. This led to the development of what came to be called the Relational Model database.

The term 'Relation' is derived from the set theory of mathematics. In the Relational Model, unlike the Hierarchical and Network models, there are no physical links. All data is maintained in the form of tables consisting of rows and columns. Data in two tables is related through common columns and not physical links. Operators are provided for operating on rows in tables.

Popular relational DBMSs are Oracle, Sybase, DB2, Microsoft SQL Server, and so on.

This model represents the database as a collection of relations. In this model's terminology, a row is called a tuple, a column, an attribute, and the table is called a relation. The list of values applicable to a particular field is called domain. It is possible for several attributes to have the same domain. The number of attributes of a relation is called degree of the relation. The number of tuples determines the cardinality of the relation.

In order to understand the relational model, consider tables 1.3 and 1.4.

Roll Number	Student Name
1	Sam Reiner
2	John Parkinson
3	Jenny Smith
4	Lisa Hayes
5	Penny Walker
6	Peter Jordan
7	Joe Wong

Table 1.3: Students Table

Roll Number	Marks Obtained
1	34
2	87
3	45
4	90
5	36
6	65
7	89

Table 1.4: Marks Table

The **Students** table displays the **Roll Number** and the **Student Name**, and the **Marks** table displays the **Roll Number** and **Marks** obtained by the students. Now, two steps must be carried out for students who have scored more than 50. First, locate the roll numbers of those who have scored more than 50 from the **Marks** table. Second, their names have to be located in the **Students** table by matching the roll number.

The result will be as shown in table 1.5.

Roll Number	Student Name	Marks Obtained
4	Lisa	90
6	Peter	65
7	Joe	89

Table 1.5: Displaying Student Names and Marks

It was possible to get this information because of two facts: First, there is a column common to both the tables - **Roll Number**. Second, based on this column, the records from the two different tables could be matched and the required information could be obtained.

In a relational model, data is stored in tables. A table in a database has a unique name that identifies its contents. Each table can be defined as an intersection of rows and columns.

Advantages of relational model

The relational database model gives programmers time to concentrate on logical view of the database rather than being bothered about physical view. One of the reasons for popularity of relational databases is querying flexibility. Most relational databases use Structured Query Language (SQL). An RDBMS uses SQL to translate user query into technical code required to retrieve requested data. Relational model is so easy to handle that even untrained people find it easy to generate handy reports and queries, without giving much thought to the requirement to design a proper database.

Disadvantages of relational model

- Though the model hides all complexities of system, it tends to be slower than other database systems.
- As compared to all other models, relational data model is the most popular and widely used.

As compared to all other models, the relational data model is the most popular and widely used.

1.6 Relational Database Management System (RDBMS)

The Relational Model is an attempt to simplify database structures. It represents all data in the database as simple row-column tables of data values. An RDBMS is a software program that helps to create, maintain, and manipulate a relational database. A relational database is a database divided into logical units called tables, where tables are related to one another within the database.

Tables are related in a relational database, allowing adequate data to be retrieved in a single query (although the desired data may exist in more than one table). By having common keys, or fields, among relational database tables, data from multiple tables can be joined to form one large resultset.

Figure 1.5 shows two tables related to one another through a common key (data value) in a relational database.

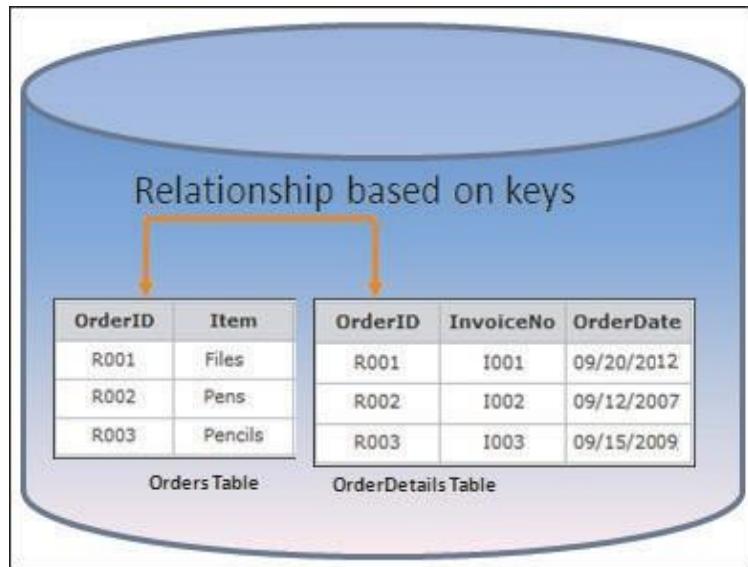


Figure 1.5: Relationship Between Tables

Thus, a relational database is a database structured on the relational model. The basic characteristic of a relational model is that in a relational model, data is stored in relations. To understand relations, consider the following example.

The **Capitals** table shown in table 1.6 displays a list of countries and their capitals, and the **Currency** table shown in table 1.7 displays the countries and the local currencies used by them.

Country	Capital
Greece	Athens
Italy	Rome
USA	Washington
China	Beijing
Japan	Tokyo
Australia	Sydney
France	Paris

Table 1.6: Capitals

Country	Currency
Greece	Drachma
Italy	Lira
USA	Dollar
China	Renminbi (Yuan)
Japan	Yen
Australia	Australian Dollar
France	Francs

Table 1.7: Currency

Both the tables have a common column, that is, the **Country** column. Now, if the user wants to display the information about the currency used in Rome, first find the name of the country to which Rome belongs. This information can be retrieved from table 1.6. Next, that country should be looked up in table

1.7 to find out the currency.

It is possible to get this information because it is possible to establish a relation between the two tables through a common column called **Country**.

1.6.1 Terms Related to RDBMS

There are certain terms that are mostly used in an RDBMS. These are described as follows:

- Data is presented as a collection of relations.
- Each relation is depicted as a table.
- Columns are attributes.
- Rows ('tuples') represent entities.
- Every table has a set of attributes that are taken together as a 'key' (technically, a 'superkey'), which uniquely identifies each entity.

For example, a company might have an **Employee** table with a row for each employee. What attributes might be interesting for such a table? This will depend on the application and the type of use the data will be put to, and is determined at database design time.

Consider the scenario of a company maintaining customer and order information for products being sold and customer-order details for a specific month, such as, August.

The tables 1.8, 1.9, 1.10, and 1.11 are used to illustrate this scenario. These tables depict tuples and attributes in the form of rows and columns. Various terms related to these tables are given in table 1.12.

Cust_No	Cust_Name	Phone No
002	David Gordon	0231-5466356
003	Prince Fernandes	0221-5762382
003	Charles Yale	0321-8734723
002	Ryan Ford	0241-2343444
005	Bruce Smith	0241-8472198

Table 1.8: Customer

Item_No	Description	Price
HW1	Power Supply	4000
HW2	Keyboard	2000
HW3	Mouse	800
SW1	Office Suite	15000
SW2	Payroll Software	8000

Table 1.9: Items

Ord_No	Item_No	Qty
101	HW3	50
101	SW1	150
102	HW2	10
103	HW3	50
104	HW2	25
104	HW3	100
105	SW1	100

Table 1.10: Order_Details

Ord_No	Ord_Date	Cust_No
101	02-08-12	002
102	11-08-12	003
103	21-08-12	003
104	28-08-12	002
105	30-08-12	005

Table 1.11: Order_August

Term	Meaning	Example from the Scenario
Relation	A table	Order_August, Order_Details, Customer and Items
Tuple	A row or a record in a relation	A row from Customer relation is a Customer tuple
Attribute	A field or a column in a relation	Ord_Date, Item_No, Cust_Name, and so on
Cardinality of a relation	The number of tuples in a relation	Cardinality of Order_Details relation is 7
Degree of a relation	The number of attributes in a relation	Degree of Customer relation is 3
Domain of an attribute	The set of all values that can be taken by the attribute	Domain of Qty in Order_Details is the set of all values which can represent quantity of an ordered item
Primary Key of a relation	An attribute or a combination of attributes that uniquely defines each tuple in a relation	Primary Key of Customer relation is Cust_No Ord_No and Item_No combination form the primary key of Order_Details
Foreign Key	An attribute or a combination of attributes in one relation R1 that indicates the relationship of R1 with another relation R2 The foreign key attributes in R1 must contain values matching with those of the values in R2	Cust_No in Order_August relation is a foreign key creating reference from Order_August to Customer. This is required to indicate the relationship between orders in Order_August and Customer

Table 1.12: Terms Related to Tables

1.6.2 RDBMS Users

The primary goal of a database system is to provide an environment for retrieving information from and storing new information into the database.

For a small personal database, one person typically defines the constructs and manipulates the database.

However, many persons are involved in the design, use, and maintenance of a large database with a few hundred users.

Database Administrator (DBA)

- Is a person who collects information that will be stored in database.
- Administering these resources is responsibility of DBA. DBA is also responsible for authorizing access to the database, for coordinating and monitoring its use, and for acquiring software and hardware resources as required. DBA is accountable for problems such as breach of security or poor system response time.

Database Designer

- Are responsible for identifying the data to be stored in the database and for choosing appropriate structures to represent and store this data. It is the responsibility of database designers to communicate with all prospective database users, in order to understand their requirements, and to come up with a design that meets the requirements.

System Analysts and Application Programmers

- Determine requirements of end users, and develop specifications for pre-determined transactions that meet these requirements. Application Programmers implement these specifications as programs; then, they test, debug, document, and maintain these pre-determined transactions.

DBMS Designers and Implementers

- Design and implement DBMS modules and interfaces as a software package. A DBMS is a complex software system that consists of many components or modules, including modules for implementing the catalog, query language, interface processors, data access, and security. A DBMS must interface with other system software such as the operating system and compilers for various programming languages.

End User

- Invokes an application to interact with the system, or writes a query for easy retrieval, modification, or deletion of data.

1.7 Entities and Tables

The components of an RDBMS are entities and tables, which will be explained in this section.

1.7.1 Entity

An entity is a person, place, thing, object, event, or even a concept, which can be distinctly identified. For example, the entities in a university are students, faculty members, and courses.

Each entity has certain characteristics known as attributes. For example, the student entity might include attributes such as student number, name, and grade. Each attribute should be named appropriately.

A grouping of related entities becomes an entity set. Each entity set is given a name. The name of the entity set reflects the contents. Thus, the attributes of all the students of the university will be stored in an entity set called **Student**.

1.7.2 Tables and their Characteristics

The access and manipulation of data is facilitated by the creation of data relationships based on a construct known as a table. A table contains a group of related entities that is an entity set. The terms entity set and table are often used interchangeably. A table is also called a relation. The rows are known as tuples. The columns are known as attributes. Figure 1.6 highlights the characteristics of a table.

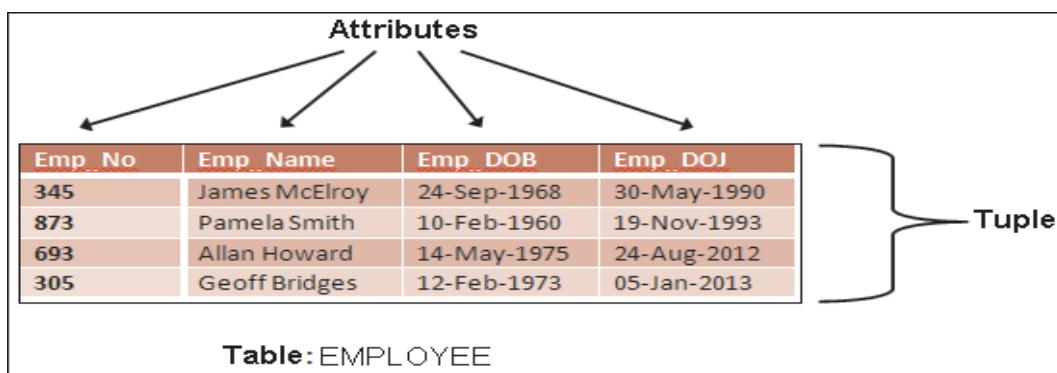


Figure 1.6: Characteristics of a Table

The characteristics of a table are as follows:

- A two-dimensional structure composed of rows and columns is perceived as a table.
- Each tuple represents a single entity within the entity set.
- Each column has a distinct name.
- Each row/column intersection represents a single data value.
- Each table must have a key known as primary key that uniquely identifies each row.
- All values in a column must conform to the same data format. For example, if the attribute is assigned a decimal data format, all values in the column representing that attribute must be in decimals.
- Each column has a specific range of values known as the attribute domain.
- Each row carries information describing one entity occurrence.
- The order of the rows and columns is immaterial in a DBMS.

1.8 Differences between a DBMS and an RDBMS

Differences between a DBMS and an RDBMS are listed in table 1.13.

DBMS	RDBMS
It does not require to have data in tabular structure nor does it enforce tabular relationships between data items.	In an RDBMS, tabular structure is a must and table relationships are enforced by the system. These relationships enable the user to apply and manage business rules with minimal coding.
Small amount of data can be stored and retrieved.	An RDBMS can store and retrieve large amount of data.
A DBMS is less secure than an RDBMS.	An RDBMS is more secure than a DBMS.
It is a single user system.	It is a multi-user system.
Most DBMSs do not support client/server architecture.	It supports client/server architecture.

Table 1.13: Difference between DBMS and RDBMS

In an RDBMS, a relation is given more importance. Thus, the tables in an RDBMS are dependent and the user can establish various integrity constraints on these tables so that the ultimate data used by the user remains correct. In case of a DBMS, entities are given more importance and there is no relation established among these entities.

1.9 Check Your Progress

1. The _____ data model allows a child node to have more than one parent.

(A)	Flat File	(C)	Network
(B)	Hierarchical	(D)	Relational

2. _____ is used to administer permissions on the databases and database objects.

(A)	Data Definition Language (DDL)	(C)	Sub-schema
(B)	Data Manipulation Language (DML)	(D)	Data Control Language (DCL)

3. In the relational model terminology, a row is called a _____, a column an _____, and a table a _____.

(A)	attribute, tuple, relation	(C)	attribute, relation, tuple
(B)	tuple, attribute, relation	(D)	row, column, tuple

4. A _____ can be defined as a collection of related records and a set of programs that access and manipulate these records.

(A)	Database Management System	(C)	Data Management
(B)	Relational Database Management System	(D)	Network Model

5. A _____ describes a container for storing data and the process of storing and retrieving data from that container.

(A)	Network model	(C)	Data model
(B)	Flat File model	(D)	Relational model

1.9.1 Answers

1.	C
2.	D
3.	B
4.	B
5.	C



Summary

- A database is a collection of related data stored in the form of a table.
- A data model describes a container for storing data and the process of storing and retrieving data from that container.
- A DBMS is a collection of programs that enables the user to store, modify, and extract information from a database.
- A Relational Database Management System (RDBMS) is a suite of software programs for creating, maintaining, modifying, and manipulating a relational database.
- A relational database is divided into logical units called tables. These logical units are interrelated to each other within the database.
- The main components of an RDBMS are entities and tables.
- In an RDBMS, a relation is given more importance, whereas, in case of a DBMS, entities are given more importance and there is no relation established among these entities.



Try It Yourself

1. Create a PowerPoint presentation highlighting in brief what is a DBMS, different database models, and key features of an RDBMS.

Session - 2

Entity-Relationship (E-R) Model and Normalization

Welcome to the Session, **Entity-Relationship (E-R) Model and Normalization**.

This session talks about Data Modeling, the E-R model, its components, symbols, diagrams, relationships, Data Normalization, and Relational Operators.

In this session, you will learn to:

- Define and describe data modeling
- Identify and describe the components of the E-R model
- Identify relationships that can be formed between entities
- Explain E-R diagrams and their use
- Describe an E-R diagram, the symbols used for drawing, and show various relationships
- Describe various Normal Forms
- Outline uses of different Relational Operators

2.1 Introduction

A data model is a group of conceptual tools that describes data, its relationships, and semantics. It also consists of the consistency constraints that the data adheres to. The Entity-Relationship, Relational, Network, and Hierarchical models are examples of data models. The development of every database begins with the basic step of analyzing its data in order to determine the data model that would best represent it. Once this step is completed, the data model is applied to the data.

2.2 Data Modeling

The process of applying an appropriate data model to the data, in order to organize and structure it, is called data modeling.

Data modeling is as essential to database development as are planning and designing to any project development. Building a database without a data model is similar to developing a project without its plans and design. Data models help database developers to define the relational tables, primary and foreign keys, stored procedures, and triggers required in the database.

Data modeling can be broken down into the following three broad steps:

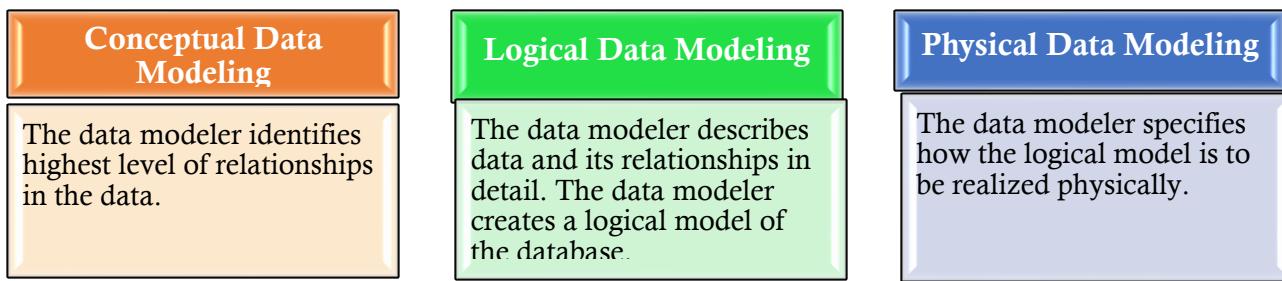


Figure 2.1 exhibits various steps involved in data modeling.

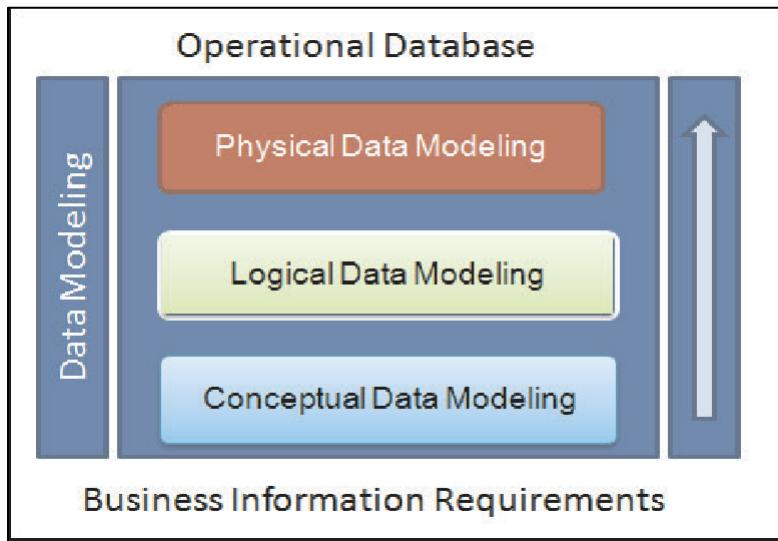
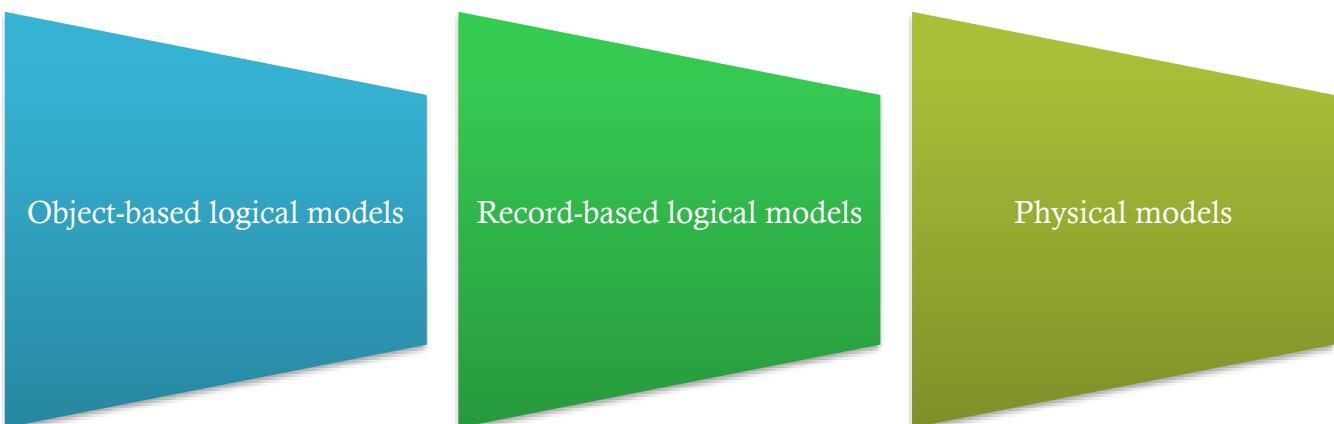


Figure 2.1: Data Modeling Steps

2.3 Entity-Relationship (E-R) Model

Data models can be classified into three different groups:



The Entity-Relationship (E-R) model belongs to the first classification.

The model is based on a simple idea. Data can be perceived as real-world objects called entities and the relationships that exist between them. For example, the data about employees working for an organization can be perceived as a collection of employees and a collection of various departments that form the organization. Both employee and department are real-world objects. An employee belongs to a department. Thus, the relation 'belongs to' links an employee to a particular department.

The employee-department relation can be modeled as shown in figure 2.2.

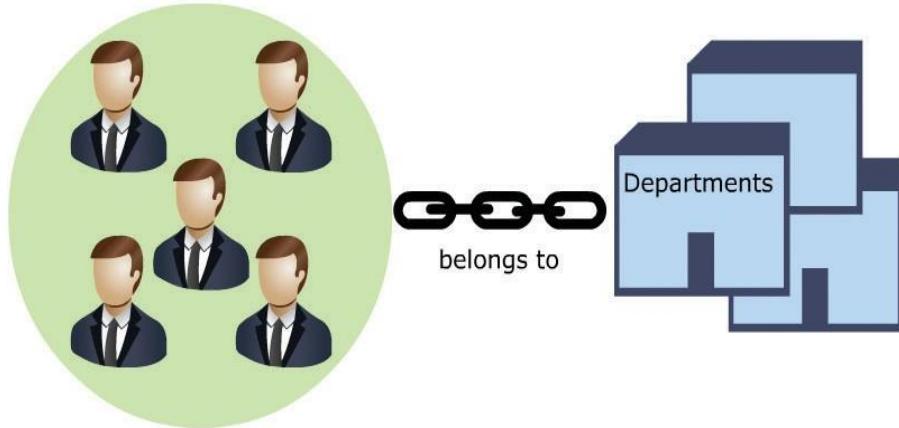


Figure 2.2: E-R Model Depiction of an Organization

An E-R model consists of five basic components. They are as follows:

Entity

An entity is a real-world object that exists physically and is distinguishable from other objects. For example, employee, department, student, customer, vehicle, and account are entities.

Relationship

A relationship is an association or bond that exists between one or more entities. For example, belongs to, owns, works for, saves in, purchased, and so on.

Attributes

Attributes are features that an entity has. Attributes help distinguish every entity from another. For example, the attributes of a student would be **roll_number**, **name**, **stream**, **semester**, and so on. The attributes of a car would be **registration_number**, **model**, **manufacturer**, **color**, **price**, **owner**, and so on.

Entity Set

An entity set is the collection of similar entities. For example, the employees of an organization collectively form an entity set called employee entity set.

Relationship Set

A collection of similar relationships between two or more entity sets is called a relationship set. For example, employees work in a particular department. The set of all 'work in' relations that exists between the employees and the department is called the 'work in' relationship set.

Various E-R model components can be seen in figure 2.3.

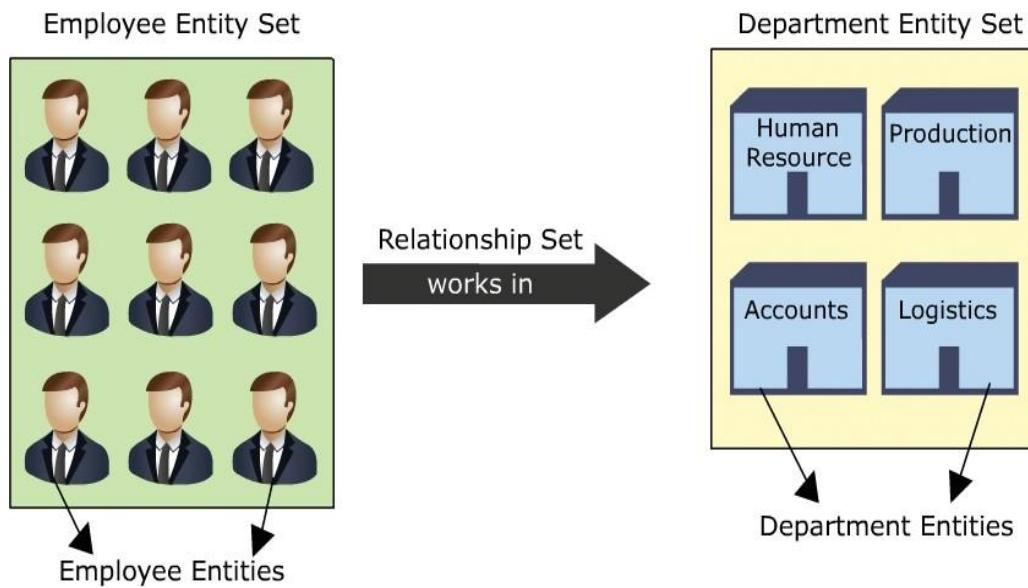


Figure 2.3: Components of the E-R Model

Relationships associate one or more entities and can be of three types. They are as follows:

➤ **Self-relationships**

Relationships between entities of the same entity set are called self-relationships. For example, a manager and his team member, both belong to the employee entity set. The team member works for the manager. Thus, the relation, 'works for', exists between two different employee entities of the same employee entity set.

The relationship can be seen in figure 2.4.

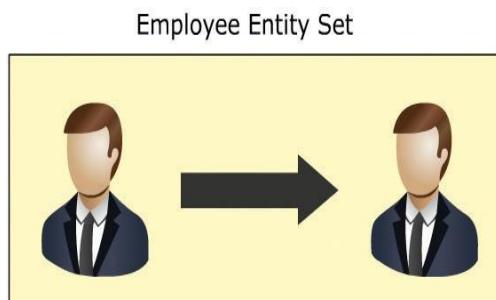


Figure 2.4: Self-Relationship

➤ **Binary relationships**

Relationships that exist between entities of two different entity sets are called binary relationships. For example, an employee belongs to a department. The relation exists between two different entities, which belong to two different entity sets. The employee entity belongs to an employee entity set. The department entity belongs to a department entity set.

The relationship can be seen in figure 2.5.

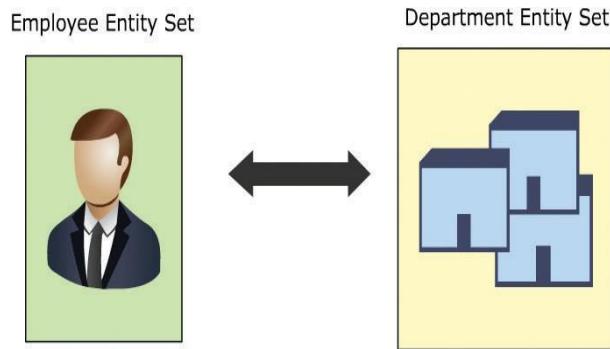


Figure 2.5: Binary Relationship

➤ **Ternary relationships**

Relationships that exist between three entities of different entity sets are called ternary relationships. For example, an employee works in the accounts department at the regional branch. The relation, 'works' exists between all three, the employee, the department, and the location.

The relationship can be seen in figure 2.6.

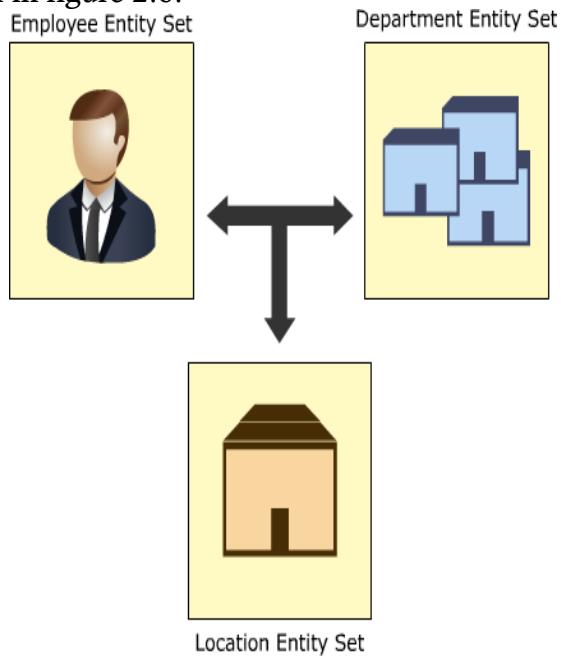


Figure 2.6: Ternary Relationship

Relationships can also be classified as per mapping cardinalities. Different mapping cardinalities are as follows:

One-to-One

This kind of mapping exists when an entity of one entity set can be associated with only one entity of another set. Consider the relationship between a vehicle and its registration. Every vehicle has a unique registration. No two vehicles can have the same registration details. The relation is one-to-one, that is, one

vehicle-one registration. The mapping cardinality can be seen in figure 2.7.

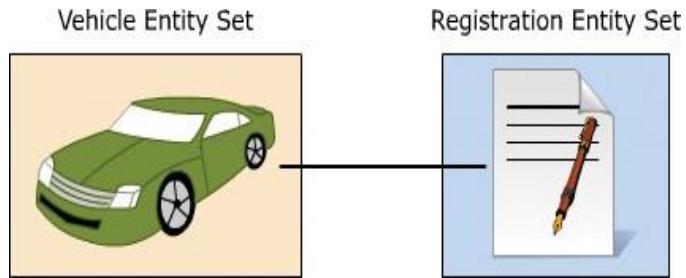


Figure 2.7: One-to-One Mapping Cardinality

One-to- Many

This kind of mapping exists when an entity of one set can be associated with more than one entity of another entity set.

Consider the relation between a customer and the customer's vehicles. A customer can have more than one vehicle. Therefore, the mapping is a one to many mapping, that is, one customer - one or more vehicles. The mapping cardinality can be seen in figure 2.8.

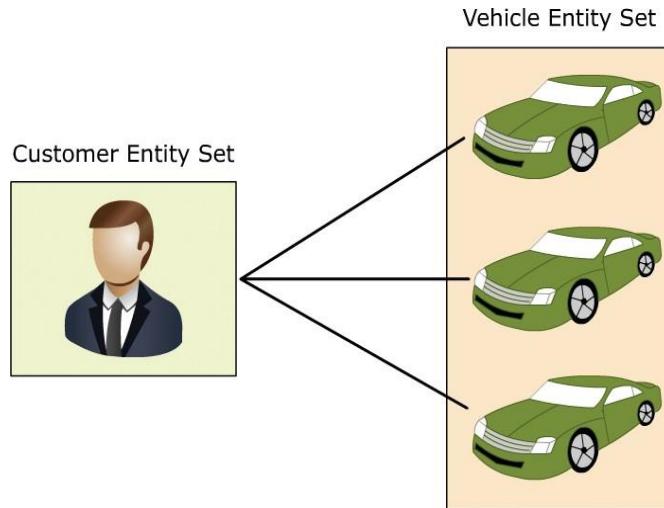


Figure 2.8: One-to-Many Mapping Cardinality

Many-to-One

This kind of mapping exists when many entities of one set is associated with an entity of another set. This association is done irrespective of whether the latter entity is already associated to other or more entities of the former entity set.

Consider the relation between a vehicle and its manufacturer. Every vehicle has only one manufacturing company or coalition associated to it under the relation, 'manufactured by', but the same company or coalition can manufacture more than one kind of vehicle.

The mapping can be seen in figure 2.9.

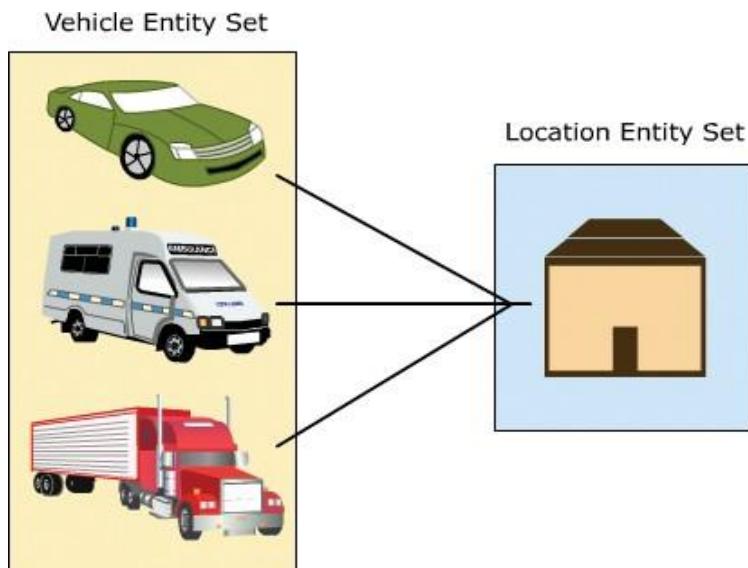


Figure 2.9: Many-to-One Mapping Cardinality

Many-to-Many

This kind of mapping exists when any number of entities of one set can be associated with any number of entities of the other entity set.

Consider the relation between a bank's customer and the customer's accounts. A customer can have more than one account and an account can have more than one customer associated with it in case it is a joint account or similar. Therefore, the mapping is many-to-many, that is, one or more customers associated with one or more accounts.

The mapping cardinality can be seen in figure 2.10.

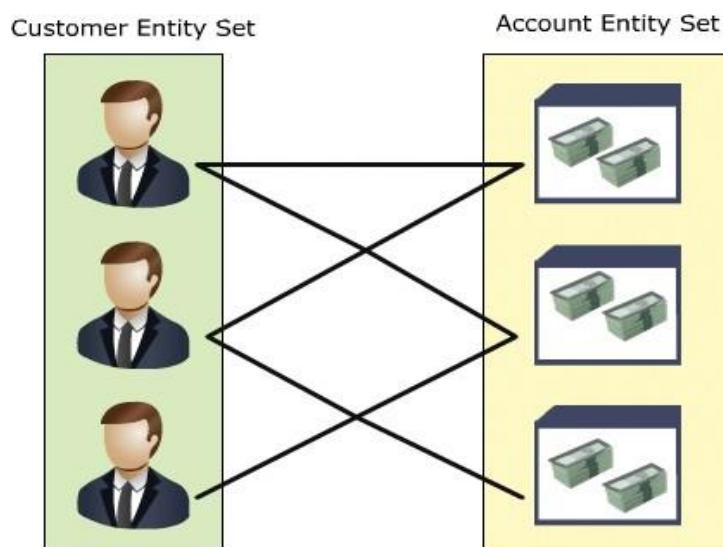


Figure 2.10: Many-to-Many Mapping Cardinality

Some additional concepts in the E-R model are as follows:

Primary keys

A primary key is an attribute that can uniquely define an entity in an entity set. Consider table 2.1 containing the details of students in a school.

Enrollment_Number	Name	Grade	Division
786	Ashley	Seven	B
957	Joseph	Five	A
1011	Kelly	One	A

Table 2.1: Student Details

In a school, every student has a unique **enrollment_number** (such as **enrollment_number** in table 2.1), which is unique to the student. Any student can be identified based on the enrollment number. Thus, the attribute **enrollment_number** plays the role of the primary key in the **Student Details** table.

Weak entity sets

Entity sets that do not have enough attributes to establish a primary key are called weak entity sets.

Strong entity sets

Entity sets that have enough attributes to establish a primary key are called strong entity sets. Consider the scenario of an educational institution where at the end of each semester, students are required to complete and submit a set of assignments. The teacher keeps track of the assignments submitted by the students. Now, an assignment and a student can be considered as two separate entities. The assignment entity is described by the attributes **assignment_number** and **subject**. The student entity is described by **roll_number**, **name**, and **semester**. The assignment entities can be grouped to form an assignment entity set and the student entities can be grouped to form a student entity set. The entity sets are associated by the relation 'submitted by'. This relation is depicted in figure 2.11.

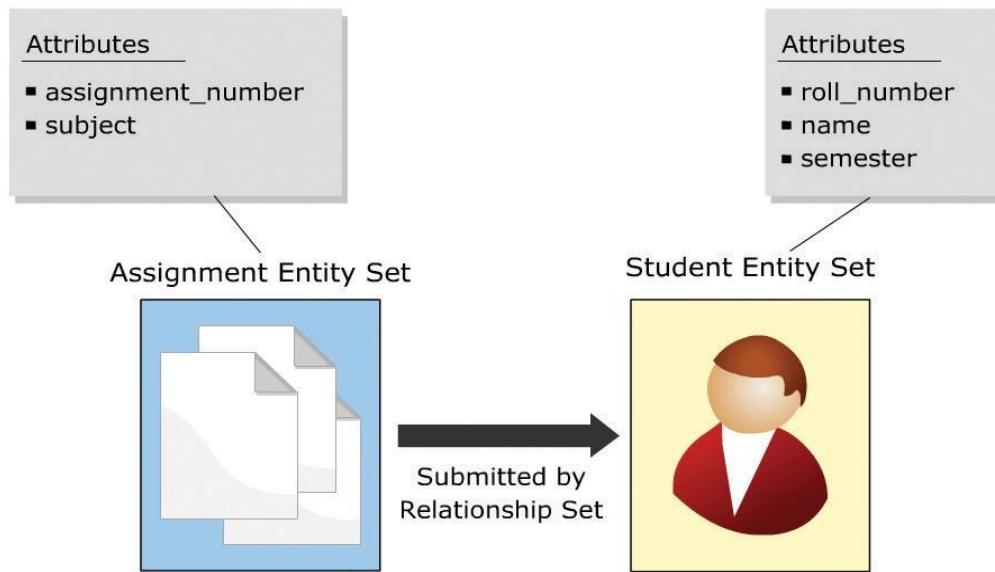


Figure 2.11: Assignment Student Relation

The attributes, `assignment_number` and `subject`, are not enough to identify an assignment entity uniquely. The `roll_number` attribute alone is enough to uniquely identify any student entity. Therefore, `roll_number` is a primary key for the student entity set. The assignment entity set is a weak entity set since it lacks a primary key. The student entity set is a strong entity set due to the presence of the `roll_number` attribute.

2.3.1 Entity-Relationship Diagrams

The E-R diagram is a graphical representation of the E-R model. The E-R diagram, with the help of various symbols, effectively represents various components of the E-R model.

The symbols used for various components can be seen in table 2.2.

Component	Symbol	Example
Entity	Entity	Student
Weak Entity	Weak Entity	Assignments
Attribute	Attribute	Roll_num

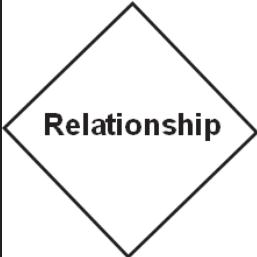
Component	Symbol	Example
Relationship	 Relationship	 Saves in
Key Attribute	 Attribute	 Acct_num

Table 2.2: E-R Diagram Symbols

Attributes in the E-R model can be further classified as follows:

➤ **Multi-valued**

A multi-valued attribute is illustrated with a double-line ellipse, which has more than one value for at least one instance of its entity. This attribute may have upper and lower bounds specified for any individual entity value.

The telephone attribute of an individual may have one or more values, that is, an individual can have one or more telephone numbers. Hence, the telephone attribute is a multi-valued attribute.

The symbol and example of a multi-valued attribute can be seen in figure 2.12.

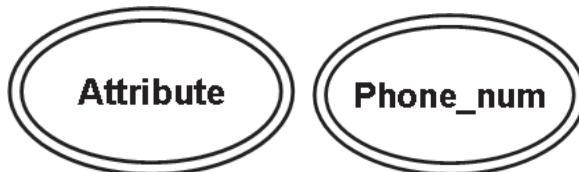


Figure 2.12: Symbol and Example of Multi-valued Attribute

➤ **Composite**

A composite attribute may itself contain two or more attributes, which represent basic attributes having independent meanings of their own.

The address attribute is usually a composite attribute, composed of attributes such as street, area, and so on. The symbol and example of a composite attribute can be seen in figure 2.13.

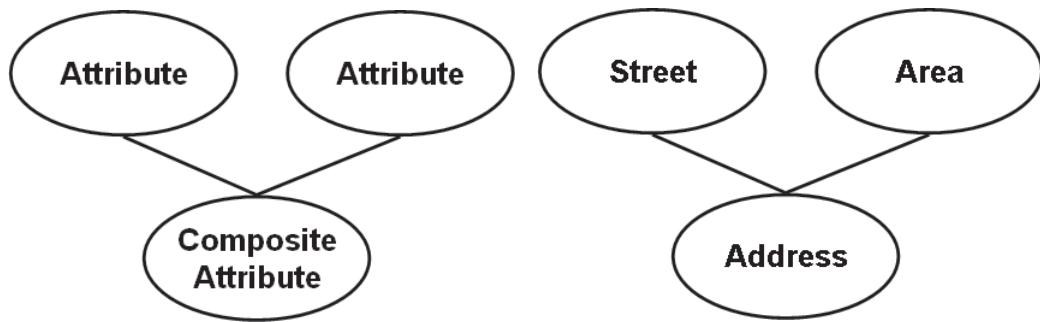


Figure 2.13: Symbol and Example of Composite Attribute

➤ **Derived**

Derived attributes are attributes whose value is entirely dependent on another attribute and are indicated by dashed ellipses.

The age attribute of a person is the best example for derived attributes. For a particular person entity, the age of a person can be determined from the current date and the person's birth date. The symbol and example of a derived attribute can be seen in figure 2.14.



Figure 2.14: Symbol and Example of Derived Attribute

Steps to construct an E-R diagram are as follows:

1. Gather all the data that must be modeled.
2. Identify data that can be modeled as real-world entities.
3. Identify the attributes for each entity.
4. Sort entity sets as weak or strong entity sets.
5. Sort entity attributes as key attributes, multi-valued attributes, composite attributes, derived attributes, and so on.
6. Identify the relations between the different entities.
7. Using different symbols draw the entities, their attributes, and their relationships. Use appropriate symbols while drawing attributes.

Consider the scenario of a bank, with customers and accounts. The E-R diagram for the scenario can be constructed as follows:

1. Step 1: Gather data

The bank is a collection of accounts used by customers to save money.

2. Step 2: Identify entities

Customer

Account

3. Step 3: Identify the attributes

Customer: customer_name, customer_address, customer_contact

Account: account_number, account_owner, balance_amount

4. Step 4: Sort entity sets

Customer entity set: weak entity set

Account entity set: strong entity set

5. Step 5: Sort attributes

Customer entity set: customer_address - composite, customer_contact - multi-valued

Account entity set: account_number → primary key, account_owner – multi-valued

6. Step 6: Identify relations

A customer 'saves in' an account. The relation is 'saves in'.

7. Step 7: Draw diagram using symbols

Figure 2.15 shows the E-R diagram for the bank.

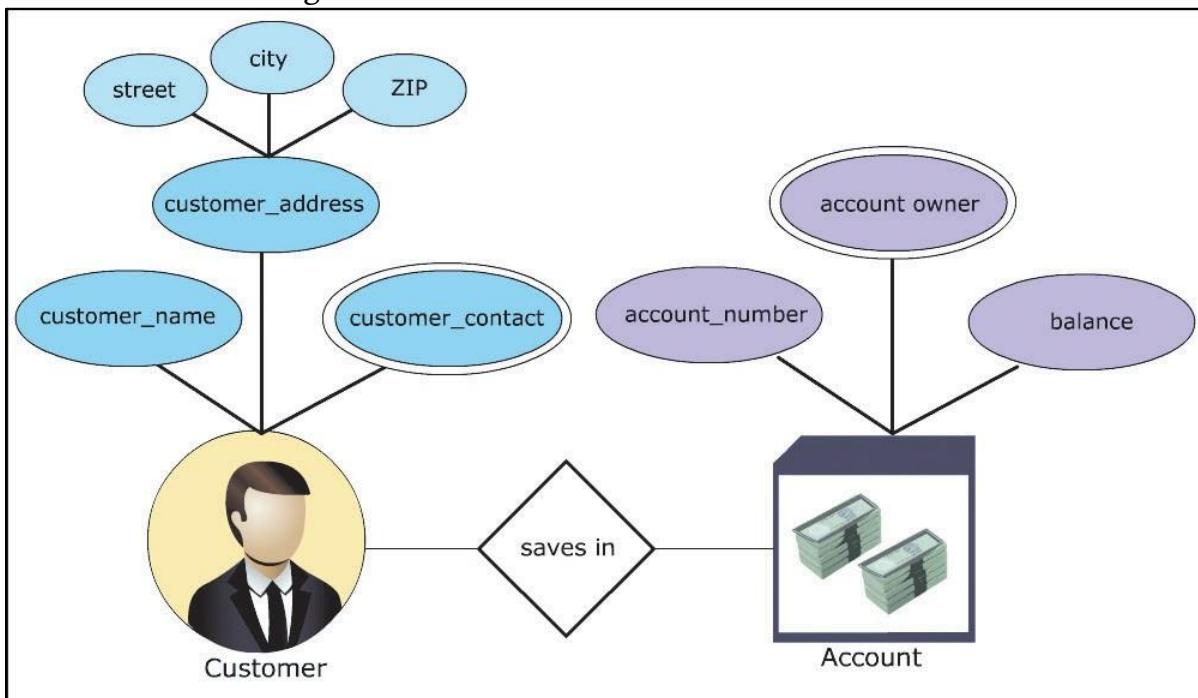


Figure 2.15: E-R Diagram for the Bank

2.4 Normalization

Initially, all databases are characterized by large number of columns and records. This approach has certain drawbacks. Consider the following details of the employees in a department. Table 2.3 consists of the employee details as well as the details of the project they are working on.

Emp_No	Project_Id	Project_Name	Emp_Name	Grade	Salary
142	113, 124	BLUE STAR, MAGNUM	John	A	20,000
168	113	BLUE STAR	James	B	15,000
263	113	BLUE STAR	Andrew	C	10,000
109	124	MAGNUM	Bob	C	10,000

Table 2.3: Department Employee Details

Repetition anomaly

The data such as **Project_Id**, **Project_Name**, **Grade**, and **Salary** repeat many times. This repetition hampers both, performance during retrieval of data and the storage capacity. This repetition of data is called the repetition anomaly.

The repetition is shown in table 2.4 with the help of shaded cells.

Emp_No	Project_Id	Project_Name	Emp_Name	Grade	Salary
142	113, 124	BLUE STAR, MAGNUM	John	A	20,000
168	113	BLUE STAR	James	B	15,000
263	113	BLUE STAR	Andrew	C	10,000
109	124	MAGNUM	Bob	C	10,000

Table 2.4: Department Employee with Insertion Anomaly

Insertion anomaly

Suppose the department recruits a new employee named **Ann**. Now, consider that **Ann** has not been assigned any project. Insertion of her details in the table would leave columns **Project_Id** and **Project_Name** empty. Leaving columns blank could lead to problems later. Anomalies created by such insertions are called insertion anomalies. The anomaly can be seen in table 2.5.

Emp_No	Project_Id	Project_Name	Emp_Name	Grade	Salary
142	113, 124	BLUE STAR, MAGNUM	John	A	20,000
168	113	BLUE STAR	James	B	15,000
263	113	BLUE STAR	Andrew	C	10,000
109	124	MAGNUM	Bob	C	10,000
195	-	-	Ann	C	10,000

Table 2.5: Department Employee Details

Deletion anomaly

Suppose, Bob is relieved from the project MAGNUM. Deleting the record deletes Bob's **Emp_No**, **Grade**, and **Salary** details too. This loss of data is harmful as all of Bob's personal details are also lost as seen in the table 2.6. This kind of loss of data due to deletion is called deletion anomaly. The anomaly can be seen in table 2.6.

Emp_No	Project_Id	Project_Name	Emp_Name	Grade	Salary
142	113, 124	BLUE STAR, MAGNUM	John Smith	A	20,000
168	113	BLUE STAR	James Kilber	B	15,000
263	113	BLUE STAR	Andrew Murray	C	10,000

Table 2.6: Employee Project Details

Updating anomaly

Suppose John was given a hike in **Salary** or John was demoted. The change in John's **Salary** or **Grade** must be reflected in all projects John works for. This problem in updating all the occurrences is called updating anomaly.

The **Department Employee Details** table is called an unnormalized table. These drawbacks lead to the need for normalization.

Normalization is the process of removing unwanted redundancy and dependencies. Initially, Codd (1972) presented three normal forms (1NF, 2NF, and 3NF), all based on dependencies among the attributes of a relation. The fourth and fifth normal forms are based on multi-value and join dependencies and were proposed later.

2.4.1 First Normal Form

In order to achieve the first normal form, following steps must be performed:

Create separate tables for each group of related data

The table columns must have atomic values

All the key attributes must be identified

Consider the **Employee Project Details** table shown in table 2.7.

Emp_No	Project_Id	Project_Name	Emp_Name	Grade	Salary
142	113, 124	BLUE STAR, MAGNUM	John	A	20,000
168	113	BLUE STAR	James	B	15,000
263	113	BLUE STAR	Andrew	C	10,000
109	124	MAGNUM	Bob	C	10,000

Table 2.7: Employee Project Details

The table has data related to projects and employees. The table must be split into two tables, that is, a **Project Details** table and an **Employee Details** table. The table columns, **Project_Id** and **Project_Name**, have multiple values. The data must be split over different rows. The resultant tables

are **Project Details** and **Employee Details** as shown in tables 2.8 and 2.9.

Project_Id	Project_Name
113	BLUE STAR
124	MAGNUM

Table 2.8: Project Details

Emp_No	Emp_Name	Grade	Salary
142	John	A	20,000
168	James	B	15,000
263	Andrew	C	10,000
109	Bob	C	10,000

Table 2.9: Employee Details

The **Project_Id** attribute is the primary key for the **Project Details** table.

The **Emp_No** attribute is the primary key for the **Employee Details** table. Therefore, in first normal form, the initial **Employee Project Details** table has been reduced to the **Project Details** and **Employee Details** tables.

2.4.2 Second Normal Form

Tables are said to be in second normal form if:

- They meet the requirements of the first normal form
- There are no partial dependencies in the tables
- The tables are related through foreign keys

Partial dependency means a non-key attribute should not be partially dependent on more than one key attribute. The **Project Details** and **Employee Details** tables do not exhibit any partial dependencies. The **Project_Name** is dependent only on **Project_Id** and **Emp_Name**, **Grade**, and **Salary** are dependent only on **Emp_No**. The tables also need to be related through foreign keys. A third table, named **Employee Project Details**, is created with only two columns, **Project_Id** and **Emp_No**.

So, the project and employee details tables on conversion to second normal form generates tables **Project Details**, **Employee Details**, and **Employee Project Details** as shown in tables 2.10, 2.11, and 2.12.

Project_Id	Project_Name
113	BLUE STAR
124	MAGNUM

Table 2.10: Project Details After Conversion to Second Normal Form

Emp_No	Emp_Name	Grade	Salary
142	John	A	20,000
168	James	B	15,000
263	Andrew	C	10,000

Emp_No	Emp_Name	Grade	Salary
109	Bob	C	10,000

Table 2.11: Employee Details After Conversion to Second Normal Form

Emp_No	Project_Id
142	113
142	124
168	113
263	113
109	124

Table 2.12: Employee Project Details After Conversion to Second Normal Form

The attributes, **Emp_no** and **Project_id**, of the **Employee Project Details** table combine together to form the primary key. Such primary keys are called composite primary keys.

2.4.3 Third Normal Form

To achieve the third normal form:

- The tables should meet the requirements of the second normal form
- The tables should not have transitive dependencies in them

The **Project Details**, **Employee Details**, and **Employee Project Details** tables are in second normal form. If an attribute can be determined by another non-key attribute, it is called a transitive dependency. To make it simpler, every non-key attribute should be determined by the key attribute only. If a non-key attribute can be determined by another non-key attribute, it must put into another table.

On observing the different tables, it is seen that the **Project Details** and **Employee Project Details** tables do not exhibit any such transitive dependencies. The non-key attributes are totally determined by the key attributes. **Project_Name** is only determined by **Project_Id**. On further scrutinizing the **Employee Details** table, a certain inconsistency is seen. The attribute **Salary** is determined by the attribute **Grade** and not the key attribute **Emp_No**. Thus, this transitive dependency must be removed.

The **Employee Details** table can be split into the **Employee Details** and **Grade Salary Details** tables as shown in tables 2.13 and 2.14.

Emp_No	Emp_Name	Grade
142	John	A
168	James	B
263	Andrew	C
109	Bob	C

Table 2.13: Employee Details After Conversion

Grade	Salary
A	20,000
B	15,000
C	10,000

Table 2.14: Grade Salary Details Table

Thus, at the end of the three normalization stages, the initial `Employee ProjectDetails` table has been reduced to the `Project Details`, `Employee Project Details`, `Employee Details`, and `Grade Salary Details` tables as shown in tables 2.15, 2.16, 2.17, and 2.18.

Project_Id	Project_Name
113	BLUE STAR
124	MAGNUM

Table 2.15: Project Details After Conversion to Third Normal Form

Emp_No	Project_Id
142	113
142	124
168	113
263	113
109	124

Table 2.16: Employee Project Details After Conversion to Third Normal Form

Emp_No	Emp_Name	Grade
142	John	A
168	James	B
263	Andrew	C
109	Bob	C

Table 2.17: Employee Details After Conversion to Third Normal Form

Grade	Salary
A	20,000
B	15,000
C	10,000

Table 2.18: Grade Salary Details After Conversion to Third Normal Form

2.4.4 Denormalization

By normalizing a database, redundancy is reduced. This, in turn, reduces the storage requirements for the database and ensures data integrity. However, it has some drawbacks. They are as follows:

Complex join queries may have to be written often to combine the data in multiple tables.

Joins may practically involve more than three tables depending on the need for information.

If such joins are used very often, the performance of the database will become very poor.

The CPU time required to solve such queries will be very large too. In such cases, storing a few fields redundantly can be ignored to increase the performance of the database. The databases that possess such minor redundancies in order to increase performance are called denormalized databases and the process of doing so is called denormalization.

2.5 Relational Operators

The relational model is based on the solid foundation of Relational Algebra. Relational Algebra consists of a collection of operators that operate on relations. Each operator takes one or two relations as its input and produces a new relation as its output.

Consider the **Branch Reserve Details** table as shown in table 2.19.

Branch	Branch_Id	Reserve (Billion €)
London	BS-01	9.2
London	BS-02	10
Paris	BS-03	15
Los Angeles	BS-04	50
Washington	BS-05	30

Table 2.19: Branch Reserve Details

SELECT

The **SELECT** operator is used to extract data that satisfies a given condition. The lowercase Greek letter sigma, ' σ ', is used to denote selection. A select operation, on the **Branch Reserve Details** table, to display the details of the branches in London would result in table 2.20.

Branch	Branch_Id	Reserve (Billion €)
London	BS-01	9.2
London	BS-02	10

Table 2.20: Details of Branches in London

A selection on the **Branch Reserve Details** table to display branches with reserve greater than 20 billion Euros would result in table 2.21.

Branch	Branch_Id	Reserve (Billion €)
Los Angeles	BS-04	50
Washington	BS-05	30

Table 2.21: Details of Branches with Reserves Greater Than 20 Billion Euros

PROJECT

The **PROJECT** operator is used to project certain details of a relational table. The **PROJECT** operator only displays the required details leaving out certain columns. The **PROJECT** operator is denoted by the Greek letter pi, ' Π '. Assume that only the **Branch_Id** and **Reserve** amounts need to be displayed.

A project operation to do the same, on the **Branch Reserve Details** table, would result in table 2.22.

Branch_Id	Reserve (Billion €)
BS-01	9.2
BS-02	10
BS-03	15
BS-04	50
BS-05	30

Table 2.22: Resultant Table with Branch_Id and Reserve Amounts

PRODUCT

The **PRODUCT** operator, denoted by 'x' helps combine information from two relational tables. Consider table 2.23.

Branch_Id	Loan Amount (Billion €)
BS-01	0.56
BS-02	0.84

Table 2.23: Branch Loan Details

The product operation on the **Branch Reserve Details** and **Branch Loan Details** tables would result in table 2.24.

Branch	Branch_Id	Reserve (Billion €)	Loan Amount (Billion €)
London	BS-01	9.2	0.56
London	BS-01	9.2	0.84
London	BS-02	10	0.56
London	BS-02	10	0.84
Paris	BS-03	15	0.56
Paris	BS-03	15	0.84
Los Angeles	BS-04	50	0.56
Los Angeles	BS-04	50	0.84

Branch	Branch_Id	Reserve (Billion €)	Loan Amount (Billion €)
Washington	BS-05	30	0.56
Washington	BS-05	30	0.84

Table 2.24: Product of Branch Reserve Details and Branch Loan Details

The product operation combines each record from the first table with all the records in the second table, somewhat generating all possible combinations between the table records.

UNION

Suppose an official of the bank with the data given in tables 2.19 and 2.23 wanted to know which branches had reserves below 20 billion Euros or loans. The resultant table would consist of branches with either reserves below 20 billion Euros or loans or both.

This is similar to the union of two sets of data; first, set of branches with reserve less than 20 billion Euros and second, branches with loans. Branches with both, reserves below 20 billion Euros and loans would be displayed only once. The **UNION** operator does just that, it collects the data from the different tables and presents a unified version of the complete data. The union operation is represented by the symbol, 'U'. The union of the **Branch Reserve Details** and **Branch Loan Details** tables would generate table 2.25.

Branch	Branch_Id
London	BS-01
London	BS-02
Paris	BS-03

Table 2.25: Unified Representation of Branches with Less Reserves or Loans

INTERSECT

Suppose the same official after seeing this data wanted to know which of these branches had both low reserves and loans too. The answer would be the intersect relational operation. The **INTERSECT** operator generates data that holds true in all the tables it is applied on. It is based on the intersection set theory and is represented by the ' \cap ' symbol. The result of the intersection of the **Branch Reserve Details** and **Branch Loan Details** tables would be a list of branches that have both reserves below 20 billion Euros and loans in their account. The resultant table generated is table 2.26.

Branch	Branch_Id
London	BS-01
London	BS-02

Table 2.26: Branches with Low Reserves and Loans

DIFFERENCE

If the same official now wanted the list of branches that had low reserves but no loans, then the official would have to use the difference operation. The **DIFFERENCE** operator, symbolized as ' $-$ ', generates data from different tables too, but it generates data that holds true in one table and not the other. Thus, the branch would have to have low reserves and no loans to be displayed.

Table 2.27 is the result generated.

Branch	Branch_Id
Paris	BS-03

Table 2.27: Branches with Low Reserves but No Loans

JOIN

The `JOIN` operation is an enhancement to the product operation. It allows a selection to be performed on the product of tables. For example, if the reserve values and loan amounts of branches with low reserves and loan values was needed, the product of the `Branch Reserve Details` and `Branch Loan Details` would be required. Once the product of tables 2.19 and 2.23 would be generated, only those branches would be listed which have both reserves below 20 billion Euros and loans.

Table 2.28 is generated as a result of the `JOIN` operation.

Branch	Branch_Id	Reserve (Billion €)	Loan Amount (Billion €)
London	BS-01	9.2	0.56
London	BS-02	10	0.84

Table 2.28: Detailed List of Branches with Low Reserve and Loans

DIVIDE

Suppose an official wanted to see the branch names and reserves of all the branches that had loans. This process can be made very easy by using the `DIVIDE` operator. All that the official must do is divide the `Branch Reserve Details` table (shown earlier in table 2.19) by the list of branches, that is, the `Branch Id` column of the `Branch Loan Details` table (shown earlier in table 2.23). Table 2.29 is the result generated.

Branch	Reserve (Billion €)
London	9.2
London	10

Table 2.29: Resultant Table of Division Operation

Note that the attributes of the divisor table should always be a subset of the dividend table. The resultant table would always be void of the attributes of the divisor table and the records not matching the records in the divisor table.

2.6 Check Your Progress

1. One or more attributes that can uniquely define an entity from an entity set is called a _____ key.

(A)	Primary	(C)	Alternate
(B)	Foreign	(D)	Super

2. An attribute that contains two or more attribute values in it is called a _____ attribute.

(A)	Derived	(C)	Multi-valued
(B)	Composite	(D)	Network

3. Transitive dependence is eliminated in the _____ normal form.

(A)	First	(C)	Third
(B)	Second	(D)	Fourth

4. Which one of these operations is further enhanced in the Product operation?

(A)	Divide	(C)	Difference
(B)	Intersection	(D)	Join

5. Which of the following are the basic components of an E-R model?

- a. Entity
- b. Relationship
- c. Attributes
- d. Relationship Chart
- e. Relationship Set

(A)	a, b, c	(C)	a, c, e
(B)	a, d, c	(D)	a, b, c, e

2.6.1 Answers

1.	A
2.	B
3.	C
4.	D
5.	D



Summary

- Data modeling is the process of applying an appropriate data model to the data at hand.
- E-R model views the real-world as a set of basic objects and relationships among them.
- Entity, attributes, entity set, relationships, and relationship sets form the five basic components of E-R model.
- Mapping cardinalities express the number of entities that an entity is associated with.
- The process of removing redundant data from the tables of a relational database is called normalization.
- Relational Algebra consists of a collection of operators that help retrieve data from the relational databases.
- SELECT, PRODUCT, UNION, and DIVIDE are some of the relational algebra operators.

Try It Yourself

1. **Dynamic Data Solutions Ltd.** is a popular company based in **Chicago, Illinois** providing database solutions to its clients. The company is in the process of conducting recruitment interviews in different technical institutes for upcoming projects. The candidates interested in database development can attend the campus interviews. Based on their performance, the candidates are provided short-term training in any DBMS of their choice and recruited for database development in the company.

Consider yourself as a student of second year Computer Science who has participated in the campus interview. You have been provided with a set of interview questions as follows:

- a) Explain in brief the meaning of data and database.
b) List the two different approaches for data management.
c) Explain the benefits of a DBMS.
d) List the different database models used in the industry.
e) Explain the RDBMS model in brief and differentiate between DBMS and RDBMS.
2. Champs Online is an online learning company that provides computer education to kids. The company has basic courses in subjects of Computer Science such as fundamentals of computers, databases, programming languages, and so on. The company has organized their data in a manner similar to this:

Student ID	Student Name	Course ID	Course Title	Fees	Marks
S001	Rob Martin	C001	Programming in Java	5000	50
S002	Maria Stevens	C001	Programming in Java	5000	70
S003	Clark Hood	C002	Networking Fundamentals	6000	80
S002	Maria Stevens	C002	Networking Fundamentals	6000	75
S003	Clark Hood	C003	Database Management	7000	68
S001	Rob Martin	C003	Database Management	7000	79
S002	Maria Stevens	C003	Database Management	7000	90

- a) Identify the problem in the given table.
b) Provide a solution to resolve the problem.

(Hint: Use the concept of Normalization.)

Session - 3

Introduction to SQL Server 2019

Welcome to the Session, **Introduction to SQL Server 2019**. This session introduces SQL Server 2019. It explains the basic architecture of SQL Server 2019 and lists the versions and editions of SQL Server. It also explains the role and structure of SQL Server along with the new features added in SQL Server 2019. Finally, the session explains the process to connect to SQL Server instances, create and organize script files, and execute Transact-SQL queries.

In this session, you will learn to:

- Describe an overview of SQL Server 2019
- Describe basic architecture and version history of SQL Server 2019
- Outline the process of connecting to SQL Server instances
- Define databases and list the key features of AdventureWorks2019 sample database
- Explain the components of SQL Server Management Studio GUI
- Explain script file creation and organization
- Explain the process to execute Transact-SQL queries

3.1 *Introduction to SQL Server 2019*

SQL Server is an RDBMS developed by Microsoft. It provides an enterprise-level data management platform for an organization. SQL Server includes numerous features and tools that make it an outstanding database and data analysis platform. It is also targeted for large-scale Online Transactional Processing (OLTP), data warehousing, and e-commerce applications.

Recent versions of SQL Server have brought a revolutionary change in areas such as speedy transactions, higher security, more profound insights, and the latest hybrid cloud. It enhances mission-essential capabilities of in-memory operations.

Microsoft launched latest version of this product, SQL Server 2019, on November 2019. SQL Server 2019 provides industry leading security, performance and intelligence over your data, regardless of whether it is structured or unstructured. SQL Server 2019 provides support for Big Data Clusters in SQL Server. Existing SQL Server tools such as database engine, SQL Server Analysis Services, SQL Server Machine Learning Services, SQL Server on Linux, and SQL Server Master Data Services have been enhanced in SQL Server 2019. Using SQL Server 2019 not only helps an organization to store and manage huge amount of information, but also to protect and utilize this data at different locations as required.

3.2 Features and Benefits of SQL Server

Some of the key features and benefits of SQL Server in general include:

Strong Security	Better Performance	Multiple Editions and Pricing Models	Simple and Easy Installation Process
Policy-Based Management to detect security policies that are non-compliant. This feature allows only authorized personnel access to the database. Security audits and events can be written automatically to log files.	SQL Server has built-in transparent data compression feature along with encryption. SQL Server provides access control coupled with efficient permission management tools. It also offers an enhanced performance when it comes to data collection.	Microsoft has made available different editions of SQL Server for different kinds of users. These are also priced accordingly. Thus, from hobbyists to professional developers to enterprise users, there is an edition suitable for each one.	SQL Server is simple to install with a one-click installation procedure and readable GUI having easy instructions for the layman.

3.3 Basic Architecture of SQL Server 2019

There are various components that form a part of SQL Server 2019. All the components come together to form the basic architecture of SQL Server 2019. These components can be represented under three major heads that are shown in figure 3.1.

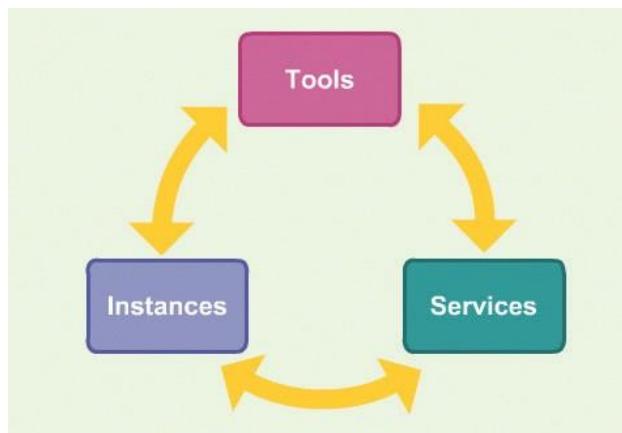


Figure 3.1: Architecture of SQL Server

Tools

There are a number of tools that are provided in SQL Server 2019 for development and query management of a database. The SQL Server Installation Center must be used to install SQL Server program features and tools. Features can also be modified or removed using SQL Server Installation Center. Table 3.1 lists different tools available for SQL Server 2019.

Tool	Description
SQL Server Management Studio (SSMS)	One of the most important tools available in SQL Server 2019 is SSMS. SSMS is a GUI-based application provided with SQL Server 2019 that helps to create databases, database objects, query data, and manage the overall working of SQL Server.
SQLCMD	SQLCMD is a command-line tool that can be used in place of SSMS. It performs similar functions as SSMS, but in command format only.
SQL Server Installation Center	The SQL Server Installation Center tool can also be used to add, remove, and modify SQL Server programs.
SQL Server Configuration Manager	SQL Server Configuration Manager is used by database administrators to manage features of SQL software installed in client machines. This tool is not available to all users. It can be used to configure services, server protocols, client protocols, client aliases, and so on.
SQL Server Profiler	SQL Server Profiler is used to monitor an instance of the Database Engine or Analysis Services.
SQL Server Data Tools (SSDT)	SSDT is an Integrated Development Environment (IDE) used for Business Intelligence Components. It helps to design the database using a tool named Microsoft Visual Studio.
Connectivity Tools	Connectivity tools include DB-Library, Open Database Connectivity (ODBC), Object Linking and Embedding Database (OLE DB), and so on. These tools are used to communicate between the clients, servers, and network libraries.

Table 3.1: Different Tools in SQL Server 2019

Services

There are various services that are executed on a computer running SQL Server. These services run along with the other Windows services and can be viewed in the task manager.

Some of the SQL Server 2019 services are as follows:

SQL Server Database Engine

Database Engine is a core service that is used for storing, processing, and securing data. It is also used for replication, full-text search, and Data Quality Services (DQS). It contains tools for managing relational and eXtensible Markup Language (XML) data.

SQL Server Analysis Services (SSAS)

Analysis Services contain tools that help to create and manage Online Analytical Processing (OLAP). This is used for personal, team, and corporate business intelligence purposes. Analysis services are also used in data mining applications.

SQL Server Reporting Services (SSRS)

Reporting Services help to create, manage, publish, and deploy reports. These reports can be in tabular, matrix, graphical, or free-form format. Report applications can also be created using Reporting Services.

SQL Server Integration Services (SSIS)

Integration Services are used for moving, copying, and transforming data using different graphical tools and programmable objects. The DQS component is also included in Integration Services. Integration services help to build high-performance data integration solutions.

SQL Server Master Data Services

Master Data Services (MDS) are used for master data management. MDS is used for analysis, managing, and reporting information such as hierarchies, granular security, transactions, business rules, and so on.

Instances

All the programs and resource allocations are saved in an instance. An instance can include memory, configuration files, and CPU. Multiple instances can be used for different users in SQL Server 2019. Even though many instances may be present on a single computer, they do not affect the working of other instances. This means that all instances work in isolation. Each instance can be customized as per the requirement. Even permissions for each instance can be granted on individual basis. The resources can also be allocated to the instance accordingly, for example, the number of databases allowed.

In other words, instances can be called as a bigger container that contains sub-containers in the form of databases, security options, server objects, and so on.

3.4 Version History of SQL Server

The first version of SQL Server was released in the year 1989. After this, there have been new versions released almost every year, with the latest one being SQL Server 2019. Figure 3.2 depicts the journey of the flagship database product by Microsoft from 1989 to the present.

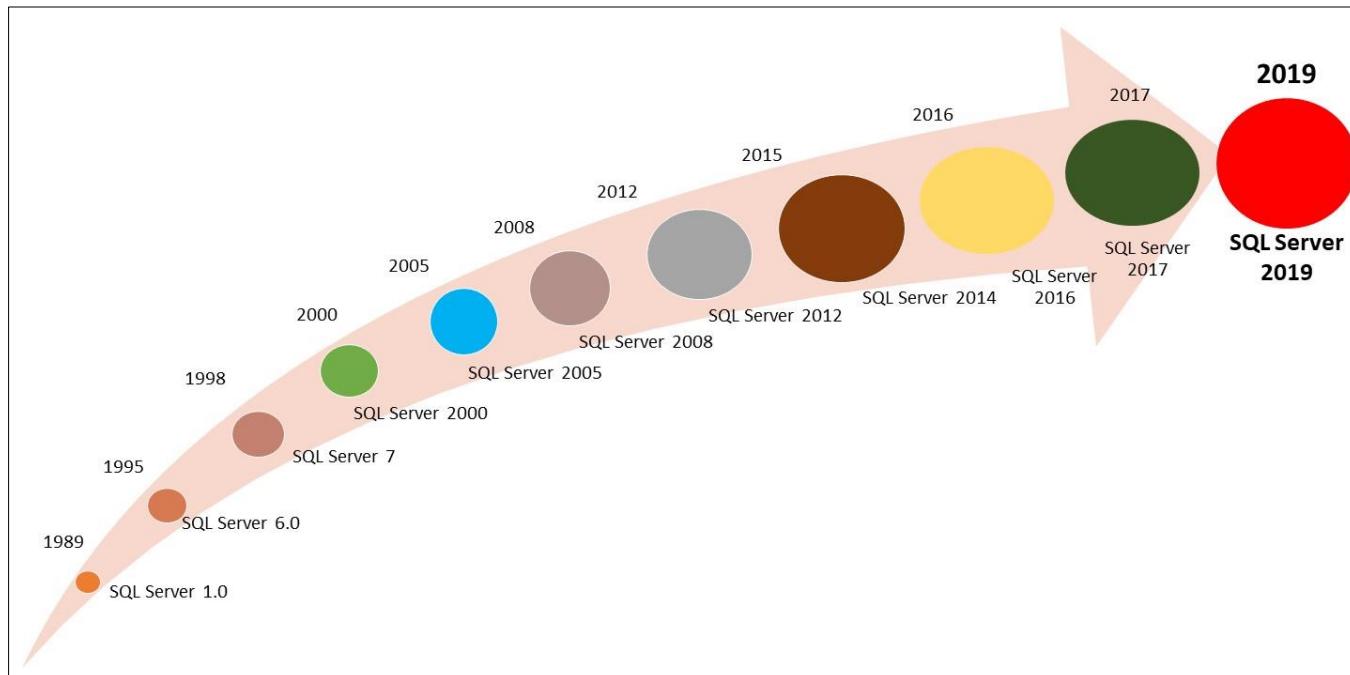


Figure 3.2: Version History of SQL Server

3.5 Editions of SQL Server

Based on database requirements, an organization can choose from any of the following editions of SQL Server 2019 that have been released.

The main editions of SQL Server 2019 are as follows:

Express/Web Edition

Free to use and provides an entry-level database for basic Web and mobile apps

Key Features

- Offers up to 16 cores of CPU for compute capacity
- Up to 64 GB of memory for buffer pool
- In-memory OLTP and Columnstore
- End-to-end encryption with secure enclaves
- Support for Linux and Windows containers
- UTF-8 character encoding
- Data classification and auditing

Standard/Web Edition

Provides full featured database for medium tier applications

Key Features

- Offers up to 24 cores of CPU for compute capacity
- Up to 128 GB of memory for buffer pool
- Supports automatic intelligent database tuning
- Azure Data Studio with notebook support
- Supports Big Data Clusters
- Supports Data virtualization by means of PolyBase
- Improved in-memory performance

In addition to Express/Web features

Enterprise/Web Edition

Provides full featured database for top tier applications

Key Features

- Supports unlimited cores of CPU
- Provides unlimited memory for buffer pool
- Industry-leading performance with unmatched scalability
- Unlimited virtualization benefits
- Access to Power Business Intelligence (BI) Report Server

In addition to Standard and Express/Web features

Developer Edition

Free to use and includes all features of Enterprise edition, licensed for use as a development and test database in a non-production environment.

With each subsequent version, enhancements are made and new features are added.

Features Added in SQL Server 2016

- Real-time operational analytics through PolyBase
- Support for R programming language
- SQL Server Machine Learning Services
- Dynamic data masking, Row level security, and Always encrypted
- Support for end to end mobile BI

Features Added in SQL Server 2017

- Support for Linux including Red Hat Enterprise Linux (RHEL), SUSE Linux Enterprise Server (SLES), and Ubuntu
- Support for Docker containers on Linux and Windows
- Python language support
- Automatic plan correction and adaptive query processing
- Cross platform availability groups
- Support for graph data
- Power BI reporting both on-premises and in the cloud
- Access to Power BI Report Server

Features Added in SQL Server 2019

- Big Data clusters with Apache Spark and HDFS
- Data virtualization to integrate external data sources
- Azure Machine Learning and Spark ML, Support for Kubernetes deployment, Free supported Java
- Native UTF-8 support Intelligent Query processing
- In-Memory Database: Persistent Memory support
- Accelerated database recovery, Free Data Recovery to Azure
- Always Encrypted with secure areas, Data classification, and auditing Vulnerability assessment
- Notebook support for T-SQL, Python, R, and Scala in Azure Data Studio
- SQL Server Analysis

3.6 Connecting to SQL Server Instances

SQL Server Management Studio (SSMS) is used to connect to SQL Server instances. SSMS is a tool used for creating, querying, and managing the databases. In order to use SSMS, you must first launch it and then, connect to SQL Server 2019 Database Engine by specifying server information and login credentials. The login credentials will include username and password.

Detailed steps to connect to SQL Server instance are as follows:

1. Locate the Microsoft SQL Server Management Studio tool on the list of programs on **Start** menu and start the tool.
2. In the **Connect to Server** dialog box, select the **Server type** as **Database Engine**.
3. Type the **Server name**.
4. Select either **Windows Authentication** or **SQL Server Authentication**, provide the required **Login** and **Password**, and click **Connect**.

Figure 3.3 shows the **Connect to Server** dialog box.

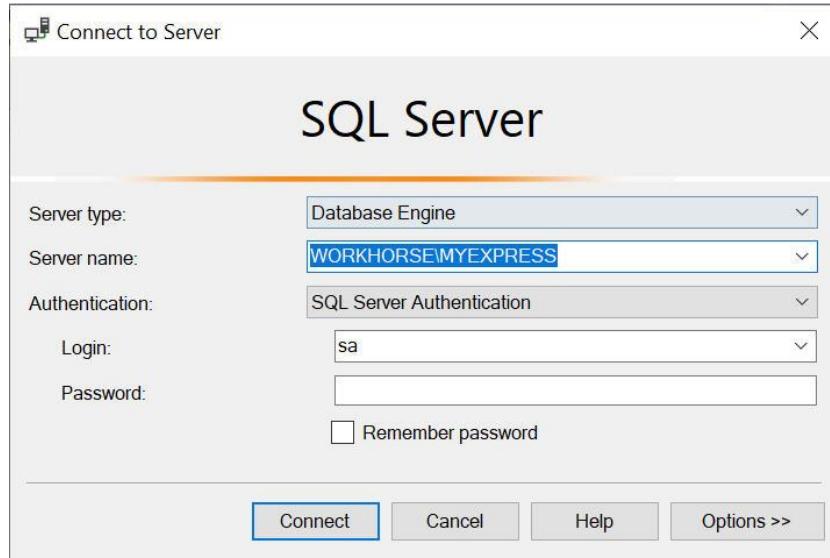


Figure 3.3: Connect to Server Dialog Box

Note - The two authentication methods provided by SQL Server are SQL Server Authentication and Windows Authentication. SQL Server Authentication requires a user account for login and password. Hence, multiple user accounts can access the information using their respective usernames and passwords. With Windows Authentication, the operating system credentials can be used to log in to the SQL Server database. This will work only on a single machine and cannot be used in any other computer.

Figure 3.4 shows the SSMS window.

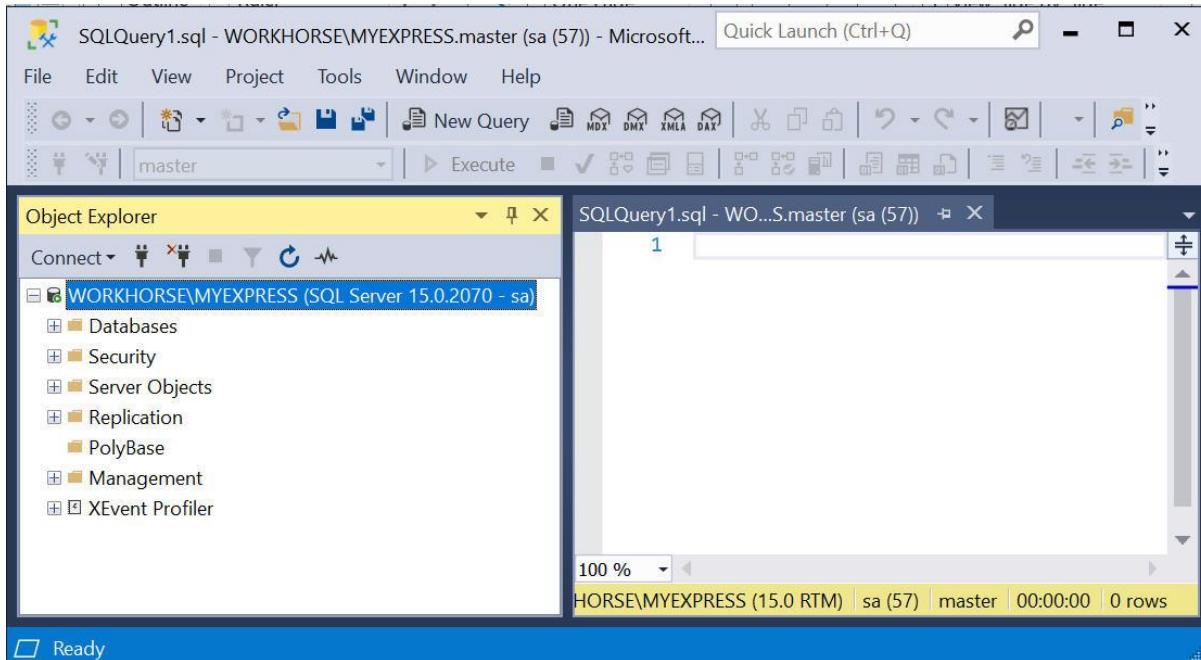


Figure 3.4: SSMS Window

3.7 Introduction to Databases

A database is a collection of data stored in data files on a disk or some removable medium. A database consists of data files to hold actual data.

An SQL Server database is made up of collection of tables that stores sets of specific structured data. A table includes a set of rows (also called as records or tuples) and columns (also called as attributes). Each column in the table is intended to store a specific type of information, for example, dates, names, currency amounts, and numbers.

SQL Server 2019 supports three kinds of databases, which are as follows:



3.7.1 System Databases

SQL Server uses system databases to support different parts of the DBMS. Each database has a specific role and stores job information that requires to be carried out by SQL Server. The system databases store data in tables, which contain the views, stored procedures, and other database objects. They also have associated database files (for example, .mdf and .ldf files) that are physically located on the SQL Server machine.

Using SQL Server 2019, users can create their own databases, also called user-defined databases, and work with them. The purpose of these databases is to store user data.

Table 3.2 shows the system databases that are supported by SQL Server 2019.

Database	Description
master	The database records all system-level information of an instance of SQL Server.
msdb	The database is used by SQL Server Agent for scheduling database alerts and various jobs.
model	The database is used as a template for all databases to be created on the particular instance of SQL Server 2019.
resource	The database is a read-only database. It contains system objects included with SQL Server 2019.
tempdb	The database holds temporary objects or intermediate result sets.

Table 3.2: System Databases

3.7.2 User-defined Databases

Using SQL Server 2019, users can create their own databases, also called user-defined databases, and work with them. The purpose of these databases is to store user data.

3.7.3 AdventureWorks2019 Sample Database

The sample database, AdventureWorks, was introduced from SQL Server 2005 onwards. This database demonstrates use of new features introduced in SQL Server. A fictitious company called Adventure Works Cycles is created as a scenario in this database. Adventure Works Cycles is a large, multinational manufacturing company. The company manufactures and sells metal and composite bicycles to North American, European, and Asian commercial markets. In SQL Server 2019, a new version of the sample database AdventureWorks2019 is used. A readymade script is available from Microsoft for developers to install this database and populate it with fictitious data.

The AdventureWorks2019 database schema covers many functional areas for a fictitious bicycle manufacturer. These areas include:

- Customer/sales force automation and analysis
- Human resources
- Purchasing/Vendor Electronic Data Interchange
- Manufacturing work flow

The database comprises several features. Some of its key features are as follows:

A database engine that includes administration facilities, data access capabilities, Full-Text Search facility, Common Language Runtime (CLR) integration advantage, and more

A set of integrated samples for two multiple feature-based samples: HRResume and Storefront

Analysis Services and Integration Services

Notification Services

Replication Facilities

Reporting Services

The sample database consists of these parts:

- AdventureWorks2019: Sample OLTP database
- AdventureWorks2019DW: Sample Data warehouse
- AdventureWorks2019AS: Sample Analysis Services database

3.8 Understanding the SSMS User Interface

SSMS shows several menu options, toolbars, and panes. The pane on the left seen in figure 3.4 is called Object Explorer. It displays all the objects in the server in a tabular format and provides a user interface to manage them. Capabilities of Object Explorer vary slightly depending on the type of server.

3.8.1 Role and Structure of Object Explorer in SQL Server

The structure of **Object Explorer** in SQL Server 2019 is shown in figure 3.5.

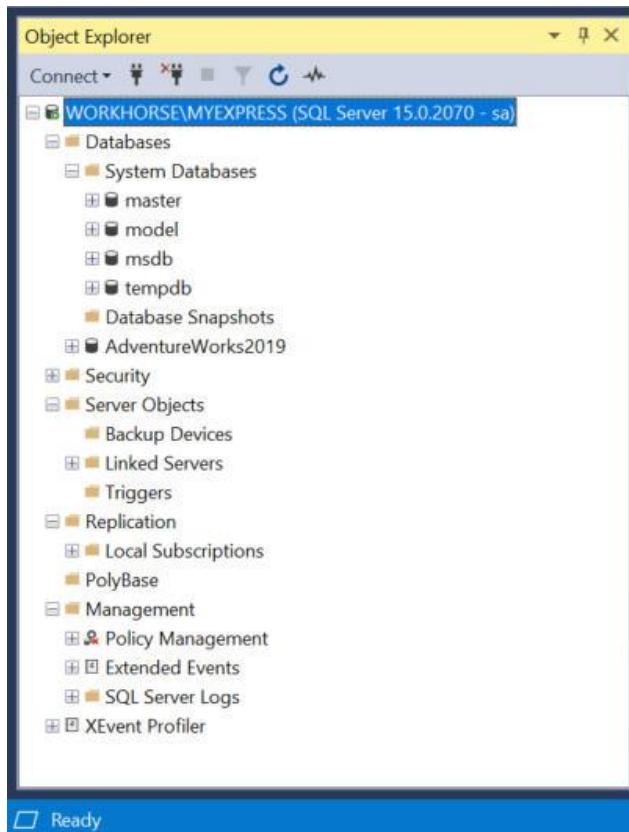
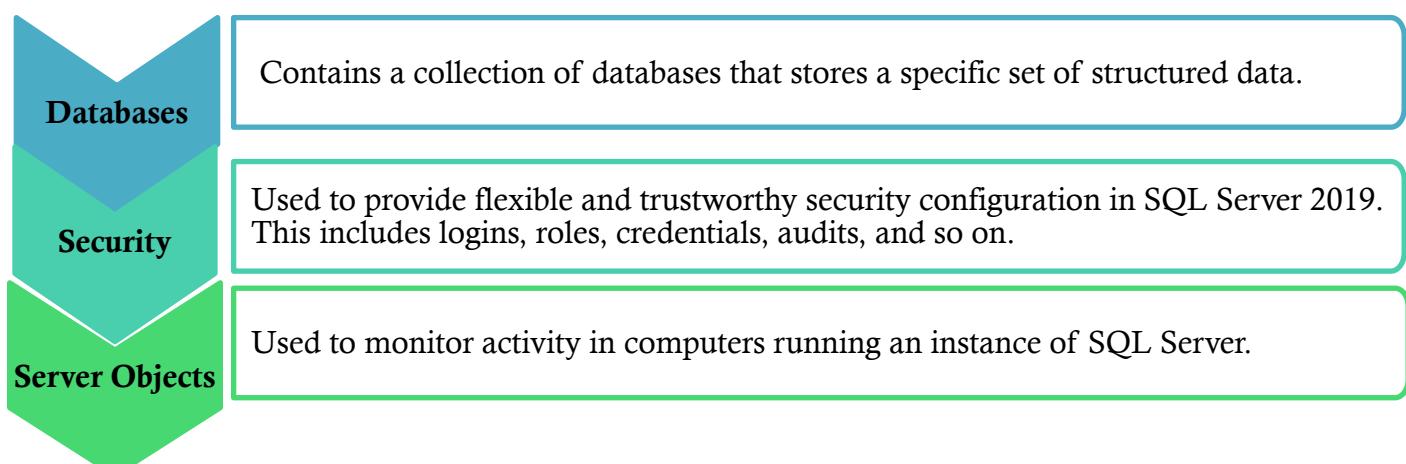
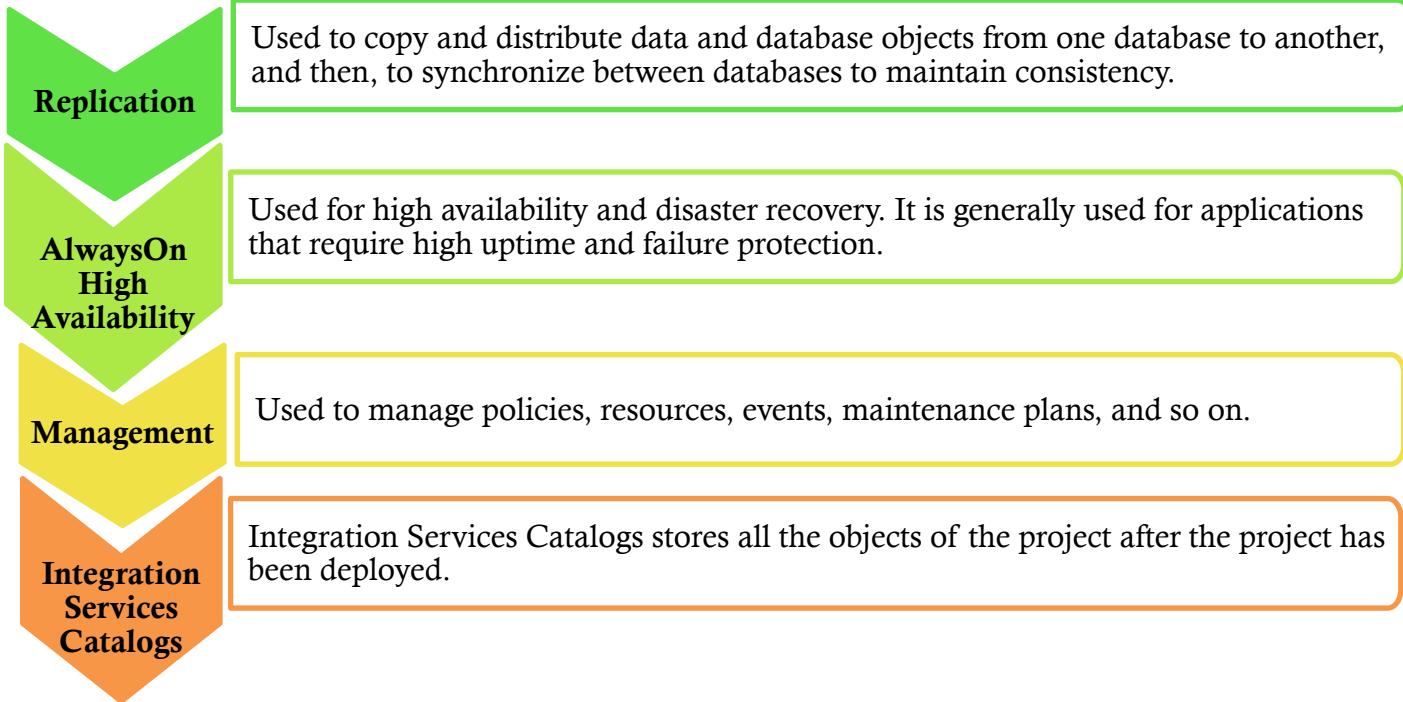


Figure 3.5: Object Explorer

The structure includes databases, security, server objects, replications, and may also show features such as AlwaysOn High Availability, Management, Integration Services Catalogs, and so on. Object Explorer can be accessed through SSMS by connecting to the database server.

Various components in the **Object Explorer** are as follows:





3.8.2 Query Window

Query window is the area where you can type Transact-SQL (T-SQL) queries. Results of your queries also appear in this window. Figure 3.6 shows an example query in the query window.

The screenshot shows the SSMS Query window with the following content:

```
SQLQuery1.sql - LA...orks2019 (sa (52))*
1 USE Adventureworks2019
2 SELECT TOP 10 Name FROM HumanResources.Shift
```

The results pane displays the following data:

Name
Day
Evening
Night

Figure 3.6: Query Window

3.9 Creating and Organizing Script Files

Script files are files that contain a set of SQL commands. A script file can contain one or more SQL statements. The script files are stored in .sql format in SQL Server.

Conceptual layers in which the script files can be organized are shown in figure 3.7.

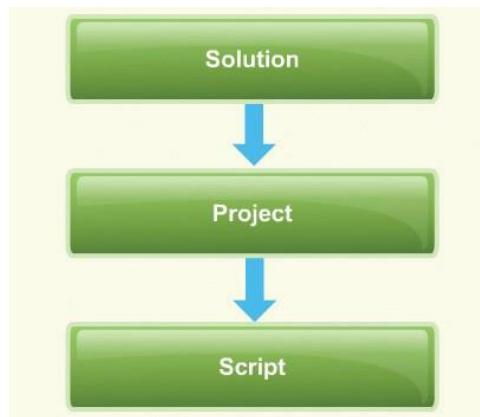


Figure 3.7: Conceptual Layers

A solution is a file in which all the projects in SQL Server are saved. This acts as a top-most node in the hierarchy. The solution file is stored as a text file with .ssmssln extension. A project comes under a solution node. There can be more than one project in SQL Server. All the data related to database connection metadata and other miscellaneous files are stored under a project. It is stored as a text file with .ssmssqlproj extension. Script files are the core files in which the queries are developed and executed. The scripts have a .sql extension.

Code Snippet 1 depicts a sample script. This script can be saved as **InsertData.sql**.

Code Snippet 1:

```
USE [AdventureWorks2019]
GO
INSERT INTO [Person].[Person]
    ([BusinessEntityID]
    ,[PersonType]
    ,[NameStyle]
    ,[Title]
    ,[FirstName]
    ,[MiddleName]
    ,[LastName]
    ,[EmailPromotion]
    ,[ModifiedDate])
VALUES(21907
    ,'EM'
    ,0
    ,'Mr.'
    ,'John'
    ,'Gareth'
    ,'Hopkins'
    ,0
    ,'2020-10-10')
GO
```

3.10 Transact-SQL Queries

Queries typed in Transact-SQL and saved as .sql files can be executed directly in the SSMS query window.

Steps to execute Transact-SQL queries are as follows:

1. In the query window, select the code to be executed.
2. On the **SSMS** toolbar, click **Execute**.
OR
On the **Query** menu, click **Execute**.
OR
Press **F5** or **Alt+X** or **Ctrl+E**.

Refer to figure 3.6 to view the execution of a sample query.

Query results can be displayed in three different formats. The three formats available are grid, text, and file view.

3.11 Check Your Progress

1. Which of the following is a command-line tool in SQL Server?

(A)	SSMS	(C)	SSMSCMD
(B)	SQLCMD	(D)	SQLSSMS

2. Which of these components of Object Explorer is used to monitor activity in computers running an instance of SQL Server?

(A)	Server Objects	(C)	Replication
(B)	Security	(D)	Integration Services

3. What does SSDT stand for?

(A)	SQL Server Deterministic Tools	(C)	SQL Server Diagnostic Tools
(B)	SQL Server Data Tools	(D)	SQL Server Database Tracking

4. Which of the following statements about the tools in SQL Server 2019 are true?

- The SQL Server Installation Center tool can be used to add, remove, and modify SQL Server programs.
- SQLCMD is an IDE used for Business Intelligence Components. It helps to design the database using Visual Studio.
- SQL Server Profiler is used to monitor an instance of the Database Engine or Analysis Services.
- SQL Server Installation Center is an application provided with SQL Server 2019 that helps to develop databases, query data, and manage the overall working of SQL Server.

(A)	a and c	(C)	b, c, and d
(B)	a, b, and c	(D)	c and d

5. Which of the following are true about these statements?

Statement 1: Script files are the core files in which the queries are developed and executed.
Statement 2: Script files are files that contain a set of SQL commands.

(A)	Both Statement 1 and 2	(C)	Only Statement 2
(B)	Only Statement 1	(D)	Neither of them

3.11.1 Answers

1.	B
2.	A
3.	B
4.	C
5.	A



Summary

- Basic architecture of SQL Server 2019 includes tools, services, and instances.
- Three major editions of SQL Server are Express, Standard, and Enterprise.
- The structure of Object Explorer includes databases, security, server objects, replications, AlwaysOn High Availability, Management, Integration Services Catalogs, and so on.
- SSMS is used to connect to SQL Server instances. SSMS is a tool used for developing, querying, and managing the databases.
- Script files should be stored in .sql format in SQL Server 2019.
- Queries typed in Transact-SQL and saved as .sql files can be executed directly into the SSMS query window.



Try It Yourself

1. Start SSMS and connect to the Database Engine of SQL Server 2019. Install the AdventureWorks2019 database. Right-click the table `HumanResources.Employee` and click **Script Table as → CREATE TO → New Query Window**. Then, right-click again and click **Select Top 1000 Rows** option.
2. Repeat this for the tables `HumanResources.EmployeeDepartmentHistory`, `Person.Person`, and `Production.Product`.

Session - 4

Transact-SQL

Welcome to the Session, **Transact-SQL**. This session explains Transact-SQL and the different categories of Transact-SQL statements. It also explains various data types and elements supported by Transact-SQL. Finally, the session explains set theory, predicate logic, and the logical order of operators in the SELECT statement.

In this session, you will learn to:

- Explain Transact-SQL
- List different categories of Transact-SQL statements
- Explain various data types supported by Transact-SQL
- Explain Transact-SQL language elements
- Explain sets and predicate logic
- Describe logical order of operators in the SELECT statement

4.1 *Introduction*

SQL is the universal language used in the database world. Most modern RDBMS products use some type of SQL dialect as their primary query language. SQL can be used to create or destroy objects such as tables on the database server and to manipulate those objects, such as adding data into them or retrieving data from them.

Transact-SQL is Microsoft's implementation of the standard SQL. Usually referred to as T-SQL, this language implements a standardized way to communicate to the database. The Transact-SQL language is an enhancement to SQL, the American National Standards Institute (ANSI) standard relational database language. It provides a comprehensive language that supports defining tables, inserting, deleting, updating, and accessing the data in the table.

4.2 *Transact-SQL*

Transact-SQL is a powerful language offering features such as data types, temporary objects, and extended stored procedures. Scrollable cursors, conditional processing, transaction control, and exception and error-handling are also some of the features which are supported by Transact-SQL.

The Transact-SQL language in SQL Server 2019 provides improved performance, increased functionality, and enhanced features. Enhancements include scalar functions, paging, sequences, meta-data discovery, and better error handling support.

Note: All the queries in this session will make use of the **AdventureWorks2019** sample database.

Code Snippet 1 shows the Transact-SQL statement, `SELECT`, which is used to retrieve `LoginID` of employees with 'Design Engineer' as the `JobTitle` from the `Employee` table.

Code Snippet 1:

```
USE AdventureWorks2019
SELECT LoginID FROM HumanResources.Employee
WHERE JobTitle = 'Design Engineer'
```

Figure 4.1 shows the result that retrieves all records of employees with 'Design Engineer' as the `JobTitle` from `HumanResources.Employee` table.

Results	
Messages	
	LoginID
1	adventure-works\gail0
2	adventure-works\jossef0
3	adventure-works\sharon0

Figure 4.1: Output of SELECT Statement

Transact-SQL includes many syntax elements that are used by or that influence most statements. These elements include data types, predicates, functions, variables, expressions, control-of-flow, comments, and batch separators.

4.3 Different Categories of Transact-SQL Statements

SQL Server supports three types of Transact-SQL statements, namely, DDL, DML, and DCL.

4.3.1 Data Definition Language (DDL)

DDL, which is usually part of a DBMS, is used to define and manage all attributes and properties of a database, including row layouts, column definitions, key columns, file locations, and storage strategy. DDL statements are used to build and modify the structure of tables and other objects such as views, triggers, stored procedures, and so on. For each object, there are usually `CREATE`, `ALTER`, and `DROP` statements (such as, `CREATE TABLE`, `ALTER TABLE`, and `DROP TABLE`).

Most DDL statements take the following form, where `object_name` can be a table, view, trigger, stored procedure, and so on:

- `CREATE object_name`
- `ALTER object_name`
- `DROP object_name`

4.3.2 Data Manipulation Language (DML)

DML is used to select, insert, update, or delete data in the objects defined with DDL. All database users can use these commands during the routine operations on a database.

Different DML statements are as follows:

- SELECT statement
- INSERT statement
- UPDATE statement
- DELETE statement

4.3.3 Data Control Language (DCL)

Data is an important part of a database, so proper steps should be taken to check that no invalid user accesses the data. Data control language is used to control permissions on database objects. Permissions are controlled by using the GRANT, REVOKE, and DENY statements. DCL statements are also used for securing the database. The three basic DCL statements are as follows:

- GRANT statement
- REVOKE statement
- DENY statement

4.4 Data Types

A data type is an attribute defining the type of data that an object can contain. Data types must be provided for columns, parameters, variables, and functions that return data values, and stored procedures that have a return code. Transact-SQL includes a number of base data types, such as `varchar`, `text`, and `int`. All data that is stored in SQL Server must be compatible with one of the base data types.

Following objects have data types:

- Columns present in tables and views
- Parameters in stored procedures
- Variables
- Transact-SQL functions that return one or more data values of a specific data type
- Stored procedures that have a return code belonging to the integer data type

Various items in SQL Server 2019 such as columns, variables, and expressions are assigned data types. SQL Server 2019 supports three kinds of data types:

- **System-defined data types**

These data types are provided by SQL Server 2019. Table 4.1 shows the commonly used system-defined data types of SQL Server.

Category	Data Type	A Column of This Type
Exact Numerics	int	Occupies four bytes of memory space. Is typically used to hold integer values. Can hold integer data from -2^{31} (-2,147,483,648) to $2^{31}-1$ (2,147,483,647).
	smallint	Occupies two bytes of memory space. Can hold integer data from -32,768 to 32,767.
	tinyint	Occupies one byte of memory space. Can hold integer data from 0 to 255.
	bigint	Occupies 8 bytes of memory space. Can hold data in the range -2^{63} (-9,223,372,036,854,775,808) to $2^{63}-1$ (9,223,372,036,854,775,807).
	numeric	Has fixed precision and scale.
Approximate Numerics	money	Occupies eight bytes of memory space. Represents monetary data values ranging from $-2^{63}/10000$ (-922,337,203,685,477.5808) to $2^{63}-1$ (922,337,203,685,477.5807).
	float	Occupies eight bytes of memory space. Represents floating point number ranging from $-1.79E+308$ through $1.79E+308$.
Date and Time	real	Occupies four bytes of memory space. Represents floating precision number ranging from $-3.40E+38$ through $3.40E+38$.
	datetime	Represents date and time. Stored as two 4-byte integers.
Character String	smalldatetime	Represents date and time.
	char	Stores character data that is fixed-length and non-Unicode.
	varchar	Stores character data that is variable-length and non-Unicode with a maximum of 8,000 characters.
Unicode Types	text	Stores character data that is variable-length and non-Unicode with a maximum length of $2^{31}-1$ (2,147,483,647) characters.
	nchar	Stores Unicode character data of fixed-length.
Other Data Types	nvarchar	Stores variable-length Unicode character data.
	timestamp	Occupies 8 bytes of memory space. Can hold automatically generated, unique binary numbers that are generated for a database.
	binary(n)	Stores fixed-length binary data with a maximum length of 8000 bytes.
	varbinary(n)	Stores variable-length binary data with a maximum length of 8000 bytes.
	image	Stores variable-length binary data with a maximum length of $2^{30}-1$ (1,073,741,823) bytes.
	uniqueidentifier	Occupies 16 bytes of memory space. Also, stores a globally unique identifier (GUID).

Table 4.1: Commonly Used System-Defined Data Types

➤ Alias data types

These are based on the system-supplied data types. Alias data types are used when more than one table stores the same type of data in a column and has similar characteristics such as length, nullability, and type. In such cases, an alias data type can be created that can be used commonly by all these tables.

Alias data types can be created using the `CREATE TYPE` statement. The syntax for the `CREATE TYPE` statement is as follows:

Syntax:

```
CREATE TYPE [ schema_name. ] type_name { FROM base_type [ ( precision [, scale ] ) ] [ NULL | NOT NULL ] } [ ; ]
```

where,

`schema_name`: identifies name of the schema in which the alias data type is being created. A schema is a collection of objects such as tables, views, and so forth in a database.

`type_name`: identifies name of the alias type being created.

`base_type`: identifies name of the system-defined data type based on which the alias data type is being created.

`precision` and `scale`: specify precision and scale for numeric data.

`NULL` | `NOT NULL`: specifies whether the data type can hold a null value or not.

Code Snippet 2 shows how to create an alias data type using `CREATE TYPE` statement.

Code Snippet 2:

```
CREATE TYPE usertype FROM varchar(20) NOT NULL
```

In the code, the built-in data type `varchar` is stored as a new data type named `usertype` by using the `CREATE TYPE` statement.

➤ User-defined types

These are created using programming languages supported by the .NET Framework.

4.5 Transact-SQL Language Elements

The Transact-SQL language elements are used in SQL Server 2019 for working on the data that is entered in SQL Server database. The Transact-SQL language elements includes predicates, operators, functions, variables, expressions, control-of-flow, errors, and transactions, comments, and batch separators.

4.5.1 Predicates and Operators

Predicates are used to evaluate whether an expression is TRUE, FALSE, or UNKNOWN.

Some of the predicates available in Transact-SQL are as follows:

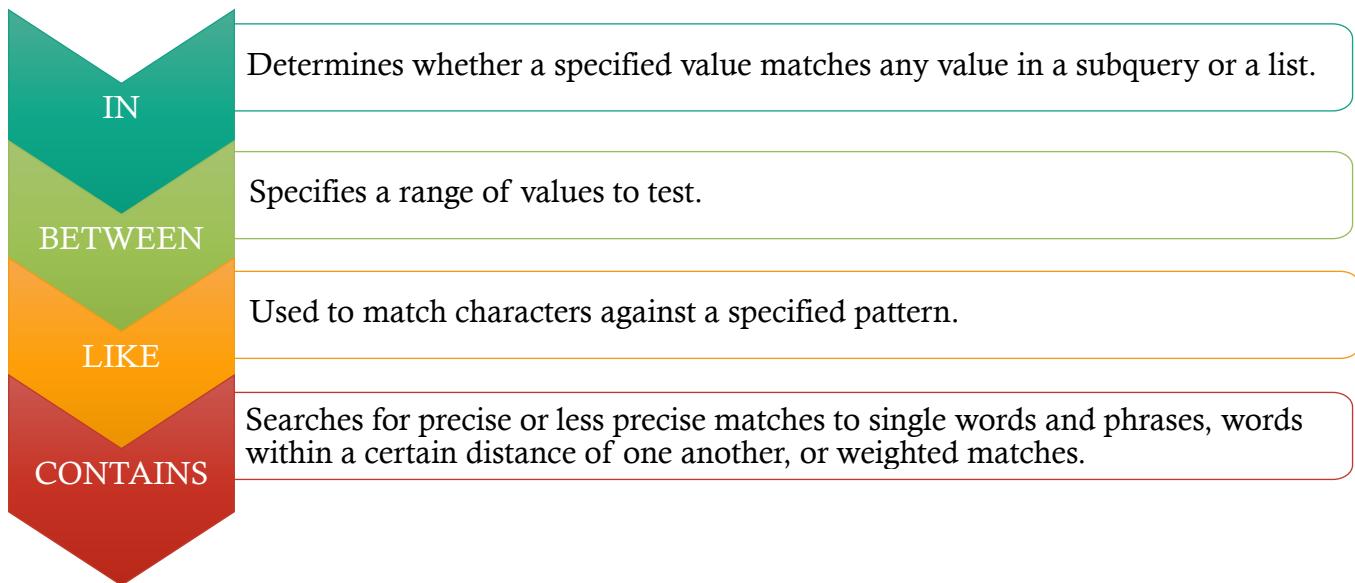


Table 4.2 shows some examples of the predicates.

Predicate	Example
IN	SELECT PersonType, Title, FirstName, LastName FROM AdventureWorks2019.Person.Person WHERE PersonType IN ('EM', 'SC')
BETWEEN	SELECT BusinessEntityID, NationalIDNumber, LoginID, JobTitle, HireDate FROM AdventureWorks2019.HumanResources.Employee WHERE HireDate BETWEEN '01-01-2010' AND '01-01-2013'
LIKE	SELECT DepartmentID, Name,GroupName,ModifiedDate FROM AdventureWorks2019.HumanResources.Department WHERE Name LIKE 'P%'
CONTAINS	SELECT * FROM AdventureWorks2019.Person.Address WHERE CONTAINS(AddressLine1, 'Street')
	Note that this statement will work only on a table with full text index.

Table 4.2: Predicate Examples

Operators are used to perform arithmetic, comparison, concatenation, or assignment of values. For example, data can be tested to verify that the **COUNTRY** column for the customer data is populated (or has a NOT NULL value). In queries, anyone who can see the data in the table requiring an operator can perform operations. Appropriate permissions are required before data can be successfully changed. SQL Server has seven categories of operators. Table 4.3 describes the different operators supported in SQL Server 2019.

Operator	Description	Example
Comparison	Compares a value against another value or an expression	=, <, >, >=, <=, !=, !=>
Logical	Tests for the truth of a condition	AND, OR, NOT
Arithmetic	Performs arithmetic operations such as addition, subtraction, multiplication, and division	+, -, *, /, %

Operator	Description	Example
Concatenation	Combines two strings into one string	+
Assignment	Assigns a value to a variable	=

Table 4.3: Operators

Table 4.4 shows the precedence of predicates and operators.

Order	Operators
1	() Parentheses
2	*, /, %
3	+, -
4	=, <, >, >=, <=, !=, !>
5	NOT
6	AND
7	BETWEEN, IN, CONTAINS, LIKE, OR
8	=

Table 4.4: Precedence of Predicates and Operators

Code Snippet 3 shows execution of operators according to precedence.

Code Snippet 3:

```
DECLARE @Number int;
SET @Number = 2 + 2 * (4 + (5 - 3))
SELECT @Number
```

Here, the steps to arrive at the result are as follows:

1. $2 + 2 * (4 + (5 - 3))$
2. $2 + 2 * (4 + 2)$
3. $2 + 2 * 6$
4. $2 + 12$
5. 14

Hence, the code will display 14.

4.5.2 Functions

A function is a set of Transact-SQL statements that is used to perform some task. Transact-SQL includes a large number of functions. These functions can be useful when data is calculated or manipulated. In SQL, functions work on the data, or group of data, to return a required value. They can be used in a `SELECT` list or anywhere in an expression.

The four types of functions in SQL Server 2019 are as follows:

Rowset functions

In Transact-SQL, the rowset function is used to return an object that can be used in place of a table reference. For example, OPENDATASOURCE, OPENQUERY, OPENROWSET, and OPENXML are rowset functions.

Aggregate functions

Transact-SQL provides aggregate functions to assist with the summarization of large volumes of data. For example, SUM, MIN, MAX, AVG, COUNT, COUNTBIG, and so on are aggregate functions.

Ranking functions

Many tasks, such as creating arrays, generating sequential numbers, finding ranks, and so on can be implemented in an easier and faster way by using ranking functions. For example, RANK, DENSE_RANK, NTILE, and ROW_NUMBER are ranking functions.

Scalar functions

In scalar functions, the input is a single value and the output received is also a single value.

Table 4.5 shows the commonly used scalar functions in SQL.

Function Type	Description	Example
Conversion function	The conversion function is used to transform a value of one data type to another. Additionally, it can be used to obtain a variety of special date formats.	CONVERT
Date and time function	Date and time functions are used to manipulate date and time values. They are useful to perform calculations based on time and dates.	GETDATE, SYSDATETIME, GETUTCDATE, DATEADD, DATEDIFF, YEAR, MONTH, DAY
Mathematical function	Mathematical functions perform algebraic operations on numeric values.	RAND, ROUND, POWER, ABS, CEILING, FLOOR
System function	SQL Server provides system functions for returning metadata or configuration settings.	HOST_ID, HOST_NAME, ISNULL
String function	String functions are used for string inputs such as char and varchar. The output can be a string or a numeric value.	SUBSTRING, LEFT, RIGHT, LEN, DATALENGTH, REPLACE, REPLICATE, UPPER, LOWER, RTRIM, LTRIM

Table 4.5: Scalar Functions

There are also other scalar functions such as cursor functions, logical functions, metadata functions, security functions, and so on that are available in SQL Server 2019.

4.5.3 Variables

A variable is an object that can hold a data value. In Transact-SQL, variables can be classified into local and global variables.

In Transact-SQL, local variables are created and used for temporary storage while SQL statements are executed. Data can be passed to SQL statements using local variables. The name of a local variable must be prefixed with '@' sign.

For example,

```
DECLARE @SearchWord NVARCHAR(30)
SET @SearchWord = N'performance'
```

In earlier versions of SQL Server, a concept called global variables existed which referred to in-built variables that are defined and maintained by the system. In SQL Server 2019, they are categorized as functions. They are prefixed with two '@' signs. The return value of these functions can be retrieved with a simple SELECT query.

For example,

```
SELECT @@LANGUAGE as 'Language'
```

This returns the language currently used in SQL Server, such as US English.

Some of the functions include:

```
@@DATEFIRST
@@LANGUAGE
@@LOCK_TIMEOUT
@@MAX_CONNECTIONS
@@MAX_PRECISION
@@SERVERNAME
@@VERSION
```

4.5.4 Expressions

An expression is a combination of identifiers, values, and operators that SQL Server can evaluate in order to obtain a result. Expressions can be used in several different places when accessing or changing data.

Code Snippet 4 shows an expression that operates on a column to add an integer to the results of the YEAR function on a `datetime` column.

Code Snippet 4:

```
USE AdventureWorks2019
SELECT SalesOrderID, CustomerID, SalesPersonID, TerritoryID, YEAR(OrderDate) AS CurrentYear, YEAR(OrderDate) + 1 AS NextYear
FROM Sales.SalesOrderHeader
```

Figure 4.2 shows results of the expression.

	SalesOrderID	CustomerID	SalesPersonID	TerritoryID	CurrentYear	NextYear
1	43659	29825	279	5	2011	2012
2	43660	29672	279	5	2011	2012
3	43661	29734	282	6	2011	2012
4	43662	29994	282	6	2011	2012
5	43663	29565	276	4	2011	2012
6	43664	29898	280	1	2011	2012
7	43665	29580	283	1	2011	2012
8	43666	30052	276	4	2011	2012
9	43667	29974	277	3	2011	2012
10	43668	29614	282	6	2011	2012
11	43669	29747	283	1	2011	2012

Figure 4.2: Expression Result

4.5.5 Control-of-Flow, Errors, and Transactions

Although Transact-SQL is primarily a data retrieval language, it supports control-of-flow statements for executing and finding errors. Control-of-flow language determines the execution flow of Transact-SQL statements, statement blocks, user-defined functions, and stored procedures.

Table 4.6 shows some of the commonly used control-of-flow statements in Transact-SQL.

Control-of-Flow Statement	Description
IF . . . ELSE	Provides branching control based on a logical test.
WHILE	Repeats a statement or a block of statements as long as the condition is true.
BEGIN . . . END	Defines the scope of a block of Transact-SQL statements.
TRY . . . CATCH	Defines the structure for exception and error handling.
BEGIN TRANSACTION	Marks a block of statements as part of an explicit transaction.

Table 4.6: Control-of-Flow Statements

Code Snippet 5 displays an example using IF...ELSE.

Code Snippet 5:

```
IF DATENAME(weekday, GETDATE()) IN (N'Saturday', N'Sunday')
    SELECT 'It is a Weekend';
ELSE
    SELECT 'It is a Weekday';
```

Here, the weekday is retrieved from the current date and checked to see if it is either a Saturday or a Sunday and then, accordingly a suitable message is displayed.

4.5.6 Comments

Comments are descriptive text strings, also known as remarks, in program code that will be ignored by the compiler. Comments can be included inside the source code of a single statement, a batch, or a stored

procedure. Comments explain the purpose of the program, special execution conditions, and provide revision history information.

SQL Server supports two types of commenting styles:

-- (double hyphens)

A complete line of code or part of a code can be marked as a comment, if two hyphens (--) are placed at the beginning. The remainder of the line becomes a comment.

/* ... */ (forward slash-asterisk character pairs)

These comment characters can be used on the same line as code to be executed, on lines by themselves, or even within executable code. Everything between /* to */ is considered part of the comment. For a multiple-line comment, the open-comment character pair /*) must begin the comment, and the close-comment character pair */) must end the comment.

Code Snippet 6 displays the use of -- (double hyphens) style of comment.

Code Snippet 6:

```
USE AdventureWorks2019
-- HumanResources.Employee table contains the details of an employee.
-- This statement retrieves all the rows of the table
-- HumanResources.Employee.
SELECT * FROM HumanResources.Employee
```

Code Snippet 7 displays the use of /* ... */ (forward slash-asterisk character pairs) style of comment.

Code Snippet 7:

```
USE AdventureWorks2019
/* HumanResources.Employee table contains the details of an employee.
This statement retrieves all the rows of the table HumanResources.Employee. */
SELECT * FROM HumanResources.Employee
```

4.5.7 Batch Separators

A batch is a collection of one or more Transact-SQL statements sent at one time from an application to SQL Server for execution. The Transact-SQL statements in a batch are compiled into a single executable unit, called an execution plan. The statements in the execution plan are then executed one at a time. The process wherein a set of commands are processed one at a time from a batch of commands is called batch processing.

A batch separator is handled by SQL Server client tools such as SSMS to execute commands. For example, you must specify GO as a batch separator in SSMS.

An example of a batch statement is given in Code Snippet 8.

Code Snippet 8:

```
USE AdventureWorks2019  
SELECT * FROM HumanResources.Employee  
GO
```

In Code Snippet 8, the two statements will be grouped into one execution plan, but executed one statement at a time. The `GO` keyword signals the end of a batch.

4.6 Sets and Predicate Logic

Sets and Predicate Logic are the two mathematical fundamentals that are used in SQL Server 2019. Both these theories are used in querying of data in SQL Server 2019.

4.6.1 Set Theory

Set theory is a mathematical foundation used in relational database model. A set is a collection of distinct objects considered as a whole. For example, all the employees under an `Employee` table can be considered as one set. Employees are different objects that form a part of the set in the `Employee` table.

Table 4.7 shows different applications in the set theory and their corresponding application in SQL Server queries.

Set Theory Applications	Application in SQL Server Queries
Act on the whole set at once.	Query the whole table at once.
Use declarative, set-based processing.	Use attributes in SQL Server to retrieve specific data.
Elements in the set must be unique.	Define unique keys in the table.
No sorting instructions.	The results of querying are not retrieved in any order.

Table 4.7: Set Theory Applications

One of the common Set operators is the `INTERSECT` operator. It returns distinct rows that are produced by both the left and right input queries operator.

Code Snippet 9 shows an example using the `INTERSECT` operator.

Code Snippet 9:

```
USE AdventureWorks2019  
GO  
SELECT ProductID  
FROM Production.Product  
INTERSECT  
SELECT ProductID  
FROM Production.WorkOrder ;
```

The outcome will show 238 rows of products that have work orders.

4.6.2 Predicate Logic

Predicate logic is a mathematical framework that consists of logical tests that gives a result. The results are always displayed as either true or false. In Transact-SQL, expressions such as WHERE and CASE expressions are based on predicate logic. Predicate logic is also used in other situations in Transact-SQL.

Some of the applications of predicate logic in Transact-SQL are as follows:

Enforcing data integrity using the CHECK constraint

Control-of-flow using the IF statement

Joining tables using the ON filter

Filtering data in queries using the WHERE and HAVING clause

Providing conditional logic to CASE expressions

Defining subqueries

4.7 Logical Order of Operators in SELECT Statement

Along with syntax of different SQL Server elements, an SQL Server user must also know the process of how the entire query is executed. This process is a logical process that breaks the query and executes the query according to a predefined sequence in SQL Server 2019. The SELECT statement is a query that will be used to explain the logical process of query execution.

Following is the syntax of the SELECT statement.

Syntax:

```
SELECT <select list>
FROM <table source>
WHERE <search condition>
GROUP BY <group by list>
HAVING <search condition>
ORDER BY <order by list>
```

Table 4.8 explains the elements of the SELECT statement.

Element	Description
SELECT <select list>	Defines the columns to be returned
FROM <table source>	Defines the table to be queried

Element	Description
WHERE <search condition>	Filters the rows by using predicates
GROUP BY <group by list>	Arranges the rows by groups
HAVING <search condition>	Filters the groups using predicates
ORDER BY <order by list>	Sorts the output

Table 4.8: Elements of SELECT Statement

Code Snippet 10 shows a SELECT statement.

Code Snippet 10:

```
USE AdventureWorks2019
SELECT SalesPersonID, YEAR(OrderDate) AS OrderYear FROM
Sales.SalesOrderHeader
WHERE CustomerID = 30084
GROUP BY SalesPersonID, YEAR(OrderDate)
HAVING COUNT(*) > 1
ORDER BY SalesPersonID, OrderYear;
```

In the example, the order in which SQL Server will execute the SELECT statement is as follows:

1. First, the `FROM` clause is evaluated to define the source table that will be queried.
2. Next, the `WHERE` clause is evaluated to filter the rows in the source table. This filtering is defined by the predicate mentioned in the `WHERE` clause.
3. After this, the `GROUP BY` clause is evaluated. This clause arranges the filtered values received from the `WHERE` clause.
4. Next, the `HAVING` clause is evaluated based on the predicate that is provided.
5. Next, the `SELECT` clause is executed to determine the columns that will appear in the query results.
6. Finally, the `ORDER BY` statement is evaluated to display the output.

The order of execution for the SELECT statement in Code Snippet 10 would be as follows:

5. `SELECT SalesPersonID, YEAR(OrderDate) AS OrderYear`
1. `FROM SalesOrderHeader`
2. `WHERE CustomerID = 30084`
3. `GROUP BY SalesPersonID, YEAR(OrderDate)`
4. `HAVING COUNT(*) > 1`
6. `ORDER BY SalesPersonID, OrderYear;`

Figure 4.3 shows the result of the SELECT statement.

Results		Messages
	SalesPersonID	OrderYear
1	279	2011
2	279	2013

Figure 4.3: SELECT Statement Result

4.8 Check Your Progress

1. Which of the following is used to define and manage all attributes and properties of a database, including row layouts, column definitions, key columns, file locations, and storage strategy?

(A) DDL	(C) DCL
(B) DML	(D) DPL

2. Which of the following is not used in DCL?

(A) GRANT statement	(C) UPDATE statement
(B) REVOKE statement	(D) DENY statement

3. Which of the following specifies a range of values to test?

(A) IN	(C) LIKE
(B) BETWEEN	(D) CONTAINS

4. Match the following.

Control-of-Flow Statement	Description
a. IF... ELSE	1. Marks a block of statements as part of an explicit transaction.
b. WHILE	2. Defines the structure for exception and error handling.
c. BEGIN... END	3. Repeats a statement or a block of statements when the condition is true.
d. TRY... CATCH	4. Defines the scope of a block of Transact-SQL statements.
e. BEGIN TRANSACTION	5. Provides branching control based on a logical test.

(A) a-4, b-2, c-3, d-1, e-5	(C) a-1, b-4, c-5, d-3, e-2
(B) a-1, b-2, c-4, d-3, e-5	(D) a-5, b-3, c-4, d-2, e-1

5. Which of the following are the two mathematical fundamentals that are used in SQL Server 2019?

(A)	Fractions and Sets	(C)	Predicate Logic and Fractions
(B)	Sets and Predicate Logic	(D)	Probability and Fractions

6. Which of the following will be the result of the code?

```
SET @Number = 2 * (4 + 5) + 2 * (4 + (5 - 3))
```

(A)	120	(C)	30
(B)	62	(D)	26

4.8.1 Answers

1.	A
2.	C
3.	B
4.	D
5.	B
6.	C



Summary

- Transact-SQL is a powerful language which offers features such as data types, temporary objects, and extended stored procedures.
- SQL Server supports three types of Transact-SQL statements, namely, DDL, DML, and DCL.
- A data type is an attribute defining the type of data that an object can contain.
- The Transact-SQL language elements includes predicates, operators, functions, variables, expressions, control-of-flow, errors, and transactions, comments, and batch separators.
- Sets and Predicate Logic are the two mathematical fundamentals that are used in SQL Server 2019.
- Set theory is a mathematical foundation used in relational database model, where a set is a collection of distinct objects considered as a whole.
- Predicate logic is a mathematical framework that consists of logical tests that gives a result.



Try It Yourself

1. Use the Query Editor to execute a query. Ensure that a connection to a new server instance is established.

Then, using the AdventureWorks2019 database, create and execute following queries:

- a) Display all the territory names from the table Sales.SalesTerritory.
 - b) Display all the product descriptions from the table Production.ProductDescription.
 - c) Display the data in columns ProductID, Name, and ProductNumber from Production.Product table.
 - d) Display the bonus, commission percentage, and sales made in the previous year from the table Sales.SalesPerson for the only those territories that have TerritoryID between 1 and 3.
2. You are given the following code that adds five days to a given date:

```
SET NOCOUNT ON
DECLARE @startdate DATETIME, @adddays INT;
SET @startdate = 'January 10, 1900 12:00 AM';
SET @adddays = 5;
SET NOCOUNT OFF;
SELECT @startdate + 1.25 AS 'Start Date',
       @startdate + @adddays AS 'Add Date';
```

Create Transact-SQL code that will subtract six days from a given date.

3. Use comparison operators to perform the following tasks:
 - a) Display all the records from Person.Address table having city as Montreal.
 - b) Display all the records from HumanResources.Department table that have a value in DepartmentID that is greater than or equal to the value 13.
 - c) Display all the records from Production.ProductCategory table that do not have value in ProductCategoryID that is equal to the value 3 or the value 2.
4. Display all the records from Person.Person for all persons whose last name begins with B.

Session - 5

Creating and Managing Databases

Welcome to the Session, **Creating and Managing Databases**. This session describes system and user defined databases. It also lists the key features of the AdventureWorks2019 database. Finally, the session describes different types of database modification.

In this session, you will learn to:

- Explain about modification of system data
- Describe adding of filegroups and transaction logs
- Outline the process to create a database
- Describe how to drop a database
- Explain database snapshots

5.1 *Modifying System Data*

Users are not allowed to directly update the information in system database objects, such as system tables, system stored procedures, and catalog views. However, users can avail a complete set of administrative tools allowing them to fully administer the system and manage all users and database objects. These are as follows:

SSMS Administration utilities: From SQL Server 2005 onwards, several SQL Server administrative utilities are integrated into SSMS. It is the core administrative console for SQL Server installations. It enables to perform high-level administrative functions, schedule routine maintenance tasks, and so forth.

SQL Server Management Objects (SQL-SMO) API: Includes complete functionality for administering SQL Server in applications.

Transact-SQL scripts and stored procedures: These use system stored procedures and Transact- SQL DDL statements. Figure 5.1 shows a Transact-SQL query window.

```

SQLQuery1.sql - WO...S.Smaster (sa (51)) * X
1  /**** Script for SelectTopNRows command from SSMS ****/
2  SELECT TOP 1000 [AddressID]
3  ,[AddressLine1]
4  ,[AddressLine2]
5  ,[City]
6  ,[StateProvinceID]
7  ,[PostalCode]
8  ,[SpatialLocation]
9  ,[rowguid]
10 ,[ModifiedDate]
11 FROM AdventureWorks2019.Person.Address

```

	AddressID	AddressLine1	AddressLine2	City	StateProvinceID	PostalCode	SpatialLocation
1	1	1970 Napa Ct.	NULL	Bothell	79	98011	0xE6100000010CAE8BFC28BCE4474067A89189898A5EC0
2	2	9833 Mt. Dias Blv.	NULL	Bothell	79	98011	0xE6100000010CD6FA851AE6D74740BC262AOA03905EC0
3	3	7484 Roundtree Drive	NULL	Bothell	79	98011	0xE6100000010C18E304C4ADE14740DA930C7893915EC0
4	4	9539 Glenside Dr	NULL	Bothell	79	98011	0xE6100000010C813A0D5F9FDE474011A5C28A7C955EC0
5	5	1226 Shoe St.	NULL	Bothell	79	98011	0xE6100000010C61C64D8ABBD94740C460EA3FD8855EC0
6	6	1399 Firestone Drive	NULL	Bothell	79	98011	0xE6100000010CE0B4E50458DA47402F12A5F80C975EC0
7	7	5672 Hale Dr.	NULL	Bothell	79	98011	0xE6100000010C18E304C4ADE1474011A5C28A7C955EC0
8	8	6387 Scenic Avenue	NULL	Bothell	79	98011	0xE6100000010C0029A5D93BDF4740E248962FD5975EC0
9	9	8713 Yosemite Ct.	NULL	Bothell	79	98011	0xE6100000010C6A80AD742DC4740851574F7198C5EC0
10	10	250 Race Court	NULL	Bothell	79	98011	0xE6100000010C219D64AE1FE4474040862564B7825EC0
11	11	1318 Lasalle Street	NULL	Bothell	79	98011	0xE6100000010CCEFFBBFD9FE9474057B9479C728E5EC0

Figure 5.1: Transact-SQL Query Window

These tools also guard applications from making changes in the system objects.

5.2 Viewing System Database Data

Database applications can determine catalog and system information by using any of these approaches:

System Catalog Views

Views displaying metadata for describing database objects in an SQL Server instance.

SQL-SMO

New managed code object model, providing a set of objects used for managing Microsoft SQL Server.

Catalog Functions, Methods, Attributes, or Properties of Data API

Used in ActiveX Data Objects (ADO), OLE DB, or ODBC applications.

Stored Procedures and Functions

Used in Transact-SQL as stored procedures and built-in functions.

5.3 User-defined Databases

Using SQL Server 2019, users can create their own databases, also called user-defined databases, and work with them. The purpose of these databases is to store user data.

5.3.1 Creating Databases Using Transact-SQL

To create a user-defined database, the information required is as follows:

- Name of the database
- Owner or creator of the database
- Size of the database
- Files and filegroups used to store it

Following is the syntax to create a user-defined database.

Syntax:

```
CREATE DATABASE DATABASE_NAME  
[ ON  
[ PRIMARY ] [ <filespec> [ ,...n ]  
[ , <filegroup> [ ,...n ] ]  
[ LOGON { <filespec> [ ,...n ] } ]  
]  
[ COLLATE collation_name ]  
]  
[ ; ]
```

where,

`DATABASE_NAME`: is the name of the database to be created.

`ON`: indicates the disk files to be used to store the data sections of the database and data files.

`PRIMARY`: is the associated `<filespec>` list defining the primary file.

`<filespec>`: controls the file properties.

`<filegroup>`: controls filegroup properties.

`LOG ON`: indicates disk files to be used for storing the database log and log files.

`COLLATE collation_name`: is the default collation for the database. A collation defines rules for comparing and sorting character data based on the standard of particular language and locale. Collation name can be either a Windows collation name or a SQL collation name.

Code Snippet 1 shows how to create a database with database file and transaction log file with collation name.

Code Snippet 1:

```
CREATE DATABASE [Customer_DB] ON PRIMARY  
( NAME = 'Customer_DBX', FILENAME = 'C:\Program Files\Microsoft SQL  
Server\MSSQL15.MYEXPRESS\MSSQL\DATA\Customer_DB.mdf')  
LOG ON  
( NAME = 'Customer_DB_log', FILENAME = 'C:\Program Files\Microsoft SQL  
Server\MSSQL15.MYEXPRESS\MSSQL\DATA\Customer_DB_log.ldf')  
COLLATE SQL_Latin1_General_CI_AS
```

After executing the code in Code Snippet 1, SQL Server 2019 displays the message 'Command(s) completed successfully'.

SQL Server databases use two files - an **.mdf** file, known as the primary database file, containing the schema and data and a **.ldf** file, which contains the logs. A database may also use secondary database file, which normally uses an **.ndf** extension.

MDF stands for Master Database File. It contains main information of a database that are part of the server. This extension also points to various other files. It plays a crucial role in information storage.

LDF stands for Log Database File. This file stores information related to transaction logs for main data file. It basically keeps track of changes that have been made in the database.

Besides these, there are **.ndf** files representing secondary data files. Secondary data files make up all the data files, other than the primary data file. Some databases may not have any secondary data files, while others have several secondary data files. The recommended file name extension for secondary data files is **.ndf**.

Note: SQL Server does not enforce the **.mdf**, **.ndf**, and **.ldf** file name extensions, but these extensions are recommended to help identify the use of the file.

Figure 5.2 shows the query and the database **Customer_DB** listed in the **Object Explorer**.

The screenshot shows the SSMS interface. On the left, the Object Explorer pane displays a tree structure of database objects under 'WORKHORSE\MYEXPRESS'. A red box highlights the 'Customer_DB' node under 'Databases'. In the center, a query results window titled 'SQLQuery1.sql - WO...S.master (sa (51))' shows the following SQL code:

```
1 CREATE DATABASE [Customer_DB] ON PRIMARY
2   ( NAME = 'Customer_DB',
3     FILENAME = 'C:\Program Files\Microsoft SQL Server\MYSQL15.MYEXPRESS\MYSQL\DATA\Customer_DB.mdf'
4   ) LOG ON
5   ( NAME = 'Customer_DB_log',
6     FILENAME = 'C:\Program Files\Microsoft SQL Server\MYSQL15.MYEXPRESS\MYSQL\DATA\Customer_DB_log.ldf'
7   ) COLLATE SQL_Latin1_General_CI_AS
```

Below the code, the 'Messages' pane shows the output: 'Commands completed successfully.' and 'Completion time: 2020-10-07T22:34:02.4374063+05:30'. At the bottom, a status bar indicates 'Query executed successfully.', 'WORKHORSE\MYEXPRESS (15.0 RTM)', 'sa (51)', 'master', '00:00:00', and '0 rows'.

Figure 5.2: Customer_DB Database

5.3.2 Modifying Databases

As a user-defined database grows or diminishes, the database size will be expanded or be shrunk automatically or manually. Based on changing requirements from time to time, it may be found necessary to modify a database.

Following is the syntax to modify a database:

Syntax:

```
ALTER DATABASE database_name
{
<add_or_modify_files>
| <add_or_modify_filegroups>
| <set_database_options>
| MODIFY NAME = new_database_name
| COLLATE collation_name
}
[ ; ]
```

where,

`database_name`: is the original name of the database.

`MODIFY NAME = new_database_name`: is the new name of the database to which it is to be renamed.

`COLLATE collation_name`: is the collation name of the database.

`<add_or_modify_files>`: is the file to be added, removed, or modified.

`<add_or_modify_filegroups>`: is the filegroup to be added, modified, or removed from the database.

`<set_database_options>`: is the database-level option influencing the characteristics of the database that can be set for each database. These options are unique to each database and do not affect other databases.

Code Snippet 2 shows how to rename a database `Customer_DB` with a new database name, `CUST_DB`.

Code Snippet 2:

```
ALTER DATABASE Customer_DB MODIFY NAME = CUST_DB
```

Figure 5.3 shows database `Customer_DB` is renamed with a new database name, `CUST_DB`.

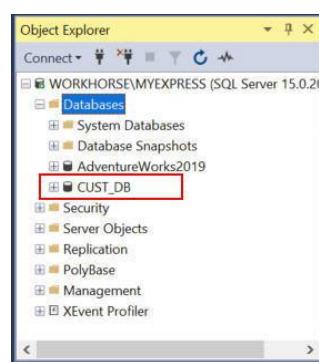


Figure 5.3: New Database Name `CUST_DB`

5.3.3 Ownership of Databases

In SQL Server 2019, the ownership of a user-defined database can be changed. Ownership of system databases cannot be changed. The system procedure `sp_changedbowner` is used to change the ownership of a database.

The syntax is as follows:

Syntax:

```
sp_changedbowner [ @loginname = ] 'login'
```

where,

login is an existing database username.

After `sp_changedbowner` is executed, the new owner is known as the `dbo` user inside the selected database. The `dbo` receives permissions to perform all activities in the database. The owner of the `master`, `model`, or `tempdb` system databases cannot be changed.

Code Snippet 3, when executed, makes the login '`sa`' the owner of the current database and maps '`sa`' to existing aliases that are assigned to the old database owner, and will display 'Command(s) completed successfully'.

Code Snippet 3:

```
USE CUST_DB  
EXEC sp_changedbowner 'sa'
```

5.3.4 Setting Database Options

Database-level options determine the characteristics of the database that can be set for each database. These options are unique to each database, so they do not affect other databases. These database options are set to default values when a database is first created, and can then, be changed by using the `SET` clause of the `ALTER DATABASE` statement.

Table 5.1 shows the database options that are supported by SQL Server 2019.

Option Type	Description
Automatic options	Controls automatic behavior of database.
Cursor options	Controls cursor behavior.
Recovery options	Controls recovery models of database.
Miscellaneous options	Controls ANSI compliance.
State options	Controls state of database, such as online/offline and user connectivity.

Table 5.1: Databases Options in SQL Server 2019

Note - Server-wide settings are set using the `sp_configure` system stored procedure or SQL Management Studio.

Code Snippet 4 when executed sets `AUTO_SHRINK` option for the `CUST_DB` database to `ON`. The `AUTO_SHRINK` options when set to `ON`, shrinks the database that have free space.

Code Snippet 4:

```
USE CUST_DB;
ALTER DATABASE CUST_DB SET AUTO_SHRINK ON
```

5.3.5 Filegroups

In SQL Server, data files are used to store database files. The data files are further subdivided into filegroups for the sake of performance. Each filegroup is used to group related files that together store a database object. Every database has a primary filegroup by default. This filegroup contains the primary data file. The primary file group and data files are created automatically with default property values at the time of creation of the database. User-defined filegroups can then be created to group data files together for administrative, data allocation, and placement purposes.

For example, three files named `Customer_Data1.ndf`, `Customer_Data2.ndf`, and `Customer_Data3.ndf`, can be created on three disk drives respectively. These can then be assigned to the filegroup `Customer_fgroup1`. A table can then be created specifically on the filegroup `Customer_fgroup1`. Queries for data from the table will be spread across the three disk drives thereby, improving performance.

Table 5.2 shows filegroups that are supported by SQL Server 2019.

Filegroup	Description
Primary	The filegroup that consists of the primary file. All system tables are placed inside the primary filegroup.
User-defined	Any filegroup that is created by the user at the time of creating or modifying databases.

Table 5.2: Filegroups in SQL Server 2019

Adding Filegroups to an existing database

Filegroups can be created when the database is created for the first time or can be created later when more files are added to the database. However, files cannot be moved to a different filegroup after the files have been added to the database.

A file cannot be a member of more than one filegroup at the same time. A maximum of 32,767 filegroups can be created for each database. Filegroups can contain only data files. Transaction log files cannot belong to a filegroup.

Following is the syntax to add filegroups while creating a database:

Syntax:

```
CREATE DATABASE database_name  
[ ON  
[ PRIMARY ] [ <filespec> [ ,...n ]  
[ , <filegroup> [ ,...n ] ]  
[ LOGON { <filespec> [ ,...n ] } ]  
]  
[ COLLATE collation_name ]  
]  
[ ; ]
```

where,

`database_name`: is the name of the new database.

`ON`: indicates the disk files to store the data sections of the database, and data files.

`PRIMARY` and associated `<filespec>` list: define the primary file. The first file specified in the `<filespec>` entry in the primary filegroup becomes the primary file.

`LOG ON`: indicates the disk files used to store the database log files. `COLLATE collation_name`: is the default collation for the database.

Code Snippet 5 shows how to add a filegroup (`PRIMARY` as default) while creating a database, called **SalesDB**.

Code Snippet 5:

```
CREATE DATABASE [SalesDB] ON PRIMARY  
( NAME = 'SalesDB', FILENAME = 'C:\Program Files\Microsoft SQL  
Server\MSSQL15.MYEXPRESS\MSSQL\DATA\SalesDB.mdf' , SIZE = 3072KB , MAXSIZE =  
UNLIMITED, FILEGROWTH = 1024KB ),  
FILEGROUP [MyFileGroup]  
( NAME = 'SalesDB_FG', FILENAME ='C:\Program Files\Microsoft SQL  
Server\MSSQL15.MYEXPRESS\MSSQL\DATA\SalesDB_FG.ndf' , SIZE = 3072KB , MAXSIZE  
= UNLIMITED, FILEGROWTH = 1024KB )  
LOG ON  
( NAME = 'SalesDB_log', FILENAME ='C:\Program Files\Microsoft SQL  
Server\MSSQL15.MYEXPRESS\MSSQL\DATA\SalesDB_log.ldf' , SIZE = 2048KB , MAXSIZE  
= 2048GB , FILEGROWTH = 10%)  
COLLATE SQL_Latin1_General_CI_AS
```

Figure 5.4 shows the file groups when creating **SalesDB** database.

Name	Date modified
model_replicatedmaster.mdf	24-09-2019 15:09
modellog.ldf	07-10-2020 22:52
MS_AgentSigningCertificate.cer	28-09-2020 13:17
MSDBData.mdf	06-10-2020 22:29
MSDBLog.ldf	06-10-2020 22:31
SalesDB.mdf	07-10-2020 22:52
SalesDB_FG.ndf	07-10-2020 22:52
SalesDB_log.ldf	07-10-2020 22:52
tempdb.mdf	08-10-2020 13:44
templog.ldf	07-10-2020 22:52

Figure 5.4: Filegroup Added When Creating SalesDB Database

Following is the syntax to add a filegroup to an existing database.

Syntax:

```
ALTER DATABASE database_name
{ <add_or_modify_files>
| <add_or_modify_filegroups>
| <set_database_options>
| MODIFY NAME = new_database_name
| COLLATE collation_name
}
[ ; ]
```

Code Snippet 6 shows how to add a filegroup to an existing database, called **CUST_DB**.

Code Snippet 6:

```
USE CUST_DB
ALTER DATABASE CUST_DB
ADD FILEGROUP FG_ReadOnly
```

After executing the code, SQL Server 2019 displays the message 'Command(s) completed successfully' and the filegroup **FG_ReadOnly** is added to the existing database, **CUST_DB**.

Default Filegroup

Objects are assigned to the default filegroup when they are created in the database. The **PRIMARY** filegroup is the default filegroup. The default filegroup can be changed using the **ALTER DATABASE** statement. System objects and tables remain within the **PRIMARY** filegroup, but do not go into the new default filegroup.

To make the **FG_ReadOnly** filegroup as default, it should contain at least one file inside it.

Code Snippet 7 shows how to create a new file, add it to the `FG_ReadOnly` filegroup, and make the `FG_ReadOnly` filegroup that was created in Code Snippet 6 as the default filegroup.

Code Snippet 7:

```
USE CUST_DB
ALTER DATABASE CUST_DB
ADD FILE (NAME = Cust_DB1, FILENAME = 'C:\Program Files\Microsoft SQL
Server\MSSQL15.MYEXPRESS\MSSQL\DATA\Cust_DB1.ndf')
TO FILEGROUP FG_ReadOnly ALTER DATABASE CUST_DB
MODIFY FILEGROUP FG_ReadOnly DEFAULT
```

After executing the code in Code Snippet 7, SQL Server 2019 displays the message saying the filegroup property 'DEFAULT' has been set.

Figure 5.5 shows a new file `Cust_DB1` created.

Microsoft SQL Server > MSSQL15.MYEXPRESS > MSSQL > DATA	
Name	Date modified
AdventureWorks2019.mdf	09-10-2020 01:38
AdventureWorks2019_log.ldf	09-10-2020 01:38
Cust_DB1.ndf	09-10-2020 01:42
Customer_DB.mdf	09-10-2020 01:39
Customer_DB_log.ldf	09-10-2020 01:39
master.mdf	09-10-2020 01:39
mastlog.ldf	09-10-2020 01:39

Figure 5.5: New File `Cust_DB1` Created

5.3.6 Transaction Log

A transaction log in SQL Server records all transactions and the database modifications made by each transaction. The transaction log is one of the critical components of the database. It can be the only source of recent data in case of system failure.

The transaction logs support operations such as the following:

Recovery of individual transactions

An incomplete transaction is rolled back in case of an application issuing a `ROLLBACK` statement or the Database Engine detecting an error. The log records are used to roll back the modifications.

Recovery of all incomplete transactions when SQL Server is started

If a server that is running SQL Server fails, the databases may be left in an inconsistent state. When an instance of SQL Server is started, it runs a recovery of each database.

Rolling a restored database, file, filegroup, or page forward to the point of failure

The database can be restored to the point of failure after a hardware loss or disk failure affecting the database files.

Supporting transactional replication

The Log Reader Agent monitors the transaction log of each database configured for replications of transactions.

Supporting standby server solutions

The standby-server solutions, database mirroring, and log shipping depend on the transaction log.

Working of Transaction Logs:

A database in SQL Server has at least one data file and one transaction log file. Data and transaction log information are kept separated, preferably on separate drives. Individual files are used by only one database.

SQL Server uses the transaction log of each database to recover transactions. The transaction log is a serial record of all modifications that have occurred in the database as well as the transactions that performed the modifications. This log keeps enough information to undo the modifications made during each transaction. The transaction log records the allocation and deallocation of pages and the commit or rollback of each transaction. This feature enables SQL Server either to roll forward or to back out.

The rollback of each transaction is executed using the following ways:

- A transaction is rolled forward when a transaction log is applied.
- A transaction is rolled back when an incomplete transaction is backed out.

➤ Adding Log files to a database

Following is the syntax to modify a database and add log files:

Syntax:

```
ALTER DATABASE database_name
{
...
}
[ ; ]
<add_or_modify_files>::=
{
ADD FILE<filespec> [ ,...n ]
[ TO FILEGROUP { filegroup_name | DEFAULT } ]
| ADD LOG FILE<filespec> [ ,...n ]
| REMOVE FILElogical_file_name
| MODIFY FILE<filespec>
}
```

Note - By default, the data and transaction logs are put on the same drive and path to accommodate single-disk systems, but may not be optimal for production environments.

5.3.7 Creating Databases Using SSMS

When a database is created, data files should be as large as possible based on maximum amount of data, which is expected in the database.

Steps to create a database using SSMS are as follows:

1. In **Object Explorer**, connect to an instance of the SQL Server Database Engine and then, expand that instance.
2. Right-click **Databases**, and then, click **New Database** as shown in Figure 5.6.

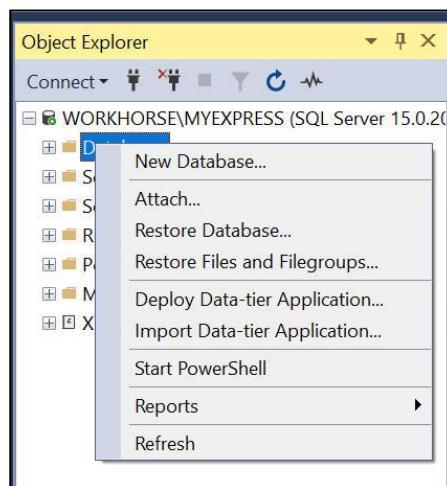


Figure 5.6: New Database Option

3. In **New Database** dialog box, enter a database name in the **Database name** box.
4. To create the database by accepting all default values, click **OK**, as shown in Figure 5.7; otherwise, continue with the following optional steps.

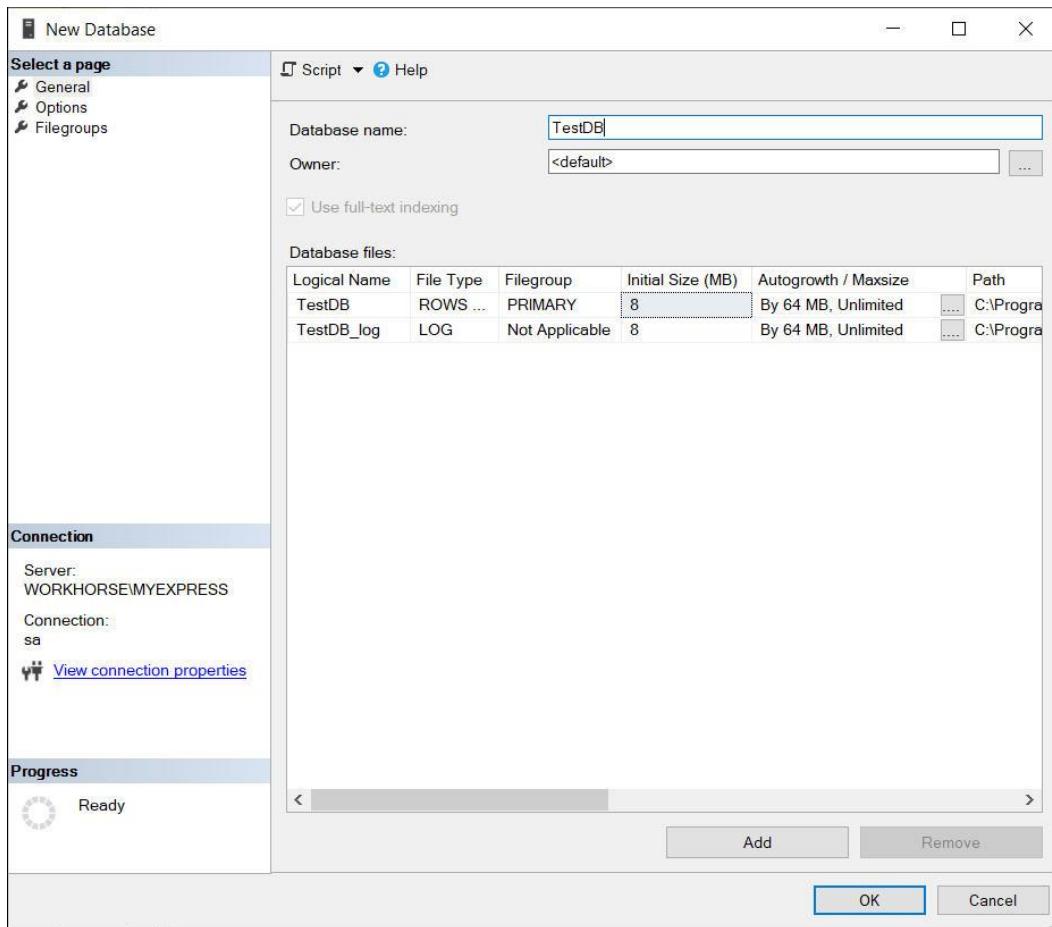


Figure 5.7: New Database Window

5. To change the owner name, click (...) to select another owner.
6. To change the default values of the primary data and transaction log files, in the **Database files** grid, click the appropriate cell and enter the new value.
7. To change the collation of the database, select the **Options** page, and then, select a collation from the list as shown in Figure 5.8.

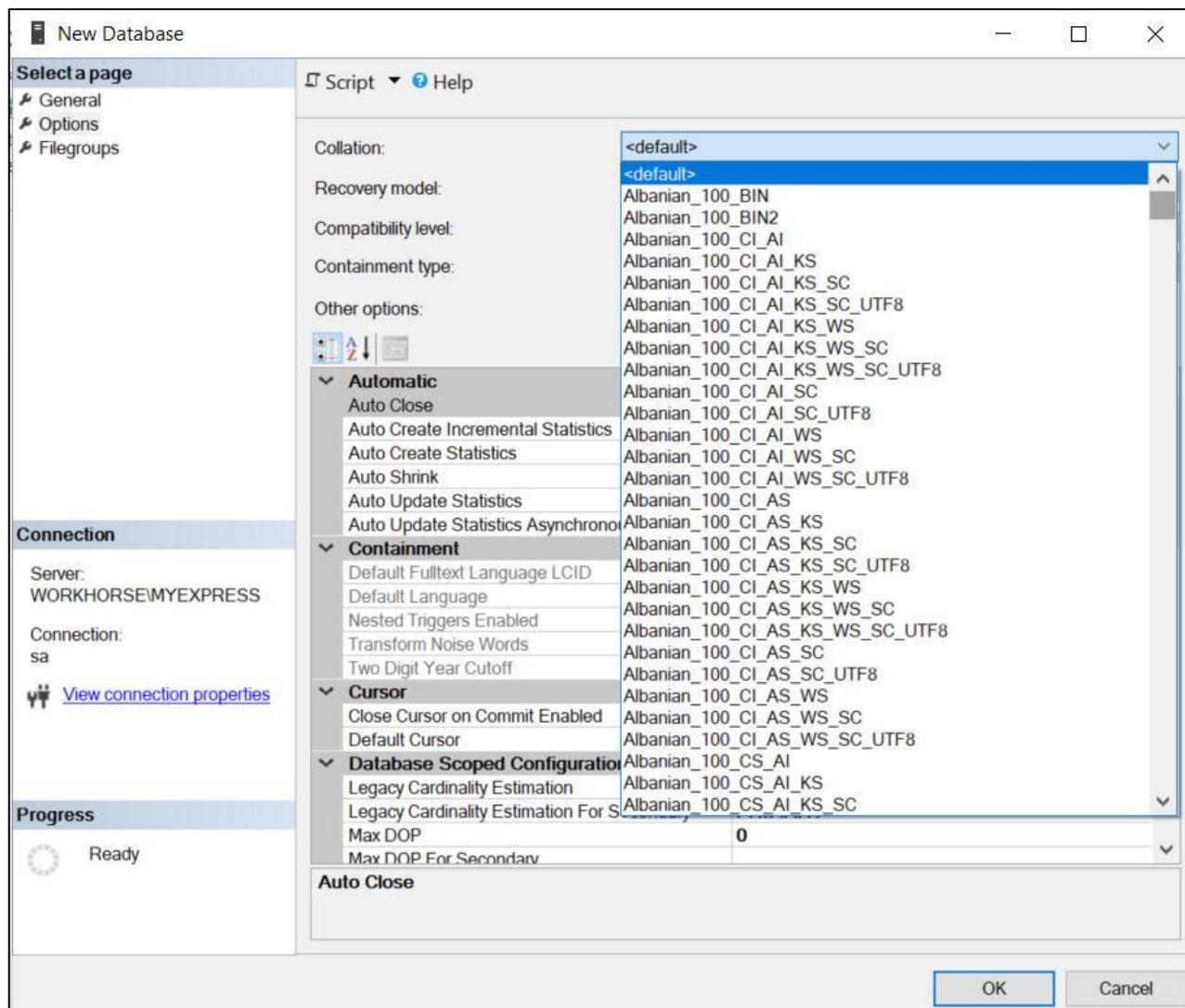


Figure 5.8: Collation List

- To change the recovery model, select the **Options** page and then, select a recovery model from the list as shown in Figure 5.9.

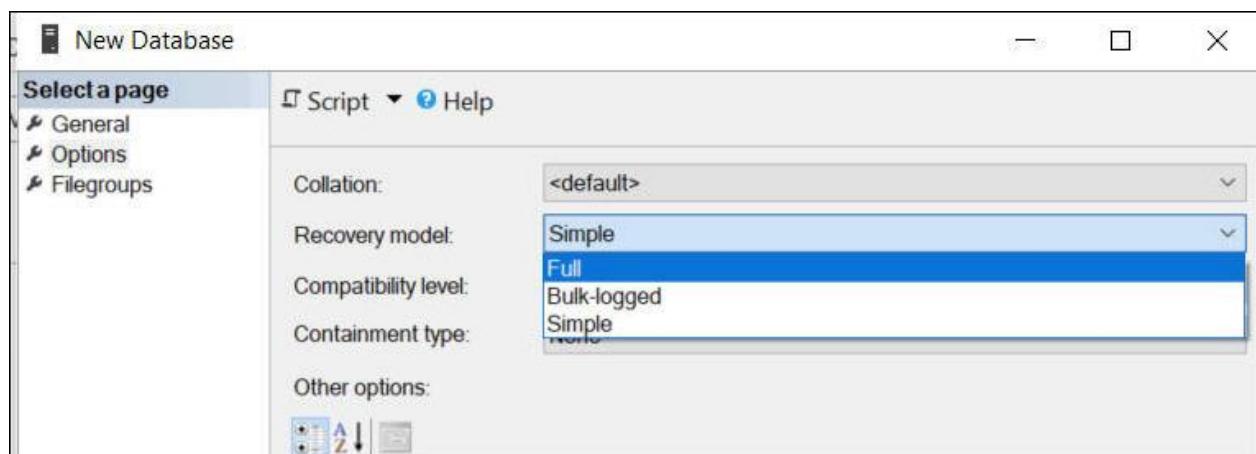


Figure 5.9: Recovery Model

- To change other database options, select the **Options** page, and then, modify the other database options.
- To add a new filegroup, click the **Filegroups** page. Click **Add** and then, enter the values for the filegroup as shown in Figure 5.10.

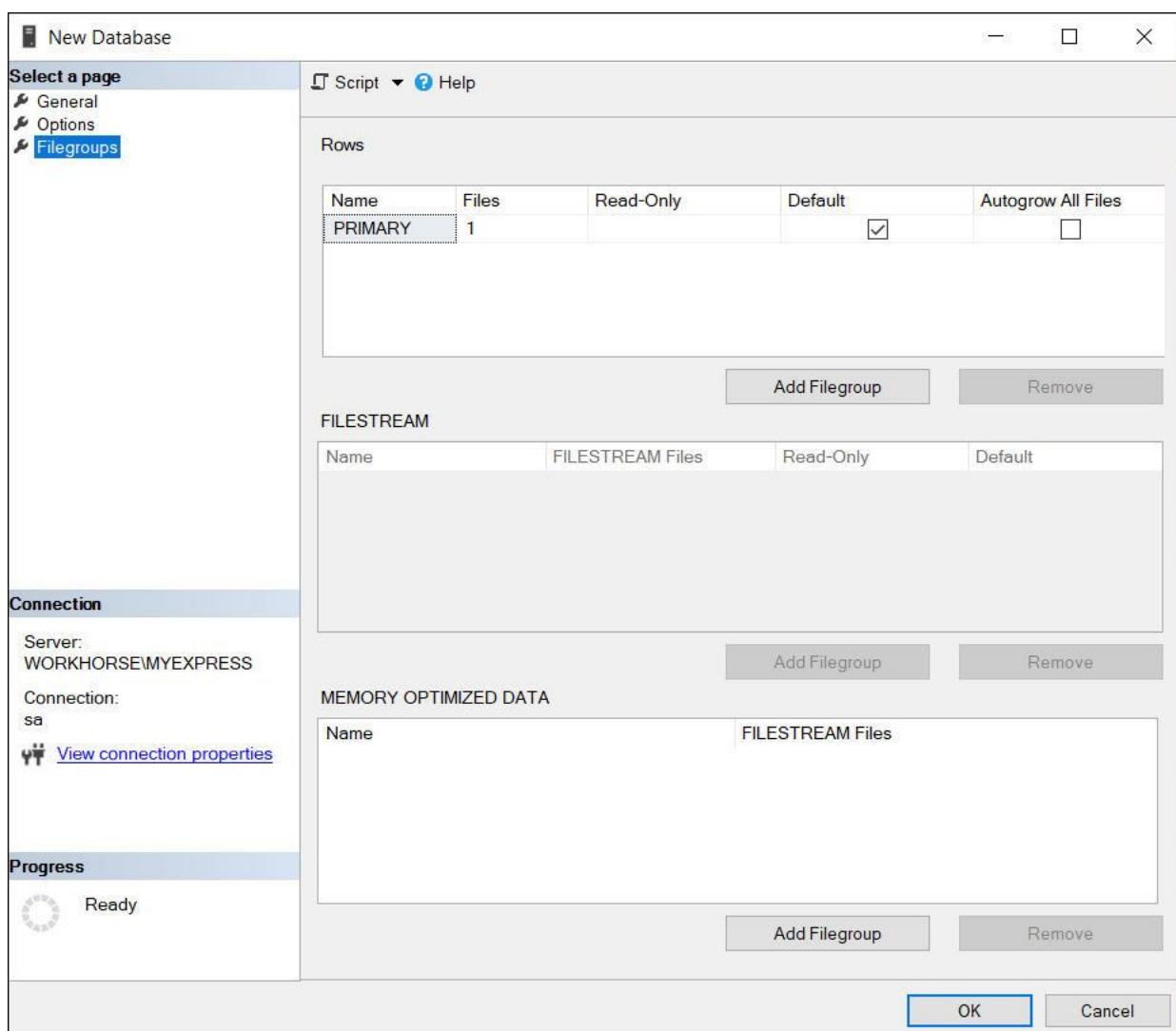


Figure 5.10: Adding a Filegroup

- To add an extended property to the database, select the **Extended Properties** page.
 - In the **Name** column, enter a name for the extended property.
 - In the **Value** column, enter the extended property text. For example, enter one or more statements that describe the database.
- To create the database, click **OK**.

The database will be successfully created and will be visible in Object Explorer after the Databases node is refreshed.

5.3.8 Dropping a Database

A full backup of the database must be taken before dropping a database. A deleted database can be re-created only by restoring a backup.

The steps to delete or drop a database using SSMS are as follows:

1. In **Object Explorer**, connect to an instance of the SQL Server Database Engine, and then, expand that instance.
2. Expand **Databases**, right-click the database to delete, and then, click **Delete**, as shown in figure 5.11.

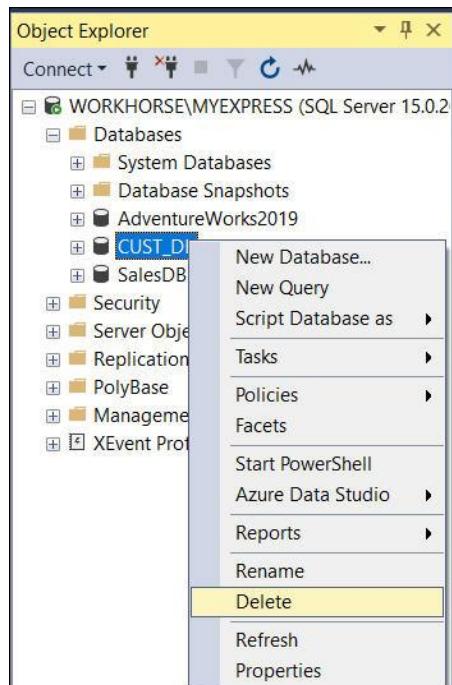


Figure 5.11: Delete a Database

3. Confirm that the correct database is selected, and then, click **OK**.

Apart from using SSMS to drop a database, one can also use a Transact-SQL command.

Following is the syntax to delete or drop a database using Transact-SQL:

Syntax:

```
DROP DATABASE [databasename]
```

5.3.9 Creating Database Snapshots

Database snapshot feature was first introduced in Microsoft SQL Server 2005 and is present in SQL Server 2019 as well. It is a feature that provides a read-only, static view of a SQL Server database. If a user makes a mistake in a source database, the source database can be reverted to the previous state when the snapshot was created. You must use Transact-SQL to create a database snapshot. SSMS does not support creation of database snapshots.

Some of the advantages and disadvantages of database snapshots are as follows:

Advantages	Disadvantages
<ul style="list-style-type: none">Provide a convenient, read-only copy of data.When queried, no deterioration of performance.Snapshot files are small and are very quick to create.	<ul style="list-style-type: none">Snapshot backup cannot be created.Snapshot must exist on the same database server as that of the source database.A new user cannot be granted access to the data in a snapshot.

Following is the syntax to create a database snapshot using Transact-SQL:

Syntax:

```
CREATE DATABASE database_snapshot_name  
ON  
(  
NAME = logical_file_name, \ FILENAME = 'os_file_name'  
) [ ,...n ]  
AS SNAPSHOT OF source_database_name  
[ ; ]
```

where,

database_snapshot_name: is the name of the new database snapshot.

ON (NAME = logical_file_name, FILENAME = 'os_file_name') [,... n]: is the list of files in the source database. For the snapshot to work, all the data files must be specified individually.

AS SNAPSHOT OF source_database_name: is a database snapshot of the source database specified by source_database_name.

Code Snippet 9 creates a database snapshot on the AdventureWorks2019 database.

Code Snippet 9:

```
CREATE DATABASE AdventureWorks_dbss100 ON  
( NAME = AdventureWorks2019, FILENAME =  
'C:\Program Files\Microsoft SQL  
Server\MSSQL15.MYEXPRESS\MYSQL\DATA\AdventureWorks_data_100.ss' )  
AS SNAPSHOT OF AdventureWorks2019;  
GO
```

5.4 Check Your Progress

1. Which of the following is a complete administration utility?

(A) SQL-SMO	(C) Transact-SQL
(B) SQL Server Management Studio	(D) Stored procedures

2. Which of these are required information in order to create a user-defined database?

- a) Name of the database
- b) Owner or creator of the database
- c) Size of the database
- d) Files and filegroups used to store it

(A) a, b, c	(C) b, d
(B) c, d	(D) All of these

3. Which of these terms correspond to the description, 'Defines rules for comparing and sorting character data based on the standard of particular language and locale.'

(A) Collations	(C) Indexes
(B) Triggers	(D) Tables

4. Which among these is the correct syntax to create a database?

(A)	CREATE DATABASE (NAME = 'Customer_DB', FILENAME = 'C:\Program Files\Microsoft SQL Server\MSSQL15.MYEXPRESS\MSSQ L\DATA\Customer_DB.mdf')	(C)	CREATE DATABASE [Customer_ DB] ON PRIMARY (NAME = 'Customer_DB', FILENAME = 'C:\Program Files\Microsoft SQL Server\MSSQL15.MYEXPRESS\MSSQL\DAT A\Customer_DB.mdf')
(B)	CREATE DB [Customer_DB] ON PRIMARY (NAME = 'Customer_DB', FILENAME = 'C:\Program Files\Microsoft SQL Server\MSSQL15.MYEXPRESS\MSSQ L\DATA\Customer_DB.mdf')	(D)	All of these

5. Which of these are valid advantages of database snapshots?

- a) Provide a convenient and read-write copy of data.
- b) When queried, no deterioration of performance.
- c) Snapshot files are small and are very quick to create.

(A)	a, b, c	(C)	b, c
(B)	c, d	(D)	All of these

5.4.1 Answers

1.	A
2.	D
3.	A
4.	C
5.	C



Summary

- SQL Server uses system databases to support different components of the DBMS.
- The SQL Server data files are used to store database files, which are further subdivided into filegroups for the sake of performance.
- Databases can be created and dropped using SSMS or Transact-SQL.
- The CREATE DATABASE statement with various options can be used to create databases.
- ALTER DATABASE and DROP DATABASE are used to modify and delete a database respectively.
- A transaction log in SQL Server records all transactions and the database modifications made by each transaction.
- Objects are assigned to the default filegroup when they are created in the database. The PRIMARY filegroup is the default filegroup.
- A database snapshot is a read-only, static view of a SQL Server database.



Try It Yourself

1. Create a database named **UnitedAir** using Transact-SQL statements with the following properties:
 - Primary filegroup with files, **UnitedAir1_dat** and **UnitedAir2_dat**. The size, maximum size, and file growth should be 5, 10, and 15% respectively. Note that **_dat** is given as a suffix to the file name to indicate that it is a filegroup name.
 - A filegroup named **UnitedAirGroup1** with the files **UnitedAirGrp1F1** and **UnitedAirGrp1F2**.
 - A filegroup with **UnitedAirGroup2** with the files **UnitedAirGrp2F1** and **UnitedAirGrp2F2**.
2. Repeat similar process for a database named **BelAir**, however, use SSMS to do this instead of Transact SQL.
3. Create database snapshots for **UnitedAir** and **BelAir** databases.

Session - 6

Creating Tables

Welcome to the Session, **Creating Tables**.

This session explores various data types provided by SQL Server 2019 and describes how to use them. The techniques for creation, modification, and removal of tables and columns are also discussed.

In this session, you will learn to:

- List SQL Server 2019 data types
- Describe the procedure to create, modify, and drop tables in an SQL Server database
- Describe the procedure to add, modify, and drop columns in a table

6.1 Introduction

One of the most important types of database objects in SQL Server 2019 is a table. Tables in SQL Server 2019 contain data in the form of rows and columns. Each column may have data of a specific type and size.

6.2 Advanced Data Types

Besides basic data types such as `int`, `float`, and so on, SQL Server also supports several advanced data types.

Name	Description
<code>hierarchyid</code>	It is a system data type with variable length. You can use it to represent a position in a hierarchy.
<code>geometry</code>	It is a spatial data type, implemented as a Common Language Runtime (CLR) data type in SQL Server. It represents data in a Euclidean (flat) coordinate system. SQL Server supports a set of methods for this data type. The <code>geometry</code> type is predefined and available in each database. You can create table columns of type <code>geometry</code> and operate on <code>geometry</code> data similar to how you do on other CLR types.
<code>geography</code>	It is a spatial type for storing ellipsoidal (round-earth) data, such as GPS latitude and longitude coordinates. SQL Server supports a set of methods for the <code>geography</code> spatial data type. This type too is predefined and available in each database. You can create table columns of type <code>geography</code> .
<code>xml</code>	It is a special data type for storing XML data in SQL Server tables.
<code>cursor</code>	It is a data type for variables or stored procedure <code>OUTPUT</code> parameters that contain a reference to a cursor.

Name	Description
table	It is a special data type useful for storing result set temporarily in a table-valued function. You can use data from this for processing later. It can be used in functions, stored procedures, and batches
rowversion	It returns automatically generated, unique binary numbers within a database.

Table 6.1: Advanced Data Types in SQL Server

6.3 Creating, Modifying, and Dropping Tables

Most tables have a primary key, made up of one or more columns of the table. A primary key is always unique. The Database Engine will enforce the restriction that any primary key value cannot be repeated in the table. Thus, the primary key can be used to identify each record uniquely.

Note: **AdventureWorks2019** database will be used for all the subsequent statements/commands.

6.3.1 Creating Tables

The `CREATE TABLE` statement is used to create tables in SQL Server 2019. A simple basic syntax is as follows:

Syntax:

```
CREATE TABLE [database_name.] [schema_name]. [table_name]
  ([<column_name>] [data_type] Null/Not Null,
  ON [filegroup | "default"])
```

where,

`database_name`: is the name of the database in which the table is created.

`table_name`: is the name of the new table. `table_name` can be a maximum of 128 characters.

`column_name`: is the name of a column in the table. `column_name` can be up to 128 characters. `column_name` are not specified for columns that are created with a timestamp data type. The default column name of a `timestamp` column is `timestamp`.

`data_type`: It specifies data type of the column.

`ON [filegroup | "default"]`: When you create a database in SQL Server, you can have multiple file groups, where storage is created in multiple places, directories or disks. Each file group can be named. The `PRIMARY` file group is the default one, which is always created. Giving `ON PRIMARY` creates the table on this file group.

Code Snippet 1 demonstrates creation of a table named `dbo.CustomerInfo`.

Code Snippet 1:

```
USE AdventureWorks2019
CREATE TABLE [dbo].[CustomerInfo] (
[CustomerID] numeric(10, 0) NOT NULL, [CustomerName] [varchar](50)
NOT NULL
ON [PRIMARY]
```

```
GO
```

The table will be successfully created, as shown in figure 6.1.

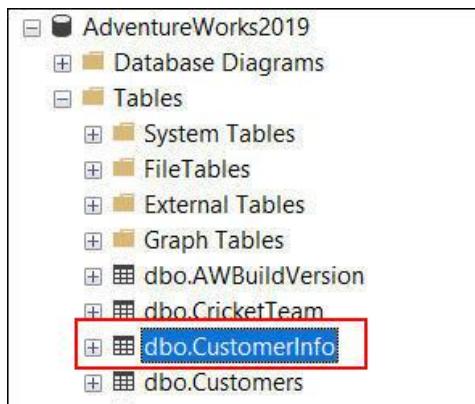


Figure 6.1: Creating a Table

6.3.2 Modifying Tables

The `ALTER TABLE` statement is used to modify a table definition by altering, adding, or dropping columns and constraints, reassigning partitions, or disabling or enabling constraints and triggers.

Syntax:

```
ALTER TABLE [ [database_name].[schema_name]. | schema_name.]table_name
  ALTER COLUMN ([<column_name>] [data_type] Null/Not Null,);
  | ADD ([<column_name>] [data_type] Null/Not Null,);
  | DROP COLUMN ([<column_name>]);
```

where,

`ALTER COLUMN`: specifies that the particular column must be changed or modified.

`ADD`: specifies that one or more column definitions are to be added.

`DROP COLUMN` ([`<column_name>`]): specifies that `column_name` is to be removed from the table.

Code Snippet 2 demonstrates altering the `CustomerID` column.

Code Snippet 2:

```
ALTER TABLE [dbo].[CustomerInfo]
ALTER Column [CustomerID] [numeric] (12, 0) NOT NULL;
```

Code Snippet 3 demonstrates adding the `ContactNumber` column.

Code Snippet 3:

```
ALTER TABLE [dbo].[CustomerInfo]
ADD [ContactNumber] [numeric] (12, 0) NOT NULL;
```

Code Snippet 4 demonstrates dropping the `ContactNumber` column.

Code Snippet 4:

```
ALTER TABLE [dbo].[CustomerInfo]
DROP COLUMN [ContactNumber];
```

Before attempting to drop columns, however, it is important to ensure that the columns can be dropped.

Under certain conditions, columns cannot be dropped, such as, if they are used in a CHECK, FOREIGN KEY, UNIQUE, or PRIMARY KEY constraint, associated with a DEFAULT definition, and so forth. Constraints will be explored in a later section.

6.3.3 Dropping Tables

The `DROP TABLE` statement removes a table definition, its data, and all associated objects such as indexes, triggers, constraints, and permission specifications for that table.

Syntax:

```
DROP TABLE <Table_Name>
```

where,

`<Table_Name>`: is the name of the table to be dropped.

Code Snippet 5 demonstrates how to drop a table.

Code Snippet 5:

```
DROP TABLE [dbo].[CustomerInfo]
```

The table will be removed from the database. Once the table is removed, all data in it will be gone too.

6.3.4 Data Modification Statements

The statements used for modifying data are `INSERT`, `UPDATE`, and `DELETE` statements. These are explained as follows:

- **INSERT Statement** - The `INSERT` statement adds a new row to a table.

Syntax:

```
INSERT [INTO] <Table_Name> VALUES <values>
```

where,

`<Table_Name>`: is the name of the table in which row is to be inserted.

`[INTO]`: is an optional keyword used between `INSERT` and the target table.

`<Values>`: specifies the values for columns of the table.

Code Snippet 6 demonstrates adding a new row to the **Person.PersonPhone** table of **AdventureWorks2019** database.

Code Snippet 6:

```
INSERT INTO [Person].[PersonPhone]
([BusinessEntityID], [PhoneNumber], [PhoneNumberTypeID], [ModifiedDate])
VALUES (299, 699-511-0142, 1, '2020-10-12' )
```

The outcome of this will be that one row with the given data is inserted into the table.

- **UPDATE Statement** - The UPDATE statement modifies the data in the table.

Syntax:

```
UPDATE <Table_Name>
SET <Column_Name = Value>
[WHERE <Search condition>]
```

where,

- <Table_Name>: is the name of the table where records are to be updated.
- <Column_Name>: is the name of the column in the table in which record is to be updated.
- <Value>: specifies the new value for the modified column.
- <Search condition>: specifies the condition to be met for the rows to be deleted.

Code Snippet 7 demonstrates the use of the UPDATE statement to modify the value in column **PhoneNumber**.

Code Snippet 7:

```
UPDATE [Person].[PersonPhone]
SET [PhoneNumber] = '731-511-0142' WHERE BusinessEntityID=299 AND
ModifiedDate='2020-10-12'
```

The specified value will be updated in the **PhoneNumber** column of **Person.PersonPhone** table.

- **DELETE Statement** - The DELETE statement removes rows from a table.

Syntax:

```
DELETE FROM <Table_Name> [WHERE <Search condition>]
```

where,

- <Table_Name>: is the name of the table from which the records are to be deleted.
- The WHERE clause is used to specify the condition. If WHERE clause is not included in the DELETE statement, all the records in the table will be deleted.

Code Snippet 8 demonstrates how to delete a row from the **Person.PersonPhone** table whose **PhoneNumber** value is 731-511-0142 and **BusinessEntityID** is 229.

Code Snippet 8:

```
DELETE FROM [Person].[PersonPhone]  
WHERE BusinessEntityID=299 AND [PhoneNumber] = '731-511-0142'
```

The specified row will be deleted from the table.

Note: If WHERE clause is not specified with the DELETE statement for a table, all the rows of the table will be deleted.

6.3.5 Column Nullability

The nullability feature of a column determines whether rows in the table can contain a null value for that column. In SQL Server, a null value is not same as zero, blank, or a zero length character string (such as ' '). For example, a null value in Color column of Production.Product table of AdventureWorks2019 database does not mean that the product has no color; it just means that color for the product is unknown or has not been set.

Nullability of a column can be defined either when creating a table or modifying a table. The NULL keyword is used to indicate that null values are allowed in the column and NOT NULL is used to indicate that null values are not allowed.

When inserting a row, if no value is given for a nullable column (that is, it allows null values), then, SQL Server automatically gives it a null value unless the column has been given a default definition. It is also possible to explicitly enter a null value into a column regardless of what data type it is or whether it has a default associated with it. Making a column non-nullable (that is, not permitting null values) enforces data integrity by ensuring that the column contains data in every row.

In Code Snippet 9, the CREATE TABLE statement uses the NULL and NOT NULL keywords with column definitions.

Code Snippet 9:

```
CREATE TABLE StoreDetails (StoreID int NOT NULL, Name varchar(40) NULL)
```

The result of the command is that **StoreDetails** table is created with **StoreID** and **Name** columns, and **StoreID** will not accept any null values whereas **Name** will allow null values.

Thus, the code in Code Snippet 10 will not succeed, because it assumes a null value will be inserted into **StoreID** column.

Code Snippet 10:

```
INSERT INTO [dbo].[StoreDetails] (Name) VALUES ('BrooklynStore')
```

Figure 6.2 depicts the resulting error when this command is executed.

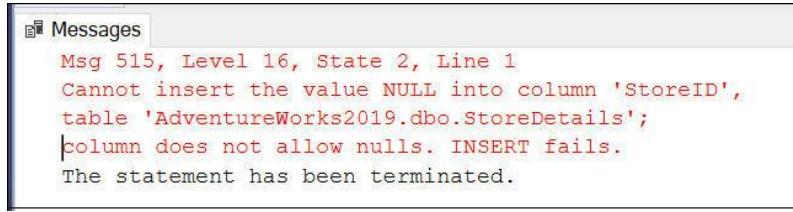


Figure 6.2: Error When Trying to Insert Row with NOT NULL Column

6.3.6 DEFAULT Definition

Consider a scenario in which details about a product must be stored in an SQL Server 2019 table, but all values for the product details may not be known even at the time of data insertion. However, as per data consistency and integrity rules, the columns in a record should typically contain a value. Storing null values into such columns where exact value of data is not known may not be desirable or practical.

In such situations, a `DEFAULT` definition can be given for the column to assign it as a default value if no value is given at the time of creation. For example, it is common to specify zero as the default for numeric columns or 'N/A' or 'Unknown' as the default for string columns when no value is specified.

A `DEFAULT` definition for a column can be created at the time of table creation or added at a later stage to an existing table. When a `DEFAULT` definition is added to an existing column in a table, SQL Server applies the new default values only to those rows of data, which have been newly added to the table.

In Code Snippet 11, `CREATE TABLE` statement uses `DEFAULT` keyword to define default value for `Price`.

Code Snippet 11:

```
CREATE TABLE StoreProduct (ProductID int NOT NULL, Name varchar (40) NOT NULL, Price money NOT NULL DEFAULT (100))
```

When a row is inserted using a statement as shown in Code Snippet 12, value of `Price` will not be blank; it will have a value of 100.00 even though a user has not entered any value for that column.

Code Snippet 12:

```
INSERT INTO dbo.StoreProduct (ProductID, Name) VALUES (111, 'Rivets')
SELECT * FROM dbo.StoreProduct
GO
```

Figure 6.3 shows the output of Code Snippet 12, where, though values are added only to `ProductID` and `Name` columns, `Price` column will still show a value of 100.00. This is because of `DEFAULT` definition.

Results			Messages		
	ProductID	Name	Price		
1	111	Rivets	100.00		

Figure 6.3: Demonstrating Use of DEFAULT

Following cannot be created on columns with `DEFAULT` definitions:

A timestamp data type

An `IDENTITY` or `ROWGUIDCOL` property

An existing default definition or default object

6.3.7 *IDENTITY Property*

The `IDENTITY` property of SQL Server is used to create identifier columns that can contain auto-generated sequential values to uniquely identify each row within a table. For example, an identifier column could be created to generate unique student registration numbers automatically whenever new rows are inserted into the `Students` table. The identity number for the first row inserted into the table is called seed value. The increment, also called Identity Increment property, is added to the seed in order to generate further identity numbers in sequence. When a new row is inserted into a table with an identifier column, the next identity value is automatically generated by SQL Server by adding the increment to the seed. An identity column is often used for primary key values.

Characteristics of the `IDENTITY` property are as follows:

A column having `IDENTITY` property must be defined using one of the following data types:

`decimal`, `int`, `numeric`, `smallint`, `bigint`, or `tinyint`

A column having `IDENTITY` property need not have a seed and increment value specified. If they are not specified, a default value of 1 will be used for both.

A table cannot have more than one column with `IDENTITY` property.

The identifier column in a table must not allow null values and must not contain a `DEFAULT` definition or object.

Columns defined with `IDENTITY` property cannot have their values updated.

The values can be explicitly inserted into the identity column of a table only if the `IDENTITY_INSERT` option is set ON. When `IDENTITY_INSERT` is ON, `INSERT` statements must supply a value.

The advantage of identifier columns is that SQL Server can automatically provide key values, thus reducing costs that would have been incurred for extra storage and improving performance. Using identifier columns simplifies programming and keeps primary key values short.

Once the `IDENTITY` property has been set, retrieving the value of the identifier column can be done by using the `IDENTITYCOL` keyword with the table name in a `SELECT` statement. To know if a table has an `IDENTITY` column, the `OBJECTPROPERTY()` function can be used. To retrieve the name of the `IDENTITY` column in a table, the `COLUMNPROPERTY` function is used.

Syntax:

```
CREATE TABLE <table_name> (column_name data_type [IDENTITY  
[(seed_value, increment_value)]] NOT NULL)
```

where,

`seed_value`: is the seed value which starts generating identity values.

`increment_value`: is the increment value by which to increase each time.

Code Snippet 13 demonstrates the use of `IDENTITY` property. `HRContactPhone` is created as a table with two columns. The `Person_ID` column is an identity column. The seed value is 500 and the increment value is 1.

Code Snippet 13:

```
CREATE TABLE HRContactPhone (Person_ID int IDENTITY(500,1) NOT NULL,  
MobileNumber bigint NOT NULL)
```

While inserting rows into the table, if `IDENTITY_INSERT` is not turned ON, then, explicit values for the `IDENTITY` column cannot be given. Instead, statements similar to Code Snippet 14 can be given.

Code Snippet 14:

```
INSERT INTO HRContactPhone (MobileNumber) VALUES (983452201)  
INSERT INTO HRContactPhone (MobileNumber) VALUES (993026654)  
SELECT * FROM HRContactPhone  
GO
```

Figure 6.4 shows the output where `IDENTITY` property is incrementing `Person_ID` column values.

Results		
	Person_ID	MobileNumber
1	500	983452201
2	501	993026654

Figure 6.4: IDENTITY Property Applied on Person_ID Column

6.3.8 Globally Unique Identifiers

In addition to the `IDENTITY` property, SQL Server also supports globally unique identifiers. Often, in a networked environment, many tables may require to have a column consisting of a common globally unique value. Consider a scenario where data from multiple database systems such as banking databases must be consolidated at a single location. When the data from around the world is collated at the central site for consolidation and reporting, using globally unique values prevents customers in different countries from having the same bank account number or customer ID. To satisfy this

requirement, SQL Server provides globally unique identifier columns. These can be created for each table containing values that are unique across all the computers in a network. Only one identifier column and one globally unique identifier column can be created for each table. To create and work with globally unique identifiers, a combination of `ROWGUIDCOL`, `uniqueidentifier` data type, and `NEWID` functions are used.

Values for a globally unique column are not automatically generated. One has to create a `DEFAULT` definition with a `NEWID()` function for a `uniqueidentifier` column to generate a globally unique value. The `NEWID()` function creates a unique identifier number which is a 16-byte binary string. The column can be referenced in a `SELECT` list by using the `ROWGUIDCOL` keyword.

To know whether a table has a `ROWGUIDCOL` column, the `OBJECTPROPERTY` function is used. The `COLUMNPROPERTY` function is used to retrieve the name of the `ROWGUIDCOL` column. Code Snippet 15 demonstrates how to use `CREATE TABLE` statement to create the `EMPCellularPhone` table.

The `Person_ID` column automatically generates a `GUID` for each new row added to the table.

Code Snippet 15:

```
CREATE TABLE EMP_CellularPhone (Person_ID uniqueidentifier DEFAULT NEWID() NOT NULL, PersonName varchar(60) NOT NULL)
```

Code Snippet 16 adds a value to `PersonName` column.

Code Snippet 16:

```
INSERT INTO EMP_CellularPhone (PersonName) VALUES ('William Smith')
SELECT * FROM EMP_CellularPhone
GO
```

Figure 6.4 shows the output where a unique identifier is displayed against a specific `PersonName`.

Results	
Person_ID	PersonName
1	362C4377-D194-4607-A466-7FF02064EAFC

Figure 6.4: Unique Identifier

6.4 Constraints

One of the important functions of SQL Server is to maintain and enforce data integrity. There are a number of means to achieve this, but one of the commonly used and preferred methods is to use constraints. A constraint is a property assigned to a column or set of columns in a table to prevent certain types of inconsistent data values from being entered. Constraints are used to apply business logic rules and enforce data integrity.

Constraints can be created when a table is created, as part of the table definition by using `CREATE TABLE` statement or can be added at a later stage using `ALTER TABLE` statement.

Constraints can be categorized as column constraints and table constraints.

Column Constraint	Table Constraint
Is specified as part of a column definition and applies only to that column.	Can apply to more than one column in a table and is declared independently from a column definition. Table constraints must be used when more than one column is included in a constraint.

SQL Server supports the following types of constraints:



6.4.1 PRIMARY KEY

A table typically has a primary key comprising a single column or combination of columns to uniquely identify each row within the table. The `PRIMARY KEY` constraint is used to create a primary key and enforce integrity of the entity in the table.

Note: Only one primary key constraint can be created per table.

Two rows in a table cannot have the same primary key value and a column that is a primary key cannot have `NULL` values. Hence, when a primary key constraint is added to existing columns of a table, SQL Server 2019 checks to see if the rules for primary key are complied with. If the existing data in the columns do not comply with the rules for primary key then, the constraint will not be added.

Syntax to add a primary key while creating a table is as follows:

Syntax:

```
CREATE TABLE <table_name> (Column_Name datatype PRIMARY KEY [column_list] )
```

Code Snippet 17 demonstrates how to create a table `EMPCONTACTPHONE` to store contact telephone details of a person. Since the column `EMP_ID` must be a primary key for identifying each row uniquely, it is created with the primary key constraint.

Code Snippet 17:

```
CREATE TABLE EmpContactPhone (Employee_ID int PRIMARY KEY, MobileNumber bigint,  
ServiceProvider varchar(30), LandlineNumber bigint)
```

An alternative approach is to use CONSTRAINT keyword. The syntax is as follows:

Syntax:

```
CREATE TABLE <table_name> (<column_name><datatype> [, column_list] CONSTRAINT  
constraint_name PRIMARY KEY)
```

Having created a primary key for `Employee_ID`, a query is written to insert rows into the table with the statements shown in Code Snippet 18.

Code Snippet 18:

```
INSERT INTO dbo.EmpContactPhone VALUES (101, 983345674, 'Verizon', NULL)  
INSERT INTO dbo.EmpContactPhone VALUES (101, 989010002, 'AT & T', NULL)  
GO
```

The first INSERT statement shown in Code Snippet 18 is executed successfully, but the next INSERT statement will fail because the value for `Employee_ID` is duplicate as shown in Figure 6.5.

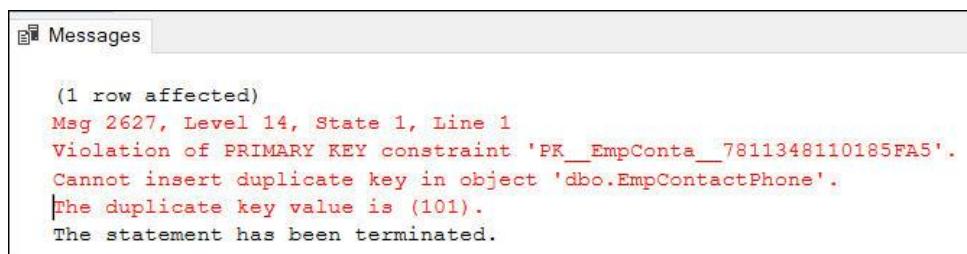


Figure 6.5: Output Error Message for Duplicate EMP_ID

Output of Code Snippet 18 is shown in Figure 6.6.

Results				
	EMP_ID	MobileNumber	ServiceProvider	LandlineNumber
1	101	983345674	Verizon	NULL

Figure 6.6: Output of the Successfully Executed First INSERT Statement

6.4.2 UNIQUE

A UNIQUE constraint is used to ensure that only unique values are entered in a column or set of columns. It allows developers to make sure that no duplicate values are entered. Primary keys are implicitly unique. Unique key constraints enforce entity integrity because once the constraints are applied; no two rows in

the table can have the same value for the columns. **UNIQUE** constraints allow null values. A single table can have more than one **UNIQUE** constraint. The syntax to create **UNIQUE** constraint is as follows:

Syntax:

```
CREATE TABLE <table_name> ([column_list] <column_name><data_type> UNIQUE  
[column_list])
```

Code Snippet 19 demonstrates how to make **MobileNumber** and **LandlineNumber** columns as unique.

Code Snippet 19:

```
CREATE TABLE NewEmpContactPhone(Person_ID int, MobileNumber bigint PRIMARY  
KEY, ServiceProvider varchar(30), LandlineNumber bigint UNIQUE)
```

Code Snippet 20 demonstrates how to insert rows into this table.

Code Snippet 20:

```
INSERT INTO NewEmpContactPhone values (111, 983345674, 'Verizon', NULL)  
INSERT INTO NewEmpContactPhone values (112, 983345674, 'AT & T', NULL)  
GO
```

Though a value of **NULL** has been given for the **LandlineNumber** columns, which are defined as **UNIQUE**, the command will execute successfully because **UNIQUE** constraints check only for the uniqueness of values, but do not prevent null entries. The first statement shown in Code Snippet 20 is executed successfully, but the next **INSERT** statement will fail even though the primary key value is different because the value for **MobileNumber** is a duplicate value as shown in Figure 6.7. This is because the column **MobileNumber** is defined to be unique and disallows duplicate values.

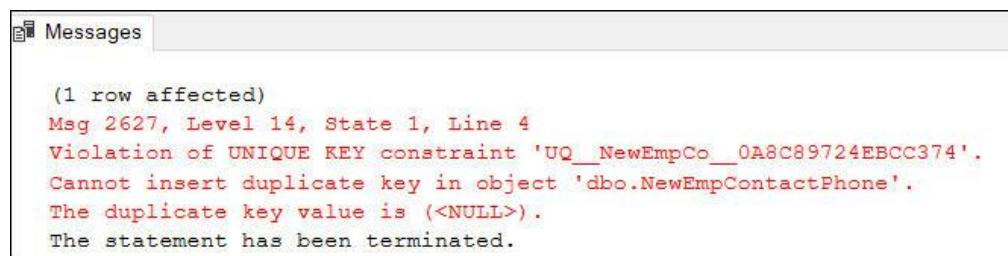


Figure 6.7: Output Error Message for Value Duplicate MobileNumber

Figure 6.8 shows the result of retrieving rows from **NewEmpContactPhone** table. It shows the row that was successfully inserted as an outcome of the first **INSERT** statement.

	Person_ID	MobileNumber	ServiceProvider	LandlineNumber
1	111	983345674	Verizon	NULL

Figure 6.8: Successfully Inserted Row

6.4.3 FOREIGN KEY

A foreign key in a table is a column that points to a primary key or unique column in a table. Foreign key constraints are used to enforce referential integrity. The syntax for foreign key is as follows:

Syntax:

```
CREATE TABLE <table_name1>([column_list,] <column_name><datatype> FOREIGN KEY  
REFERENCES <table_name> (pk_column_name) [, column_list])
```

where,

- <table_name>: is the name of the table from which to reference primary key.
- <pk_column_name>: is the name of the primary key column.

Code Snippet 21 demonstrates how to create a foreign key constraint. Here, it ensures that **MobileNumber** in **EmpPhoneExpenses** is linked to **MobileNumber** column in **NewEmpContactPhone**. This means that you cannot insert any values in the **MobileNumber** column of **EmpPhoneExpenses** that do not exist in **MobileNumber** column of **NewEmpContactPhone**.

Code Snippet 21:

```
CREATE TABLE EmpPhoneExpenses (Expense_ID int PRIMARY KEY, MobileNumber bigint  
FOREIGN KEY REFERENCES NewEmpContactPhone (MobileNumber), Amount bigint)
```

Figure 6.9 shows the database diagram depicting the relationship between **NewEmpContactPhone** and **EmpPhoneExpenses**. The two tables are related based on the column **MobileNumber**.

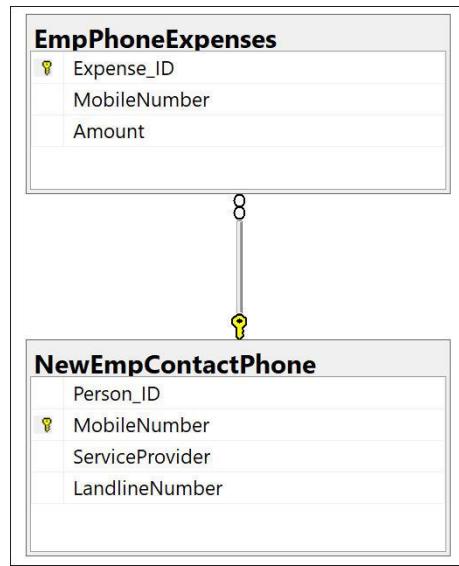


Figure 6.9: Database Diagram Showing Foreign Key and Relationship Between Tables

Now, a row is inserted into **EmpPhoneExpenses** table such that the mobile number is the same as one of the mobile numbers in **NewEmpContactPhone**. The statements that will be written are shown in Code Snippet 22.

Code Snippet 22:

```
INSERT INTO dbo.EmpPhoneExpenses VALUES (101, 993026654, 500)  
SELECT * FROM dbo.EmpPhoneExpenses
```

The error message of Code Snippet 22 is shown in Figure 6.10.

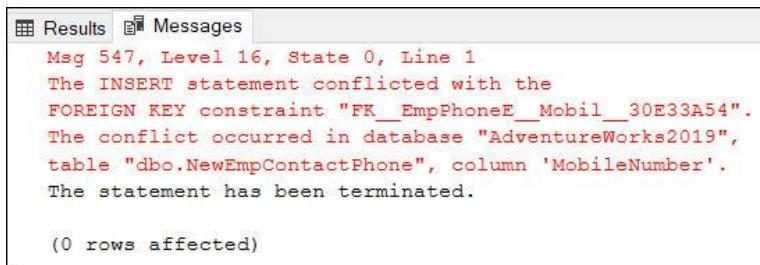


Figure 6.10: Output Error Message of FOREIGN KEY REFERENCES

If there is no key in the referenced table having a value that is being inserted into the foreign key, the insertion will fail as shown in Figure 6.10. It is, however, possible to add `NULL` value into a foreign key column.

6.4.4 CHECK

A `CHECK` constraint limits the values that can be placed in a column. Check constraints enforce integrity of data. For example, a `CHECK` constraint can be given to check if the value being entered into `VoterAge` is greater than or equal to 18. If the data being entered for the column does not satisfy the condition, then, insertion will fail.

A `CHECK` constraint operates by specifying a search condition, which can evaluate to `TRUE`, `FALSE`, or `UNKNOWN`. Values that evaluate to `FALSE` are rejected. Multiple `CHECK` constraints can be specified for a single column. A single `CHECK` constraint can also be applied to multiple columns by creating it at the table level.

Code Snippet 23 demonstrates creating a `CHECK` constraint to ensure that `Amount` value will always be non-zero. A `NULL` value can, however, be added into `Amount` column if the value of `Amount` is not known.

Code Snippet 23:

```
CREATE TABLE NewEMP_PhoneExpenses (Expense_ID int PRIMARY KEY, MobileNumber  
bigint, Amount bigint CHECK (Amount >10))
```

Once a `CHECK` constraint has been defined, if an `INSERT` statement is written with data that violates the constraint, it will fail as shown in Code Snippet 24.

Code Snippet 24:

```
INSERT INTO dbo.NewEMP_PhoneExpenses values (101, 983345674, 9)
```

The error message of Code Snippet 24 that appears when the **Amount** value is less than 10 is shown in Figure 6.10.

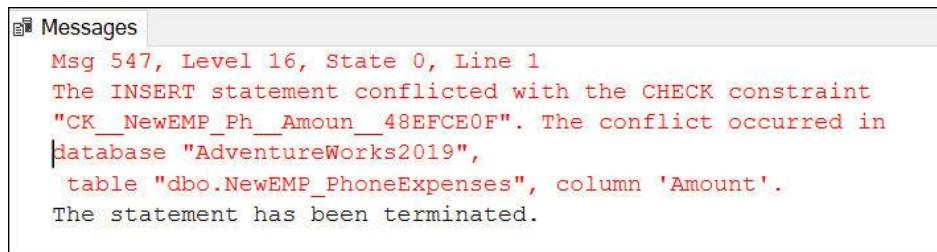


Figure 6.10: Output Error Message of CHECK Constraint

6.4.5 NOT NULL

A **NOT NULL** constraint enforces that the column will not accept null values. The **NOT NULL** constraints are used to enforce domain integrity, similar to **CHECK** constraints.

6.5 Check Your Progress

1. Which of the following feature of a column determines whether rows in the table can contain a null value for that column?

(A)	Default	(C)	Group BY
(B)	Multiplicity	(D)	Nullability

2. A _____ in a table is a column that points to a primary key column in another table.

(A)	Foreign key	(C)	Repeated key
(B)	Secondary key	(D)	Local key

3. Which of the following code is used to drop a table from **AdventureWorks2019** database?

(A)	DROPTABLE [dbo].[Table_1]	(C)	USE [AdventureWorks2019] GO DELETE TABLE [dbo].[Table_1]
(B)	USE [AdventureWorks2019] GO DROP TABLE [dbo].[Table_1]	(D)	USE [AdventureWorks2019] GO DROP [dbo].[Table_1]

4. Which of the following property of SQL Server is used to create identifier columns that can contain auto-generated sequential values to uniquely identify each row within a table?

(A)	SELECT	(C)	INSERT
(B)	IDENTITY	(D)	DEFAULT

5. A _____ constraint is used to ensure that only unique values are entered in a column or set of columns.

(A)	UNIQUE	(C)	Foreign key
(B)	DEFAULT	(D)	INSERT

6.5.1 Answers

1.	D
2.	A
3.	B
4.	B
5.	A



Summary

- Most tables have a primary key, made up of one or more columns of the table that identifies records uniquely.
- The nullability feature of a column determines whether rows in the table can contain a null value for that column.
- A DEFAULT definition for a column can be created at the time of table creation or added at a later stage to an existing table.
- The IDENTITY property of SQL Server is used to create identifier columns that can contain auto-generated sequential values to uniquely identify each row within a table.
- Constraints are used to apply business logic rules and enforce data integrity.
- A UNIQUE constraint is used to ensure that only unique values are entered in a column or set of columns.
- A foreign key in a table is a column that points to a primary key column in another table.
- A CHECK constraint limits the values that can be placed in a column.

Try It Yourself

1. Saint Clara Insurance (SCI) services is a leading Insurance company based in New York, USA. SCI Services wanted a faster, more accurate, and less expensive way to handle insurance claims adjusting for its insurance company customers. With an ever-increasing customer base, they decided to create a Web-based application that will be used not only by employees who work on field, but will also be used by the administrators in the head office.

SCI handles approximately 650 claims per month, but that can soar to 15,000 or more when a hurricane or some other disaster strikes. Officers can use the software on the device type of their choice: Tablet PCs or laptops in the field, or desktop PCs back in their offices. The use of Microsoft SQL Server 2019 as the software's database enables to receive and update all the necessary information regarding a customer or claimer.

With thousands of customers expected every month, data integrity of the data in the database is very important. You must perform the following tasks:

- a. Create a database called **SaintClaraServices** to store the details of the company. Create a table **CustomerHeader** with the details given in table 6.2.

Field Name	Data Type	Description
ClientID	int	Stores client id. This column is the Primary Key
FirstName	char	Stores first name of the client
LastName	char	Stores last name of the client
MiddleName	char	Stores middle name of the client
Gender	char	Stores gender of the client
DateOfBirth	datetime	Stores date of birth of the client
Address	varchar(max)	Stores address of the client
MaritalStatus	char	Stores marital status of the client
Age	int	Stores age of the client
Employment	char	Stores occupation of the client
CompanyName	varchar(max)	Stores the company name
CompanyAddress	varchar(max)	Stores the company address

Table 6.2: CustomerHeader Table

Try It Yourself

- b. Create a table **CustomerDetails** with the specifications given in table 6.3.

Field Name	Data Type	Description
ClientID	int	Stores client id
FatherName	char	Stores the name of the client's father
MotherName	char	Stores the name of the client's mother
Amount	money	Stores the principal amount
Period	int	Stores period for insurance
Plan	char	Stores plan for insurance
Premium	money	Stores premium
NomineeName	char	Stores nominee name
Date	datetime	Stores the date on which insurance is made

Table 6.3: CustomerDetails Table

- c. Add a foreign key constraint for **ClientID** to **CustomerDetails** table.
- d. Create a **CHECK** constraint in **CustomerDetails** table to ensure that the **Amount** value will always be non-zero.
- e. Add **NOT NULL** constraint to Period, Plan, and Premium columns.

Session - 7

Azure SQL

Welcome to the Session, **Azure SQL**. This session explains Azure SQL and its benefits. It also lists differences between Azure SQL and on-premises SQL Server. Finally, the session explains the process to create a cloud database and connect Azure SQL database with SSMS.

In this session, you will learn to:

- Explain Azure SQL
- List the features and benefits of Azure SQL
- State the differences between Azure SQL and on-premises SQL Server
- Explain steps to connect Azure SQL with SSMS

7.1 Introduction

Cloud computing is a technology trend, that involves delivery of software, platforms, and infrastructure as services through the Internet. Microsoft Azure is a key offering in Microsoft's suite of cloud computing products and services. The database functions of Microsoft's cloud platform are provided by Azure SQL.

The data on Azure SQL does not have the constraint of being location-specific. This means that data stored in Azure SQL can be viewed and edited from any location, as the entire data is stored on cloud storage platform.

7.2 Azure SQL

Consider a scenario of the Income Tax department. During the month of March, the department is flooded with heavy workload. During the rest of the year, the workload may be less. As a result, resources, server, and computing power are under-utilized during those months and over-utilized during peak periods. In such a scenario, using a cloud database service such as Microsoft Azure SQL can help in optimal use of resources only as and when required.

Azure SQL is a cloud based relational database service that leverages existing SQL Server technologies. It extends the functionality of Microsoft SQL Server for developing applications that are Web-based, scalable, and distributed. Azure SQL is not just a single product but refers to a family of managed, intelligent, and secure products that use the SQL Server database engine in the Azure cloud.

Azure SQL was earlier known by other names such as SQL Azure, SQL Server Data Services, SQL Services, and Windows Azure SQL Database. From the time it was first released in 2010 and then renamed to Azure SQL in 2014 to the present version, there have been many new features and enhancements added to it.

Azure SQL can be used to store and manage data using queries and other functions that are similar to SQL

Server 2019. Azure SQL also enables users to perform relational queries, search operations, and synchronize data with mobile users and remote back offices. Azure SQL can store and retrieve both structured and unstructured data.

It can also be used in collaboration with other Azure applications, through Visual Studio IDE as shown in figure 7.1.

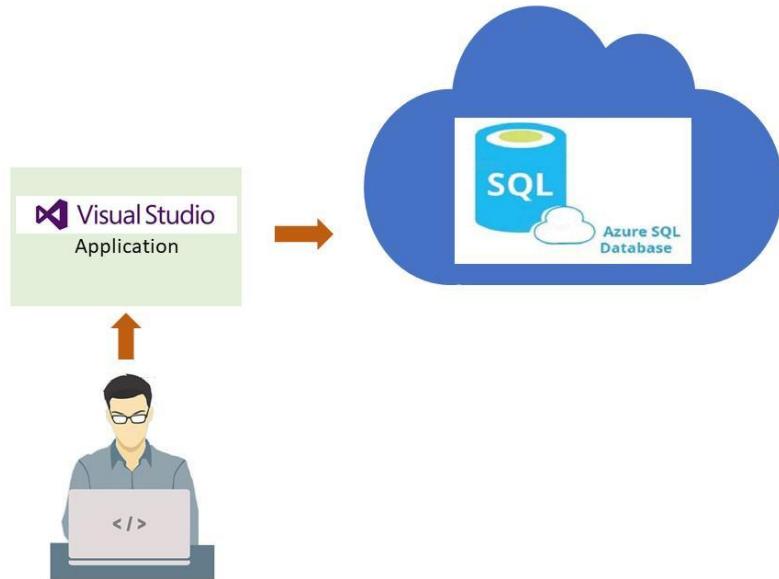
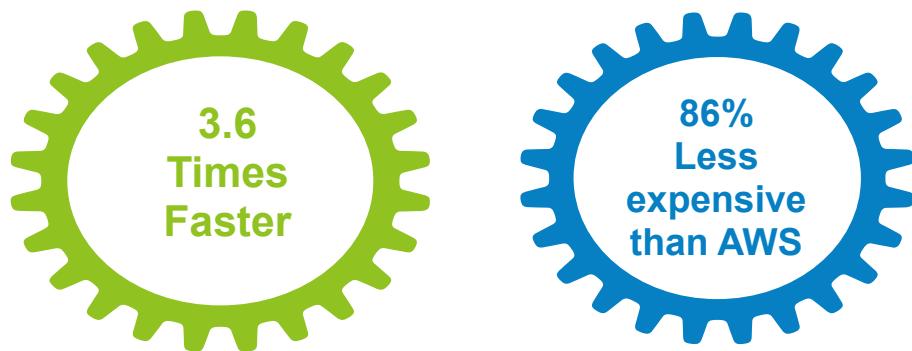


Figure 7.1: Example of Using Azure SQL

One of the competitors to Azure SQL is Amazon Web Services (AWS) and its Relational Database Services (RDS) product. Azure SQL is often compared to AWS RDS.

Azure SQL is:



Both cloud based as well as on-premises applications can use the Azure SQL database. Applications retrieve data from Azure SQL through a protocol known as Tabular Data Stream (TDS). This protocol is not new to Azure SQL. Whenever on-premises applications involve interaction with SQL Server Database Engine, this protocol is used by the client and the server.

7.3 Services and Products in the Azure SQL Family

Following are services and products in the Azure SQL family:

SQL Server on Azure Virtual Machines	Azure SQL Managed Instance	Azure SQL Database	Azure SQL Edge
Facilitates migration of existing apps or building new apps on the cloud for mission-critical SQL Server workloads.	Is the intelligent, scalable, cloud database service combining the broadest SQL Server engine compatibility with benefits of a fully managed platform as a service.	Is the intelligent, scalable, relational database service built for the cloud. It is always up to date, with AI-powered and automated features that optimize performance.	Is a small-footprint, edge-optimized SQL database engine with built-in Artificial Intelligence (AI).

Figure 7.2 outlines these services along with depicting their purpose.

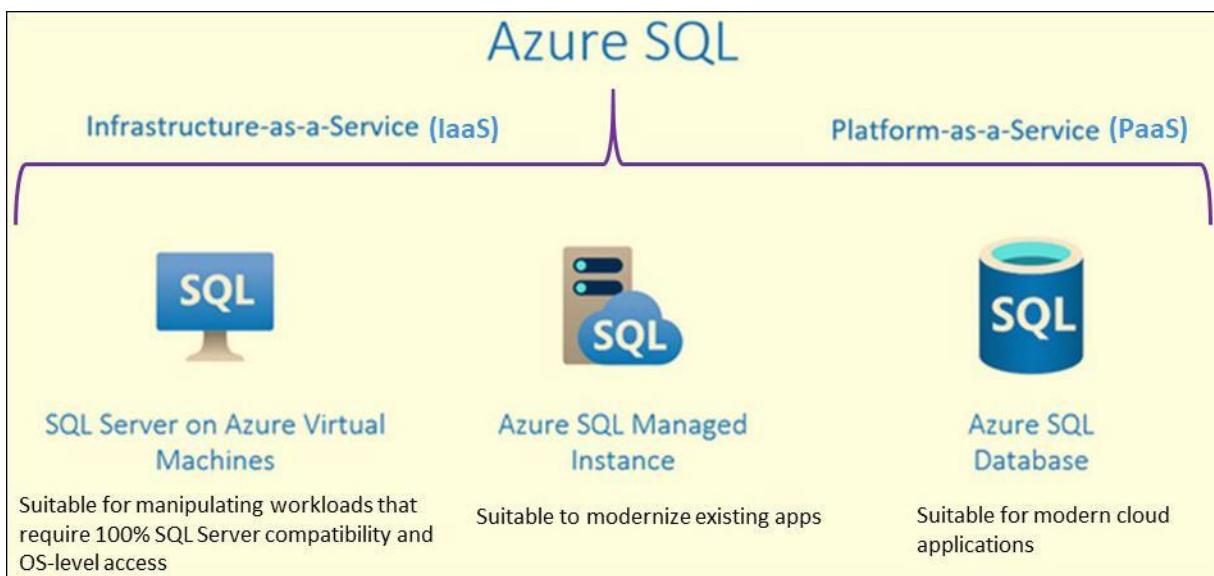


Figure 7.2: Azure SQL Services

SQL Server in a VM is ideal for existing applications that require full product functionality. It offers up to 20,000 Input/output Operations Per Second (IOPS) in terms of scalability.

Azure SQL Database on the other hand is more suited for applications that require elastic scale and/or lesser overhead. Azure SQL Database can scale out to thousands of databases and process terabytes of data.

7.4 Azure SQL Architecture

Azure SQL Architecture comprises four key layers, as shown in figure 7.3.

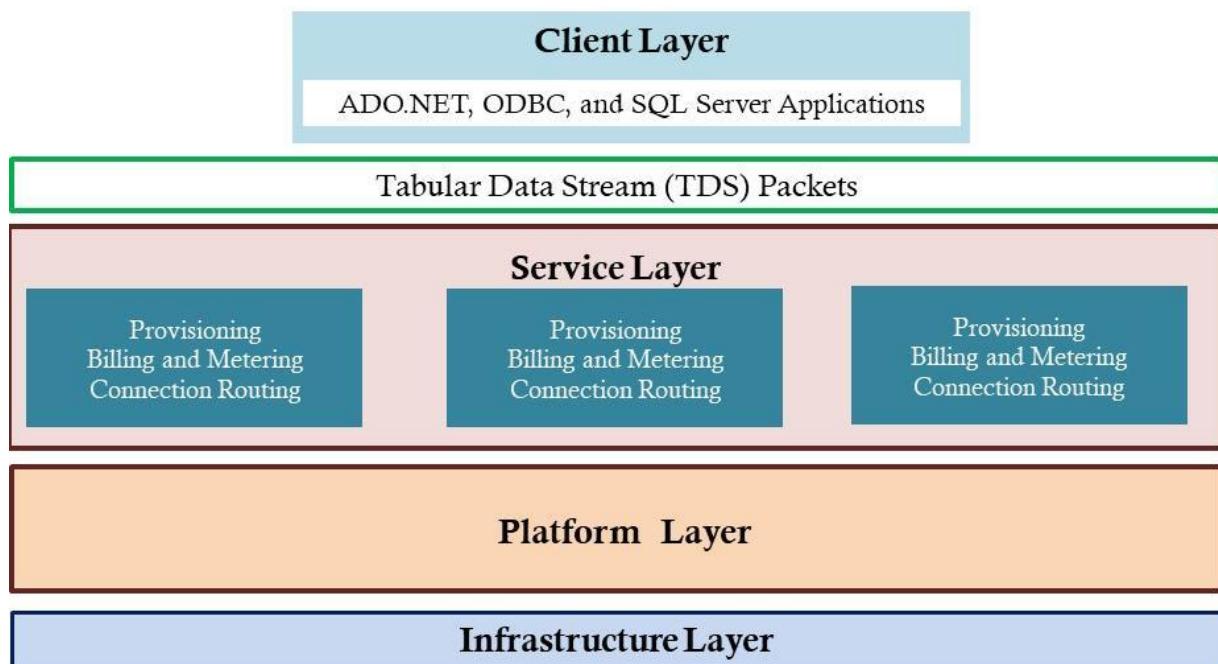


Figure 7.3: Azure SQL Architecture

Client Layer

Acts as an interface for applications. Includes SQL Server tools, Open Database Connectivity (ODBC), ADO.NET, and Hypertext Preprocessor (PHP).

TDS transfers data between applications and SQL Databases and also communicates with applications.

Platform Layer

Includes systems (data nodes) that host actual Azure SQL Server in the data center. Each SQL Database is stored in one of the nodes and is replicated twice across two different physical servers. Azure SQL ensures that multiple copies of servers are kept within the Azure Cloud. It also ensures that the copies are synchronized when data in them is manipulated.

Service Layer

Lies in between platform and client layers and acts as a doorway between the two. It validates Microsoft Azure SQL Database requests and authenticates a user. Also, it establishes a connection between client and server and routes packets using this connection.

Infrastructure Layer

Is the bottom-most layer of the architecture and is responsible for administration of OS and physical hardware.

7.5 Features and Benefits of Azure SQL

Features and benefits of Azure SQL are as follows:

Lower cost

Azure SQL provides several features similar to on-premises SQL Server at a lower cost. Also, as Azure SQL is on the cloud platform, it can be accessed from any location. Hence, there is no additional cost required to develop a dedicated IT infrastructure and department to manage the databases.

Scalability

Elastic Pool feature in Azure SQL allows you to assign a shared set of compute resources to a collection of Azure SQL databases. The benefit of this is that, a single database can be moved in and out of an elastic pool, which gives us flexibility and in turn, achieve cost efficiency.

Usage of TDS

TDS is used in on-premises SQL Server databases for client libraries. Hence, most developers are familiar with TDS and its use. The same kind of TDS interface is used in Azure SQL to build client libraries. Hence, it is easier for developers to work on Azure SQL.

Automatic failover measures

Azure SQL stores multiple copies of data on different physical locations. Even if there is a hardware failure due to heavy usage or excessive load, Azure SQL helps to maintain business operations by providing availability of data through other physical locations. This is done by using automatic failover measures that are provided in Azure SQL.

Flexibility in service usage

Even small organizations can use Azure SQL as pricing model for Azure SQL is based on the storage capacity that is used by an organization. If the organization requires more storage, the price can be altered to suit the requirement. This helps organizations to be flexible in the investment depending on the service usage.

Transact-SQL support

As Azure SQL is completely based on the relational database model, it also supports Transact-SQL operations and queries. This concept is similar to the working of the on-premises SQL Servers. Hence, administrators do not need any additional training or support to use Azure SQL.

7.6 Difference between Azure SQL and On-Premises SQL Server

Major difference between Azure SQL and on-premises SQL Server is the presence of physical hardware and storage. Some other key distinctions between Azure SQL and on-premises SQL Server are as follows:

Tools

On-premises SQL Server provides a number of tools for monitoring and management. All these tools may not be supported by Azure SQL, as there are a limited set of tools that are available in this version.

Backup

Backup and restore function must be supported in on-premises SQL Server for disaster recovery. For Azure SQL, as all the data is on the cloud platform, backup and restore is not required.

USE statement

USE statement is not supported by Azure SQL. Hence, the user cannot switch between databases in Azure SQL as compared to on-premises SQL Server.

Authentication

Azure SQL supports only SQL Server authentication and on-premises SQL Server supports both SQL Server authentication and Windows Authentication.

Transact-SQL support

Not all Transact-SQL functions are supported by Azure SQL.

Accounts and Logins

In Azure SQL, administrative accounts are created in the Azure management portal. Hence, there are no separate instance-level user logins.

Firewalls

Firewalls settings for allowed ports and IP addresses can be managed on physical servers for on-premises SQL Server. As an Azure SQL database is present on cloud, authentication through logins is the only method to verify the user.

7.7 Azure SQL Pricing

There are two types of approaches one can use to work with Azure SQL:

1. Sign up for an Azure Free Account and get credit for 30 days and get 12 months of free access to Azure SQL Database. After 30 days, however, your Azure account will be charged.
2. Alternatively, opt for a paid purchasing model. This approach is complicated because various database service-tier options such as Database Transaction Units (DTUs), maximum database size, and disaster recovery options are used to determine pricing instead of hardware (CPU/RAM/HD). A DTU is a metric that defines a mixture of CPU, memory, and read/write rates.

Paid purchasing models in turn have two categories:

Virtual core (vCore)-based purchasing model

Is available for both Azure SQL Database and Azure SQL Managed Instance.

Provides a choice between a provisioned compute tier and a serverless compute tier. With the provisioned compute tier, you can choose exact amount of compute resources provisioned for workload. With serverless compute tier, you specify autoscaling of the compute resources over a range.

DTU-based purchasing model

Provides bundled compute and storage packages balanced for common workloads. DTU-based purchasing model is available for Azure SQL Database.

Refer to the Website <http://dtucalculator.azurewebsites.net/> to determine DTU based costs.

For a single database, the vCore-based purchase model is ideal as it is flexible and allows you to scale memory and storage based on your requirements. Using the serverless compute tier optimizes price and performance.

7.8 Connect to Azure SQL with SSMS

To access Azure SQL with SSMS, a Microsoft Azure account must be created. To begin with, you can opt for a free trial account for 30 days.

1. Type the address <http://portal.azure.com> in the Address bar of your browser. You will be redirected to a page to enter your login credentials. Refer to Figure 7.4.

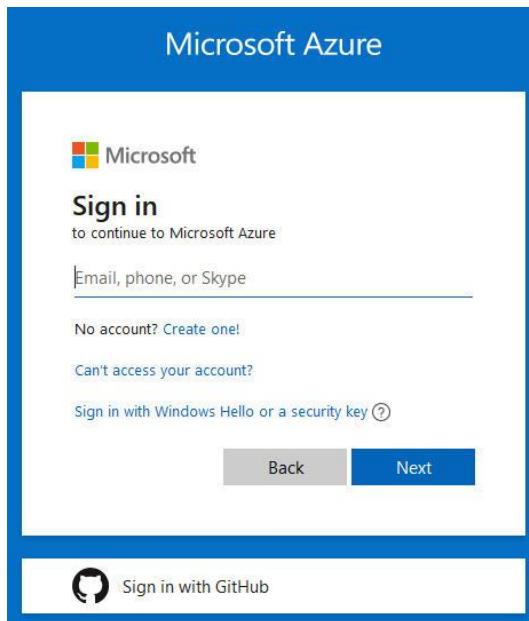


Figure 7.4: Signing Into Microsoft Account

2. If you already have a Microsoft account, you can use that to sign in. Refer to figure 7.5.

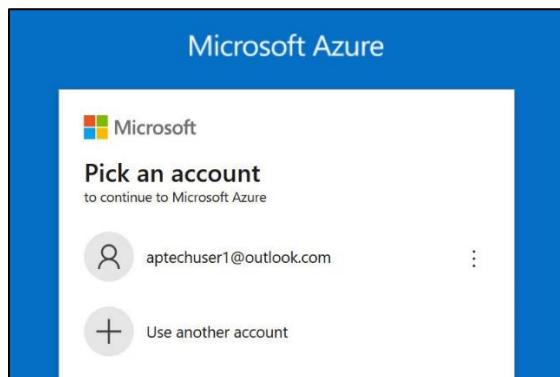


Figure 7.5: Signing Up For Azure Account

The Azure Portal page will open up and you will be prompted with a variety of actions as shown in figure 7.6.

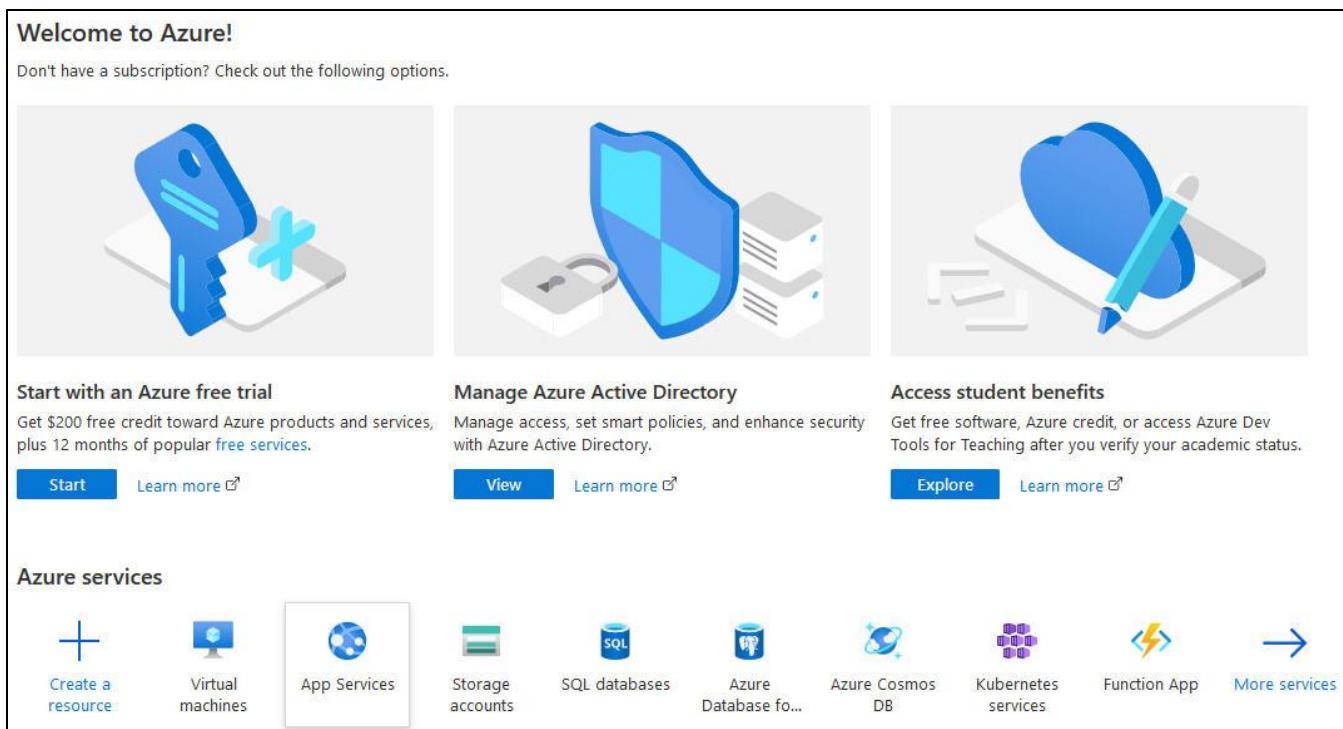


Figure 7.6: Azure Portal

Click **Start** under **Start with an Azure free trial**. You will be asked to fill in your profile details and verify your identity through phone and credit card as shown in figures 7.7, 7.8, and 7.9 (steps 1, 2, and 3).

1 Your profile

Country/Region i
 ▼

Choose the location that matches your billing address. You cannot change this selection later. If your country is not listed, the offer is not available in your region.
[Learn More](#)

First name

Last name

Email address i

Phone
Example: (425) 555-0100

Next

2 Identity verification by phone

A text or phone call helps us make sure this is you.

Country code ▼

Phone number

Text me **Call me**

3 Identity verification by card

4 Agreement

Figure 7.7: Entering Profile

1 Your profile

2 Identity verification by phone

A text or phone call helps us make sure this is you.

Country code ▼

Phone number

Text me **Call me**

3 Identity verification by card

4 Agreement

Figure 7.8: Verifying Identity Details

3 Identity verification by card

We'll make a temporary authorization on this card, but **you won't be charged unless you upgrade.**

We accept the following cards:

Cardholder Name

Card number

Expires

 MM YY

CVV

Address line 1

Address line 2 (Optional)

City

State

ZIP code

Country/Region

Next

Figure 7.9: Identity Verification by Credit Card

Finally, upon successful verification of your credit card, your account will be set up. You will see various services available under Azure services. Click **SQL databases** under Azure services as shown in figure 7.10.

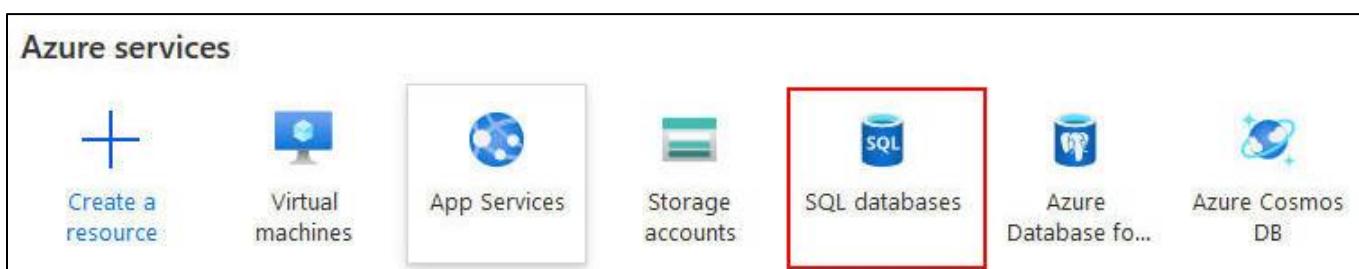


Figure 7.10: Azure Services

The **SQL databases** page will be displayed as shown in figure 7.11. Here, you can create and work with

Azure SQL databases.

The screenshot shows the Microsoft Azure portal interface for managing SQL databases. At the top, there's a navigation bar with 'Microsoft Azure' and a search bar. Below it, the 'SQL databases' section is displayed under 'Default Directory'. A prominent message encourages users to try the new Azure SQL resource browser. The main area shows a table header for 'Subscriptions: Free Trial' with columns: Name (sorted by Name), Status, Replication role, Server, Pricing tier, and Location. Below the table, a large blue cylinder icon with the letters 'SQL' is centered, accompanied by the text 'No SQL databases to display' and a note to 'Try changing your filters if you don't see what you're looking for.' A blue 'Create SQL database' button is located at the bottom of this section.

Figure 7.11: SQL Databases Page

Click **Create SQL database** at the bottom of the page. You will be asked to fill up information such as database name and server name. Refer to Figure 7.12.

Home > SQL databases >

Create SQL Database

Microsoft

Basics • Networking Additional settings Tags Review + create

Create a SQL database with your preferred configurations. Complete the Basics tab then go to Review + Create to provision with smart defaults, or visit each tab to customize. [Learn more](#)

Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * ⓘ Resource group * ⓘ [Create new](#)

Database details

Enter required settings for this database, including picking a logical server and configuring the compute and storage resources

Database name *

Server * ⓘ [Create new](#)

✖ The value must not be empty.

Want to use SQL elastic pool? * ⓘ Yes No

Compute + storage * ⓘ [Configure database](#)

[Review + create](#) [Next : Networking >](#)

Figure 7.12: Entering New Database Details

Since a database server does not exist yet, you will create a new one as shown in figure 7.13.

New server X

Microsoft

Server name * .database.windows.net

Server admin login *

Password *

Confirm password *

Location *

Figure 7.13: Creating a New Database Server

Observe the rules for the login name shown in figure 7.14.



Figure 7.14: Rules for Server Admin Login Name

Ensure that you specify valid details. Refer to Figure 7.15 for an example.

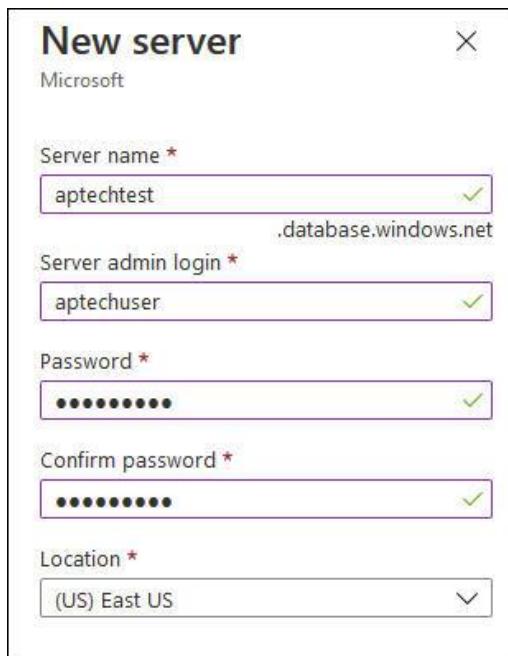


Figure 7.15: Specifying Valid Details for the New Server

When you finish creating the new server and specify that server name, click **Review+create** on **Create SQL Database** page. The database will be successfully created and a dashboard will be displayed as shown

in figure 7.16.

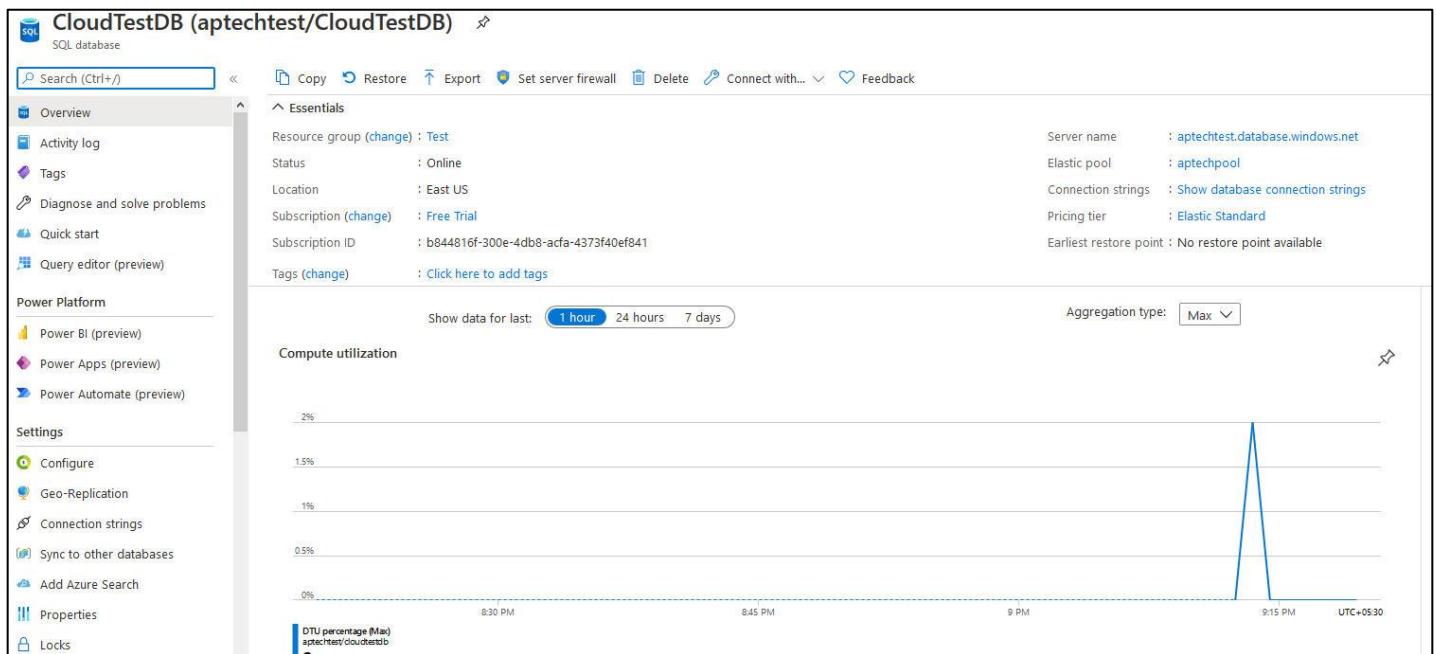


Figure 7.16: Database Dashboard

Now that the database is ready, you can use SSMS to work with it, just as you do with on-premises SQL Server databases.

The process of connecting Azure SQL with SSMS is as follows:

1. Sign in to your Microsoft Azure account online.
2. Open Microsoft SQL Server Management Studio.
3. In the **Connect to Server** dialog box, specify the name of the Azure SQL server as shown in figure 7.17. Each user account of Azure SQL has a specific Server name.

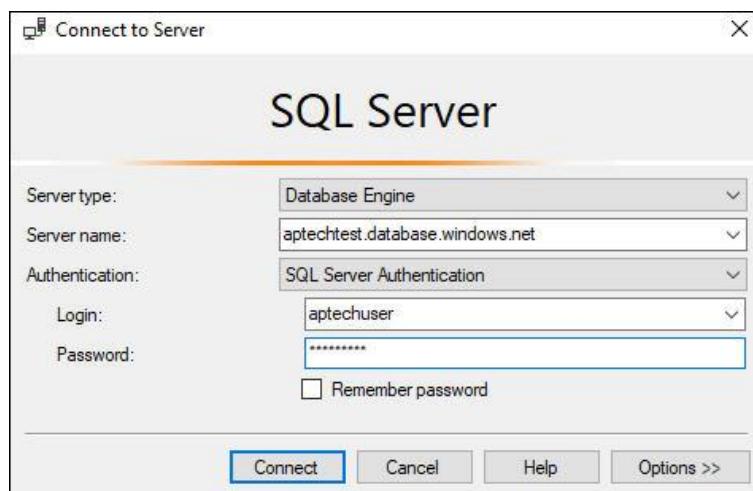


Figure 7.17: Connect to Server Dialog Box with Azure SQL Server

4. In the **Authentication** box, select **SQL Server Authentication**.

5. In the **Login** box, type the name of the Azure SQL administrator account and the password.
6. Click **Connect**. You may then be prompted to create a new firewall rule as shown in figure 7.18.

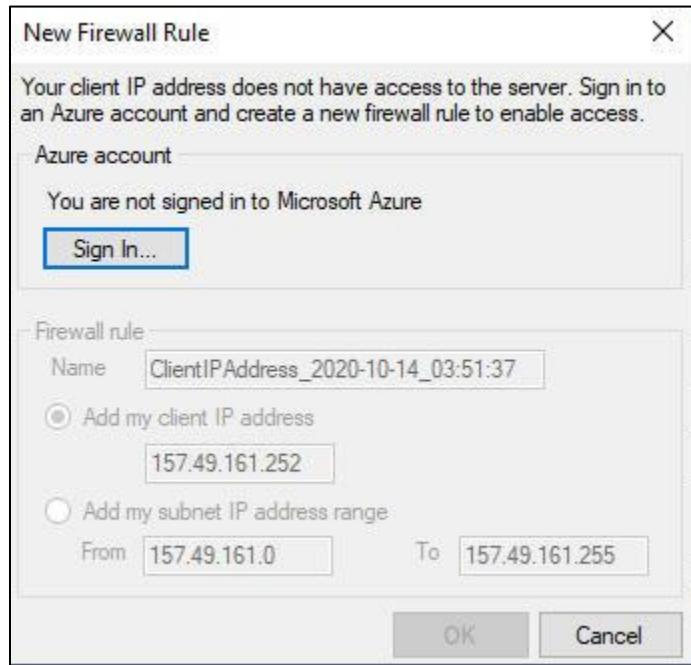


Figure 7.18: Creating New Firewall Rule

7. Sign in to Azure by clicking **Sign In**. Your client IP address will be automatically populated in the corresponding box as shown in figure 7.19.

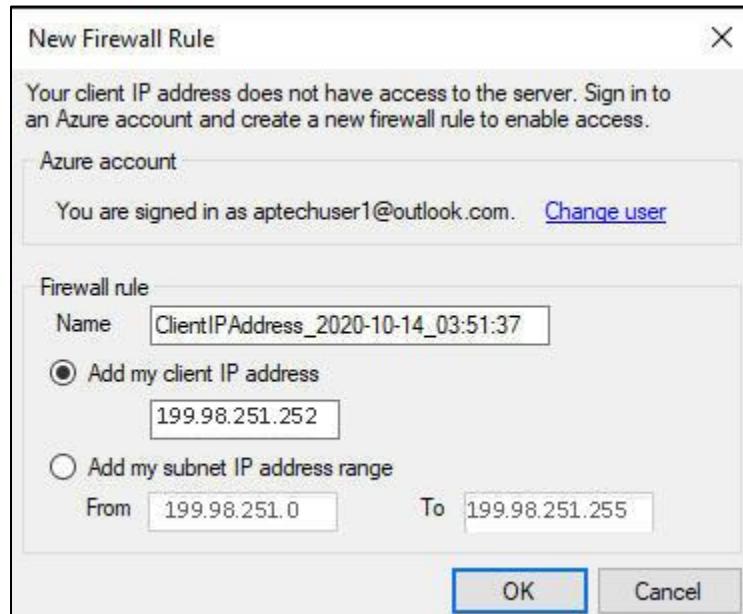


Figure 7.19: Adding Client IP Address to the Rule

8. Click **Connect**. The connection to the database is successfully established. You will see the Azure SQL Database in the Object Explorer.

9. Right-click the **Tables** node and click **New→Table** as shown in figure 7.20.

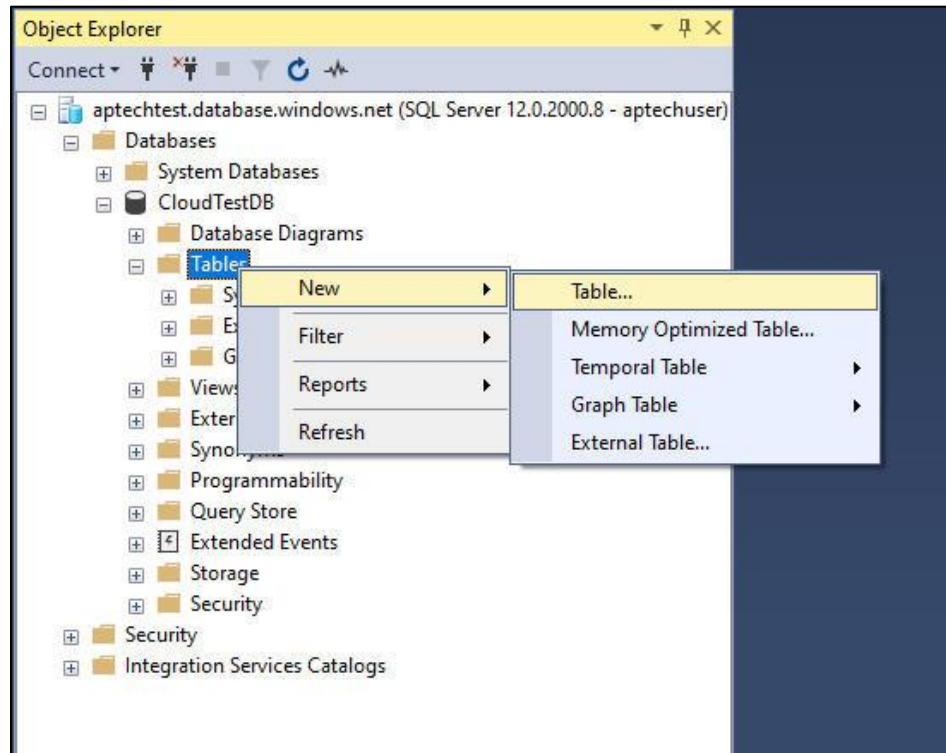


Figure 7.20: Creating a New Table

10. Add columns and save the table as shown in figure 7.21.

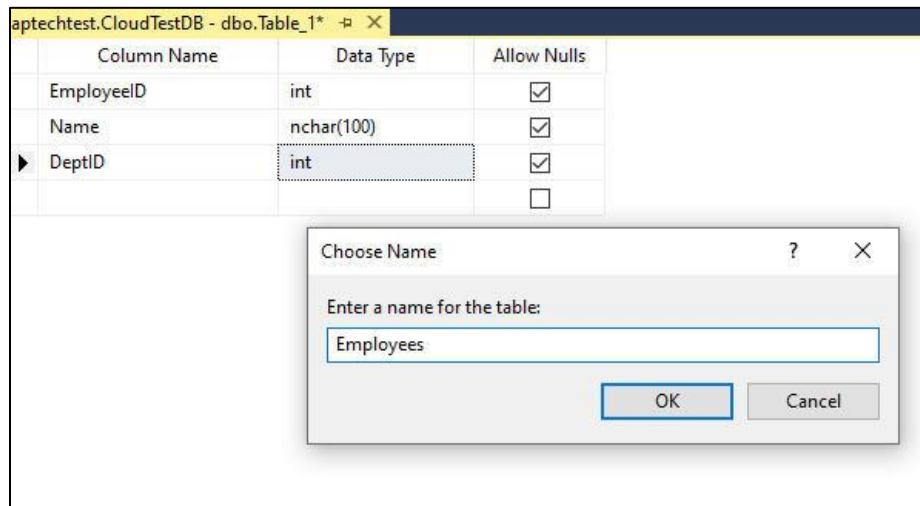


Figure 7.21: Specifying Columns and Table Name

11. Open the Azure portal and in the SQL Database dashboard, click Query Editor on the left pane.

12. You will be asked to authenticate your user credentials. Specify the login name and password as shown in figure 7.22.

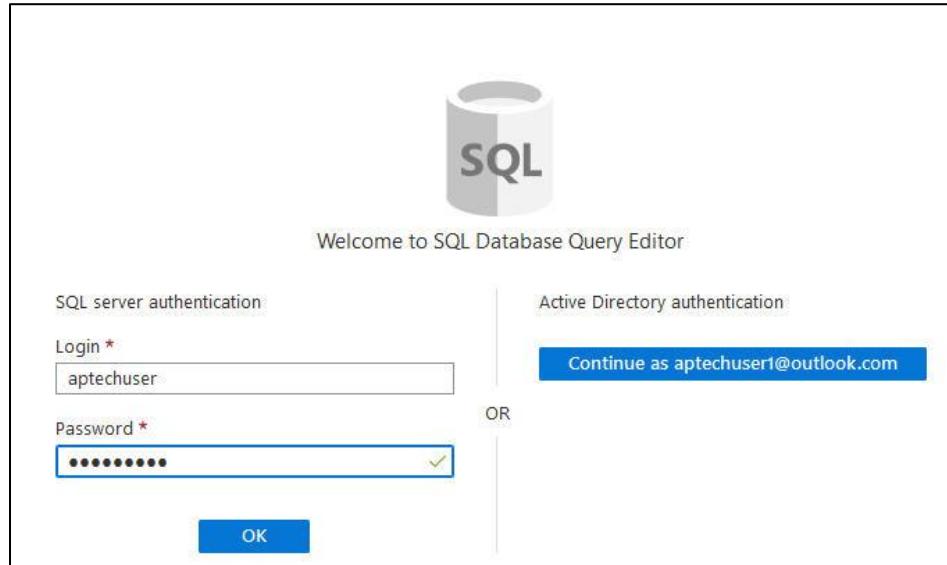


Figure 7.22: Specifying Login Details for SQL Database Query Editor

13. The Query Editor on the cloud will be displayed. Observe in figure 7.23 that the table you created in SSMS is now visible on the left pane. Thus, you created a table on the cloud SQL Server database using local SSMS.

A screenshot of the Azure Cloud Test DB (aptechuser) interface. On the left, the Object Explorer shows 'CloudTestDB (aptechuser)' with a note: 'Showing limited object explorer here. For full capability please open SSDT.' It lists 'Tables' (including 'dbo.Employees'), 'Views', and 'Stored Procedures'. On the right, the 'Query 1' editor window is open with the title bar 'Query 1'. The editor contains a single digit '1' in a text area. Below the editor are tabs for 'Results' and 'Messages', and a search bar 'Search to filter items...'. The status bar at the bottom shows the connection details: 'CloudTestDB (aptechuser)'.

Figure 7.23: Query Editor Showing Newly Created Table

Now, you can perform similar other tasks, including creating and executing queries from SSMS on cloud databases.

7.9 Check Your Progress

1. Which of these are services under Azure SQL?

(A)	SQL Server on Azure Virtual Machines	(C)	Azure SQL Database
(B)	Azure SQL Managed Instance	(D)	All of these

2. Which one of the following protocols is used by applications retrieve data from SQL Server?

(A)	ABS	(C)	TDS
(B)	DTS	(D)	WSQL

3. Which of the following authentication is required to connect to Azure SQL?

(A)	Windows Authentication	(C)	System Administrator Authentication
(B)	SQL Server Authentication	(D)	No Authentication

4. Which layer in the Azure SQL architecture offers Provisioning, Billing and Metering, and Connection Routing?

(A)	Client Layer	(C)	Infrastructure Layer
(B)	Service Layer	(D)	None of these

5. Identify the statements that are true:

- USE statement is not supported by Azure SQL. Hence, the user cannot switch between databases in Azure SQL as compared to on-premises SQL Server.
- For a single database, the vCore-based purchase model is ideal as it is flexible and allows you to scale memory and storage based on your requirements.
- Automatic failover measures are provided in Azure SQL.
- All Transact-SQL functions are supported by Azure SQL.
- In Azure SQL, there are separate instance-level user logins.

(A)	a, b, c	(C)	a, c, d
(B)	c, d, e	(D)	c, d

6. TDS stands for _____.

(A)	Tabular Data Stream	(C)	Tabular Distinction Stream
(B)	Tabular Data System	(D)	Tabular Direct Stream

7.9.1 Answers

1.	D
2.	C
3.	B
4.	B
5.	A
6.	A



Summary

- Microsoft Azure SQL is a cloud based relational database service that leverages existing SQL Server technologies.
- Azure SQL enables users to perform relational queries, search operations, and synchronize data with mobile users and remote back offices.
- Azure SQL can store and retrieve both structured and unstructured data.
- Applications retrieve data from Azure SQL through a protocol known as Tabular Data Stream (TDS).
- SQL Server on Azure Virtual Machines, Azure SQL Database, Azure SQL Managed Instance, and Azure SQL Edge are the services under Azure SQL.
- Users can connect to Azure SQL using SSMS.

Try It Yourself

1. Create a database named **Library** on Azure SQL.
2. Then, using SSMS, create a **Members** table in this database with following columns:

CardNo: 5 characters
LastName: up to 15 characters
FirstName: up to 15 characters
Address: up to 150 characters
DOB: date
Gender: 1 char (M or F)
Phone_No: up to 15 characters

3. Then, create a **Books** table with following columns:

BookID: up to 5 characters
Title: up to 200 characters
Author: up to 100 characters
Publish Date: date

Verify using Azure SQL Query Editor whether the tables have been created.

4. Alter the tables to add **CardNo** as primary key in **Members** and **BookID** as primary key in **Books** table.
5. Insert some rows into the **Members** and **Books** tables on the cloud database, **Library**.
6. Display records from **Members** table.
7. Display records from **Books** table.
8. Create a database named **Supermarket** on Azure SQL.
9. Create tables **Customer**, **Products**, and **ProductOrders** in this database with following structures:

Customer

CustNo: 5 characters
FirstName: up to 45 characters
LastName: up to 45 characters
Address: up to 150 characters
DOB: date
Gender: 1 char (M or F)
Phone_No: up to 15 characters

Products

ProductNo: int, primary key
Name: up to 45 characters
Category: up to 15 characters
Manufacturer: up to 45 characters

ProductOrders

ProductNo: int, foreign key
OrderDate: date
Quantity: int
CustNo: 5 characters

10. Insert rows into these tables.

Session - 8

Accessing Data

Welcome to the Session, **Accessing Data**.

This session describes SELECT statement. It also explains the expressions and various clauses used with SELECT statement. Finally, the session introduces the new `xml` data type and describes how to work with XML data in SQL Server 2019 tables.

In this session, you will learn to:

- Describe SELECT statement, its syntax, and use
- Explain various clauses used with SELECT
- State the use of ORDER BY clause
- Describe working with typed and untyped XML
- Explain the procedure to create, use, and view XML schemas

8.1 Introduction

The `SELECT` statement is a core command used to access data in SQL Server 2019. XML allows developers to develop their own set of tags and makes it possible for other programs to understand these tags. XML is the preferred means for developers to store, format, and manage data on the Web.

8.2 SELECT Statement

A table with its data can be viewed using the `SELECT` statement. The `SELECT` statement in a query will display the required information in a table. The `SELECT` statement retrieves rows and columns from one or more tables. The output of the `SELECT` statement is another table called resultset. The `SELECT` statement also joins two tables or retrieves a subset of columns from one or more tables. The `SELECT` statement defines the columns to be used for a query. The syntax of `SELECT` statement can consist of a series of expressions separated by commas. Each expression in the statement is a column in the resultset. The columns appear in the same sequence as the order of the expression in the `SELECT` statement. The `SELECT` statement retrieves rows from the database and enables the selection of one or many rows or columns from a table. Following is the syntax for the `SELECT` statement.

Syntax:

```
SELECT <column_name1>...<column_nameN> FROM <table_name>
```

where,

`table_name`: is the table from which the data will be displayed.

`<column_name1>...<column_nameN>`: are the columns that are to be displayed.

8.2.1 SELECT Without FROM

Many SQL versions use `FROM` in their query, but in all the versions from SQL Server 2005, including SQL Server 2019, one can use `SELECT` statements without using the `FROM` clause. Code Snippet 1 demonstrates the use of `SELECT` statement without using the `FROM` clause.

Code Snippet 1:

```
SELECT LEFT('International', 5)
```

The code will display only the first five characters from the extreme left of the word 'International'.

The output is shown in figure 8.1.

The screenshot shows the SSMS Results pane. It has two tabs at the top: 'Results' (which is selected) and 'Messages'. The results grid contains one row with one column. The column header is '(No column name)' and the data value is 'Inter'. There is also a row number '1' on the far left.

(No column name)
1 Inter

Figure 8.1: First Five Characters from the Extreme Left of the Word

8.2.2 Displaying All Columns

The asterisk (*) is used in the `SELECT` statement to retrieve all the columns from the table. It is used as a shorthand to list all the column names in the table named in the `FROM` clause. Following is the syntax for selecting all columns.

Syntax:

```
SELECT * FROM <table_name>
```

where,

`*`: specifies all columns of the named tables in the `FROM` clause.

`<table_name>`: is the name of the table from which the information is to be retrieved. It is possible to include any number of tables. When two or more tables are used, the row of each table is mapped with the row of others. This activity takes a lot of time if the data in the tables are huge. Hence, it is recommended to use this syntax with a condition.

Code Snippet 2 demonstrates the use of '`*`' in the `SELECT` statement.

Code Snippet 2:

```
USE AdventureWorks2019
SELECT * FROM HumanResources.Employee
GO
```

The partial output of Code Snippet 2 with some columns of `HumanResources.Employee` table is shown in figure 8.2.

	BusinessEntityID	NationalIDNumber	LoginID	OrganizationNode	OrganizationLevel	JobTitle
1	1	295847284	adventure-works\ken0	NULL	NULL	Chief Executive Officer
2	2	245797967	adventure-works\terri0	0x58	1	Vice President of Engineering
3	3	509647174	adventure-works\roberto0	0x5AC0	2	Engineering Manager
4	4	112457891	adventure-works\rob0	0x5AD6	3	Senior Tool Designer
5	5	695256908	adventure-works\gail0	0x5ADA	3	Design Engineer
6	6	998320692	adventure-works\jossef0	0x5ADE	3	Design Engineer
7	7	134969118	adventure-works\dylan0	0x5AE1	3	Research and Development Manager
8	8	811994146	adventure-works\diane1	0x5AE158	4	Research and Development Engineer
9	9	658797903	adventure-works\gigi0	0x5AE168	4	Research and Development Engineer
10	10	879342154	adventure-works\michael6	0x5AE178	4	Research and Development Manager

Figure 8.2: Displaying All Columns

8.2.3 Displaying Selected Columns

The `SELECT` statement displays or returns certain relevant columns that are chosen by the user or mentioned in the statement. To display specific columns, knowledge of the relevant column names in the table is required. Following is the syntax for selecting specific columns.

Syntax:

```
SELECT <column_name1>..<column_nameN> FROM <table_name>
```

where,

`<column_name1>..<column_nameN>`: are the columns that are to be displayed.

For example, to display the cost rates in various locations from `Production.Location` table in `AdventureWorks2019` database, the `SELECT` statement is shown in Code Snippet 3.

Code Snippet 3:

```
USE AdventureWorks2019
SELECT LocationID, CostRate FROM Production.Location
GO
```

Figure 8.3 shows `LocationID` and `CostRate` columns from `AdventureWorks2019` database.

	LocationID	CostRate
1	1	0.00
2	2	0.00
3	3	0.00
4	4	0.00
5	5	0.00
6	6	0.00
7	7	0.00
8	10	22.50
9	20	25.00
10	30	14.50
11	40	15.75
12	45	18.00
13	50	12.25
14	60	12.25

Figure 8.3: LocationID and CostRate Columns

8.3 Different Expressions with *SELECT*

SELECT statement allows users to specify different expressions in order to view the resultset in an ordered manner. These expressions assign different names to columns in the resultset, compute values, and eliminate duplicate values.

8.3.1 Using Constants in Result Sets

Character string constants are used when character columns are joined. They help in proper formatting or readability. These constants are not specified as a separate column in the resultset. It is usually more efficient for an application to build the constant values into the results when they are displayed, rather than making use of the server to incorporate the constant values. For example, to include ' : ' and '->' in the resultset so as to display the country name, country region code, and its corresponding group, the *SELECT* statement is shown in Code Snippet 4.

Code Snippet 4:

```
USE AdventureWorks2019
SELECT Name + ' : ' + CountryRegionCode + ' -> ' + Group FROM Sales.SalesTerritory
GO
```

In this code, the column name **Group** of **Sales.SalesTerritory** is given in brackets because it must not conflict with the keyword **GROUP** of Transact-SQL.

Figure 8.4 displays the country name, country region code, and corresponding group from **Sales.SalesTerritory** of **AdventureWorks2019** database.

	(No column name)
1	Northwest:US->North America
2	Northeast:US->North America
3	Central:US->North America
4	Southwest:US->North America
5	Southeast:US->North America
6	Canada:CA->North America
7	France:FR->Europe
8	Germany:DE->Europe
9	Australia:AU->Pacific
10	United Kingdom:GB->Europe

Figure 8.4: Country Name, Country Region Code, and Corresponding Group

8.3.2 Renaming ResultSet Column Names

Columns displayed in resultsets of queries have corresponding headings specified in the table. These headings can be changed, renamed, or can be assigned a new name by using `AS` clause. By customizing the headings, they become more understandable and meaningful.

If you observe, the output in figure 8.4 had no column heading. The code in Code Snippet 4 can be modified as given in Code Snippet 5 in order to display a named column heading.

Code Snippet 5:

```
USE AdventureWorks2019
SELECT Name + ':' + CountryRegionCode + '-' + [Group] AS NameRegionGroup FROM
Sales.SalesTerritory
GO
```

Figure 8.5 shows the output.

	NameRegionGroup
1	Northwest:US->North America
2	Northeast:US->North America
3	Central:US->North America
4	Southwest:US->North America
5	Southeast:US->North America
6	Canada:CA->North America
7	France:FR->Europe
8	Germany:DE->Europe
9	Australia:AU->Pacific
10	United Kingdom:GB->Europe

Figure 8.5: Column Heading Modified to NameRegionGroup

Code Snippet 6 demonstrates another example for this. It shows how to display 'ChangedDate' as the heading for **ModifiedDate** column in the **Person.Person** table, the SELECT statement.

Code Snippet 6:

```
USE AdventureWorks2019
SELECT ModifiedDate as 'ChangedDate' FROM Person.Person
GO
```

The output displays 'ChangedDate' as the heading for **ModifiedDate** column in the **Person.Person** table. Figure 8.6 shows the original heading and the changed heading.

Results		Messages
ModifiedDate		ChangedDate
1	2009-01-07 00:00:00.000	2009-01-07 00:00:00.000
2	2008-01-24 00:00:00.000	2008-01-24 00:00:00.000
3	2007-11-04 00:00:00.000	2007-11-04 00:00:00.000
4	2007-11-28 00:00:00.000	2007-11-28 00:00:00.000
5	2007-12-30 00:00:00.000	2007-12-30 00:00:00.000
6	2013-12-16 00:00:00.000	2013-12-16 00:00:00.000
7	2009-02-01 00:00:00.000	2009-02-01 00:00:00.000
8	2008-12-22 00:00:00.000	2008-12-22 00:00:00.000
9	2009-01-09 00:00:00.000	2009-01-09 00:00:00.000
10	2009-04-26 00:00:00.000	2009-04-26 00:00:00.000

Figure 8.6: Column Heading Modified to ChangedDate

8.3.3 Computing Values in ResultSet

A SELECT statement can contain mathematical expressions by applying operators to one or more columns. It allows a resultset to contain values that do not exist in the base table, but which are calculated from the values stored in the base table.

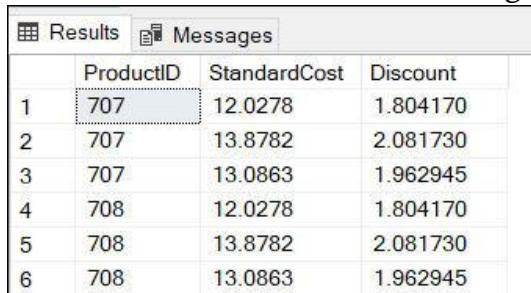
Note - The table used in the **FROM** clause of a query is called as a base table.

For example, consider the table **Production.ProductCostHistory** from **AdventureWorks2019** database. Consider the example where the production people decide to give 15% discount on the standard cost of all the products. The discount amount does not exist but can be calculated by executing the SELECT statement shown in Code Snippet 7.

Code Snippet 7:

```
USE AdventureWorks2019
SELECT ProductID, StandardCost, StandardCost * 0.15 AS Discount FROM
Production.ProductCostHistory
GO
```

Figure 8.7 shows the output where discount amount is calculated using `SELECT` statement.



	ProductID	StandardCost	Discount
1	707	12.0278	1.804170
2	707	13.8782	2.081730
3	707	13.0863	1.962945
4	708	12.0278	1.804170
5	708	13.8782	2.081730
6	708	13.0863	1.962945

Figure 8.7: Calculated Discount Amount

8.3.4 Using `DISTINCT`

The keyword `DISTINCT` prevents the retrieval of duplicate records. It eliminates rows that are repeating from the resultset of a `SELECT` statement. For example, if the `StandardCost` column is selected without using the `DISTINCT` keyword, it will display all the standard costs present in the table. On using the `DISTINCT` keyword in the query, SQL Server will display every record of `StandardCost` only once as shown in Code Snippet 8.

Code Snippet 8:

```
USE AdventureWorks2019
SELECT DISTINCT StandardCost FROM Production.ProductCostHistory
GO
```

8.3.5 Using `TOP` and `PERCENT`

The `TOP` keyword will display only first few set of rows as a resultset. The set of rows is either limited to a number or a percent of rows. The `TOP` expression can also be used with other statements such as `INSERT`, `UPDATE`, and `DELETE`.

Syntax:

```
SELECT [ALL|DISTINCT] [TOP expression [PERCENT] [WITH TIES]]
```

where,

`expression`: is the number or the percentage of rows to be returned as the result.

`PERCENT`: returns the number of rows limited by percentage.

`WITH TIES`: is the additional number of rows that is to be displayed.

The `SELECT` statement has various clauses associated with it.

8.3.6 `SELECT` with `INTO`

The `INTO` clause creates a new table and inserts rows and columns listed in the `SELECT` statement into it.

`INTO` clause also inserts existing rows into the new table. In order to execute this clause with the `SELECT` statement, the user must have the permission to `CREATE TABLE` in the destination database.

Syntax:

```
SELECT <column_name1>..<column_nameN> [INTO new_table] FROM table_list
```

where,

`new_table`: is the name of the new table that is to be created.

Code Snippet 9 uses an `INTO` clause which creates a new table `Production.ProductName` with details such as the product's ID and its name from the table `Production.ProductModel`.

Code Snippet 9:

```
USE AdventureWorks2019
SELECT ProductModelID, Name INTO Production.ProductName FROM Production.
ProductModel
GO
```

After executing the code, a message stating '(128 row(s) affected)' is displayed. If a query is written to display the rows of the new table, the output will be similar to figure 8.8.

	ProductModelID	Name
1	122	All-Purpose Bike Stand
2	119	Bike Wash
3	115	Cable Lock
4	98	Chain
5	1	Classic Vest
6	2	Cycling Cap
7	121	Fender Set - Mountain
8	102	Front Brakes
9	103	Front Derailleur
10	3	Full-Finger Gloves
11	4	Half-Finger Gloves

Figure 8.8: New Table

8.3.7 *SELECT with WHERE*

The `WHERE` clause with `SELECT` statement is used to conditionally select or limit the records retrieved by the query. A `WHERE` clause specifies a Boolean expression to test the rows returned by the query. The row is returned if the expression is true and is discarded if it is false.

Syntax:

```
SELECT <column_name1>...<column_nameN> FROM <table_name> WHERE <search_condition>
```

where,

`search_condition`: is the condition to be met by the rows.

Table 8.1 shows different operators that can be used with the `WHERE` clause.

Operator	Description
=	Equal to
< >	Not equal to
>	Greater than
<	Less than
> =	Greater than or equal to
< =	Less than or equal to
!	Not

Operator	Description
BETWEEN	Between a range
LIKE	Search for an ordered pattern
IN	Within a range

Table 8.1: Operators

Code Snippet 10 demonstrates the equal to operator with `WHERE` clause to display data with `EndDate` '`2013-05-29 00:00:00.000`'.

Code Snippet 10:

```
USE AdventureWorks2019
SELECT * FROM Production.ProductCostHistory WHERE EndDate='2013-05-29 00:00:00.000'
GO
```

Code Snippet 10 will return all records from the table `Production.ProductCostHistory` which has the end date as `2013-05-29 00:00:00.000`.

The output of the `SELECT` with `WHERE` clause will return is shown a large number of rows. Some of the output is shown in figure 8.9.

	ProductID	StartDate	EndDate	StandardCost	ModifiedDate
1	707	2012-05-30 00:00:00.000	2013-05-29 00:00:00.000	13.8782	2013-05-29 00:00:00.000
2	708	2012-05-30 00:00:00.000	2013-05-29 00:00:00.000	13.8782	2013-05-29 00:00:00.000
3	711	2012-05-30 00:00:00.000	2013-05-29 00:00:00.000	13.8782	2013-05-29 00:00:00.000
4	712	2012-05-30 00:00:00.000	2013-05-29 00:00:00.000	5.2297	2013-05-29 00:00:00.000
5	713	2012-05-30 00:00:00.000	2013-05-29 00:00:00.000	29.0807	2013-05-29 00:00:00.000

Figure 8.9: SELECT with WHERE clause

All queries in SQL use single quotes to enclose the text values. For example, consider the following query, which retrieves all the records from Person.Address table having Bothell as city.

Code Snippet 11 demonstrates the equal to operator with WHERE clause to display data with address having Bothell city.

Code Snippet 11:

```
USE AdventureWorks2019
SELECT * FROM Person.Address WHERE City='Bothell'
GO
```

The output of the query will return is shown a large number of rows. Some of the output is shown in figure 8.10.

	AddressID	AddressLine1	AddressLine2	City	StateProvinceID	PostalCode
1	5	1226 Shoe St.	NULL	Bothell	79	98011
2	11	1318 Lasalle Street	NULL	Bothell	79	98011
3	6	1399 Firestone Drive	NULL	Bothell	79	98011
4	18	1873 Lion Circle	NULL	Bothell	79	98011
5	40	1902 Santa Cruz	NULL	Bothell	79	98011
6	1	1970 Napa Ct.	NULL	Bothell	79	98011
7	10	250 Race Court	NULL	Bothell	79	98011
8	868	25111 228th St Sw	NULL	Bothell	79	98011
9	19	3148 Rose Street	NULL	Bothell	79	98011
10	16	40 Ellis St.	NULL	Bothell	79	98011

Figure 8.10: Query with Single Quotes

Numeric values are not enclosed within any quotes as shown in Code Snippet 12.

Code Snippet 12:

```
USE AdventureWorks2019
SELECT * FROM HumanResources.Department WHERE DepartmentID<10
GO
```

The query in Code Snippet 12 displays all those records where the value in DepartmentID is less than 10. The output of the query is shown in figure 8.11.

	DepartmentID	Name	GroupName	ModifiedDate
1	1	Engineering	Research and Development	2008-04-30 00:00:00.000
2	2	Tool Design	Research and Development	2008-04-30 00:00:00.000
3	3	Sales	Sales and Marketing	2008-04-30 00:00:00.000
4	4	Marketing	Sales and Marketing	2008-04-30 00:00:00.000
5	5	Purchasing	Inventory Management	2008-04-30 00:00:00.000
6	6	Research and Development	Research and Development	2008-04-30 00:00:00.000
7	7	Production	Manufacturing	2008-04-30 00:00:00.000
8	8	Production Control	Manufacturing	2008-04-30 00:00:00.000
9	9	Human Resources	Executive General and Administration	2008-04-30 00:00:00.000

Figure 8.11: Output of Where Clause with < Operator

WHERE clause can also be used with wildcard characters as shown in table 8.2. All wildcard characters are used along with LIKE keyword to make the query accurate and specific.

Wildcard	Description	Example
-	It will display a single character	SELECT * FROM Person.Contact WHERE Suffix LIKE 'Jr_'
%	It will display a string of any length	SELECT * FROM Person.Contact WHERE LastName LIKE 'B%'
[]	It will display a single character within the range enclosed in the brackets	SELECT * FROM Sales.CurrencyRate WHERE ToCurrencyCode LIKE 'C[AN][DY]'
[^]	It will display any single character not within the range enclosed in the brackets	SELECT * FROM Sales.CurrencyRate WHERE ToCurrencyCode LIKE 'A[^R][^S]'

Table 8.2: Wildcard Characters

WHERE clause also uses logical operators such as AND, OR, and NOT. These operators are used with search conditions in WHERE clauses.

AND operator joins two or more conditions and returns TRUE only when both the conditions are TRUE. Therefore, it returns all the rows from the tables where both the conditions that are listed are true.

Code Snippet 13 demonstrates AND operator.

Code Snippet 13:

```
USE AdventureWorks2019
SELECT * FROM Person.Address WHERE AddressID > 900 AND City='Seattle'
GO
```

OR operator returns TRUE and displays all the rows if it satisfies any one of the conditions.

Code Snippet 14 demonstrates OR operator.

Code Snippet 14:

```
USE AdventureWorks2019
SELECT * FROM Person.Address WHERE AddressID > 900 OR City='Seattle'
GO
```

The query in Code Snippet 14 will display all the rows whose `AddressID` is greater than 900 or whose `City` is Seattle.

The `NOT` operator negates the search condition. Code Snippet 15 demonstrates `NOT` operator.

Code Snippet 15:

```
USE AdventureWorks2019
SELECT * FROM Person.Address WHERE NOT AddressID = 5
GO
```

Code Snippet 15 will display all the records whose `AddressID` is not equal to 5. Multiple logical operators in a single `SELECT` statement can be used. When more than one logical operator is used, `NOT` is evaluated first, then `AND`, and finally `OR`.

8.3.8 GROUP BY Clause

The `GROUP BY` clause partitions the resultset into one or more subsets. Each subset has values and expressions in common. If an aggregate function is used in the `GROUP BY` clause, the resultset produces single value per aggregate.

The `GROUP BY` keyword is followed by a list of columns, known as grouped columns. Every grouped column restricts the number of rows of the resultset. For every grouped column, there is only one row.

The `GROUP BY` clause can have more than one grouped column.

Syntax:

```
SELECT <column_name1>..<column_nameN> FROM <table_name> GROUP BY <column_name>
```

where,

`column_name1`: is the name of the column according to which the resultset should be grouped.

For example, consider that if the total number of resource hours has to be found for each work order, the query in Code Snippet 16 would retrieve the resultset.

Code Snippet 16:

```
SELECT WorkOrderID, SUM(ActualResourceHrs) FROM Production.WorkOrderRouting GROUP
BY WorkOrderID
GO
```

The output is shown in figure 8.12. The output shows total number of resource hours grouped by work orders. The `SUM()` function is used to get the total and the `GROUP BY` clause is used for grouping.

	WorkOrderID	(No column name)
1	13	17.6000
2	14	17.6000
3	15	4.0000
4	16	4.0000
5	17	4.0000
6	18	4.0000
7	19	4.0000

Figure 8.12: GROUP BY Clause

The `GROUP BY` clause can be used with different clauses, such as `WHERE`, `ORDER BY`, and `HAVING`. For example, Code Snippet 17 shows the use of `HAVING` with `GROUP BY`.

Code Snippet 17:

```
USE AdventureWorks2019
SELECT WorkOrderID, SUM(ActualResourceHrs) FROM Production.WorkOrderRouting
GROUP BY WorkOrderID HAVING WorkOrderID <50
GO
```

The output will show total resource hours only for those workers that have `WorkOrderID` less than 50.

8.4 ORDER BY Clause

It specifies the order in which the columns should be sorted in a resultset. It sorts query results by one or more columns. A sort can be in either ascending (`ASC`) or descending (`DESC`) order. By default, records are sorted in an `ASC` order. To switch to the descending mode, use the optional keyword `DESC`. When multiple fields are used, SQL Server considers the leftmost field as the primary level of sort and others as lower levels of sort.

Syntax:

```
SELECT <column_name> FROM <table_name> ORDER BY <column_name> {ASC | DESC}
```

The `SELECT` statement in Code Snippet 18 sorts the query results on the `SalesLastYear` column of the `Sales.SalesTerritory` table.

Code Snippet 18:

```
SELECT * FROM Sales.SalesTerritory ORDER BY SalesLastYear
GO
```

The output is shown in figure 8.13.

	TerritoryID	Name	CountryRegionCode	Group	SalesYTD	SalesLastYear
1	8	Germany	DE	Europe	3805202.3478	1307949.7917
2	10	United Kingdom	GB	Europe	5012905.3656	1635823.3967
3	9	Australia	AU	Pacific	5977814.9154	2278548.9776
4	7	France	FR	Europe	4772398.3078	2396539.7601
5	3	Central	US	North America	3072175.118	3205014.0767
6	1	Northwest	US	North America	7887186.7882	3298694.4938
7	2	Northeast	US	North America	2402176.8476	3607148.9371
8	5	Southeast	US	North America	2538667.2515	3925071.4318
9	4	Southwest	US	North America	10510853.8739	5366575.7098
10	6	Canada	CA	North America	6771829.1376	5693988.86

Figure 8.13: ORDER BY Clause

8.5 Working with XML

Extensible Markup Language (XML) allows developers to develop their own set of tags and makes it possible for other programs to understand these tags. XML is the preferred means for developers to store, format, and manage data on the Web. Applications of today have a mix of technologies such as ASP, Microsoft .NET technologies, XML, and SQL Server 2019 working in tandem. In such a scenario, it is better to store XML data within SQL Server 2019.

Native XML databases in SQL Server 2019 have a number of advantages. Some of them are listed as follows:

Easy Data Search and Management - All the XML data is stored locally in one place, thus making it easier to search and manage.

Better Performance - Queries from a well-implemented XML database are faster than queries over documents stored in a file system. Also, the database essentially parses each document when storing it.

Easy data processing - Large documents can be processed easily.

SQL Server 2019 supports native storage of XML data by using the `xml` data type. A native XML database defines a logical model for an XML document -- as opposed to the data in that document -- and stores and retrieves documents according to that model. At a minimum, the model must include elements, attributes, PCDATA, and document order.

A native XML database has an XML document as its fundamental unit of (logical) storage, just as a relational database has a row in a table as its fundamental unit of (logical) storage.

8.5.1 XML Data Type

In addition to regular commonly used data types, SQL Server 2019 supports the `xml` data type. The `xml` data type is used to store XML documents and fragments in an SQL Server database. An XML fragment is an XML instance with the top-level element missing from its structure. Following is the syntax to create a table with columns of type `xml`:

Syntax:

```
CREATE TABLE <table_name> ( [column_list,] <column_name> xml [, column_list])
```

Code Snippet 19 creates a new table named `PhoneBilling` with `CallDetails` column belonging to `xml` data type. It is assumed that `AdventureWorks2019` is selected as the database.

Code Snippet 19:

```
CREATE TABLE Person.PhoneBilling (Bill_ID int PRIMARY KEY, MobileNumber bigint  
UNIQUE, CallDetails xml)  
GO
```

A column of type `xml` can also be added to a table at the time of creation or after its creation. The `xml` data type columns support `DEFAULT` values as well as `NOT NULL` constraint. Data can be inserted into the `xml` column in the `Person.PhoneBilling` table as shown in Code Snippet 20.

Code Snippet 20:

```
USE AdventureWorks2019  
INSERT INTO Person.PhoneBilling VALUES (100, 9833276605,  
'<Info><Call>Local</Call><Time>45 minutes</Time><Charges>200</Charges>  
</Info>')  
SELECT CallDetails FROM Person.PhoneBilling  
GO
```

The output is shown in figure 8.14. The `CallDetails` column shows the XML document that has been added as a single value. Thus, each row in the `Person.PhoneBilling` table can have an XML document in the `CallDetails` column.

Results	
	Messages
1	<Info><Call>Local</Call><Time>45 minutes</Time><...>

Figure 8.14: XML Data in Columns

The `DECLARE` statement is used to create variables of type `xml`. The purpose of the `DECLARE` statement is used to declare a variable in SQL Server.

The syntax of the DECLARE statement is as follows:

Syntax:

```
DECLARE { @LOCAL_VARIABLE data_type [ = value ] }
```

Local variable names have to start with an at (@) sign. The value argument which is indicated in the syntax is an optional parameter that helps to assign an initial value to a variable during the declaration. If you do not specify any initial value assigned to a variable, it is initialized as NULL.

Code Snippet 21 shows how to create a variable of type `xml`.

Code Snippet 21:

```
DECLARE @xmlvar xml  
SELECT @xmlvar='<Employee name="Joan" />'
```

The `xml` data type columns cannot be used as a primary key, foreign key, or as a unique constraint.

8.5.2 Typed and Untyped XML

There are two ways of storing XML documents in the `xml` data type columns, namely, typed and untyped XML. An XML instance which has a schema associated with it is called typed XML instance. A schema is a header for an XML instance or document. It describes the structure and limits the contents of XML documents by associating `xml` data types with XML element types and attributes. Associating XML schemas with the XML instances or documents is recommended because data can be validated while it is being stored into the `xml` data type column.

SQL Server does not perform any validation for data entered in the `xml` column. However, it ensures that the data stored is well-formed. Untyped XML data can be created and stored in either table columns or variables depending upon the need and scope of the data.

The first step in using typed XML is registering a schema. This is done by using the `CREATE XML SCHEMA COLLECTION` statement as shown in Code Snippet 22.

Code Snippet 22:

```
CREATE XML SCHEMA COLLECTION SoccerSchemaCollection  
AS N'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" >  
<xsd:element name="MatchDetails">  
<xsd:complexType>  
<xsd:complexContent>  
<xsd:restriction base="xsd:anyType">  
<xsd:sequence>  
<xsd:element name="Team" minOccurs="0" maxOccurs="unbounded">  
<xsd:complexType>  
<xsd:complexContent>  
<xsd:restriction base="xsd:anyType">  
<xsd:sequence />
```

```

<xsd:attribute name="country" type="xsd:string" />
<xsd:attribute name="score" type="xsd:string" />
</xsd:restriction>
</xsd:complexContent>
</xsd:complexType>
</xsd:element>
</xsd:sequence>
</xsd:restriction>
</xsd:complexContent>
</xsd:complexType>
</xsd:element>
</xsd:schema>'  

GO

```

The CREATE XML SCHEMA COLLECTION statement creates a collection of schemas, any of which can be used to validate typed XML data with the name of the collection. This example shows a new schema called **SoccerSchemaCollection** being added to the AdventureWorks2019 database. Once a schema is registered, the schema can be used in new instances of the `xml` data type.

Code Snippet 23 creates a table with an `xml` type column and specifies a schema for the column.

Code Snippet 23:

```

CREATE TABLE SoccerTeam ( TeamID int identity not null, TeamInfo xml  

(SoccerSchemaCollection) )

```

To create new rows with the typed XML data, the `INSERT` statement can be used as shown in Code Snippet 24.

Code Snippet 24:

```

INSERT INTO SoccerTeam (TeamInfo) VALUES ('<MatchDetails><Team country="Australia"  

score="3"></Team><Team country="Zimbabwe" score="2"></Team><Team  

country="England" score="4"></Team></MatchDetails>')

```

A normal `SELECT` can display the data in XML format and on clicking the data in the output, it is expanded to reveal the full XML. Figures 8.15 and 8.16 depict this.



Figure 8.15: Displaying XML Column with SELECT



The screenshot shows a SQL Server Management Studio window with two tabs: 'TeamInfo1.xml' and 'SQLQuery1.sql'. The 'TeamInfo1.xml' tab is active and displays the following XML code:

```
1 <MatchDetails>
2   <Team country="Australia" score="3" />
3   <Team country="Zimbabwe" score="2" />
4   <Team country="England" score="4" />
5 </MatchDetails>
```

Figure 8.16: Expanded XML Data in Column

A typed XML variable can also be created by specifying the schema collection name. For instance, in Code Snippet 25, a variable `team` is declared as a typed XML variable with schema name as `SoccerSchemaCollection`. The `SET` statement is used to assign the variable as an XML fragment.

Code Snippet 25:

```
DECLARE @team xml (SoccerSchemaCollection)
SET @team = '<MatchDetails><Team country="Australia"></Team></MatchDetails>'
SELECT @team
GO
```

8.6 Check Your Progress

1. Which of the following allows developers to develop their own set of tags and makes it possible for other programs to understand these tags?

(A)	XQuery	(C)	DHTML
(B)	HTML	(D)	XML

2. The _____ statement retrieves rows and columns from one or more tables.

(A)	SELECT	(C)	INSERT
(B)	DISPLAY	(D)	SHOW

3. Which of the following is the correct code to display all records from EmployeeDepartmentHistory for all employees who have DepartmentID more than 10 and whose ShiftID is 1?

(A)	USE AdventureWorks2019 SELECT * FROM [HumanResources].[EmployeeDepartm entHistory] WHERE DepartmentID > 10 with ShiftID=1	(C)	USE AdventureWorks2019 SELECT * FROM [HumanResources].[EmployeeDepart mentHistory] WHERE DepartmentID > 10 OR ShiftID=1
(B)	USE AdventureWorks2019 SELECT * FROM HumanResources].[EmployeeDepartmen tHistory] WHERE DepartmentID > 10 AND ShiftID=1	(D)	USE AdventureWorks2019 SELECT * FROM [HumanResources].[EmployeeDepart mentHistory] WHERE DepartmentID > 10

4. Which of the following clause with the SELECT statement is used to specify tables to retrieve records?

(A)	WHERE	(C)	.VALUE
(B)	FROM	(D)	.WRITE

5. _____ is used to improve the efficiency of queries on XML documents that are stored in an XML column.

(A)	XML indexing	(C)	XML querying
(B)	XML import	(D)	XML export

8.6.1 Answers

1.	D
2.	A
3.	B
4.	B
5.	A



Summary

- The SELECT statement retrieves rows and columns from tables.
- SELECT statement allows users to specify different expressions in order to view the resultset in an ordered manner.
- A SELECT statement can contain mathematical expressions by applying operators to one or more columns.
- The keyword DISTINCT prevents the retrieval of duplicate records.
- XML allows developers to develop their own set of tags and makes it possible for other programs to understand these tags.
- A typed XML instance is an XML instance which has a schema associated with it.
- XML data can be queried and retrieved using XQuery language.



Try It Yourself

1. Transcorp United Inc. is an import export company in USA. The database for the company is created in SQL Server 2019. Transcorp has around 3,000 employees worldwide. The details of the employees such as Employee Code, Employee Name, Employee Department, Date of Joining, City, and so on are stored in **Employee_Details** table.
As the database administrator of Transcorp, you have to perform the following tasks:
 - a) Retrieve data of the employees who has joined the company before the year 2019 and after 2010.
 - b) Get the data of all the employees who are from Houston.
2. Using the **Sales.Customer** table from **AdventureWorks2019** database, perform the following tasks:
 - a) Write a SELECT statement that lists only non-duplicate and non NULL Store Ids in sorted order.
 - b) Write a SELECT statement that lists all customer details whose **TerritoryID** is less than 10.
3. Using the **Sales.SalesOrderHeader** table from **AdventureWorks2019** database, perform the following task:
 - a) Write a SELECT statement that lists customer ID numbers and sales order ID numbers.
4. Write a query displaying all the columns of the **Production.ProductCostHistory** table from the rows that were modified on June 17, 2003.
5. Using the **Production.Product** table from **AdventureWorks2019** database, perform the following tasks:
 - a) Write a query that displays the product ID and name for each product from the table with the name starting with Chain.
 - b) Write a query similar to the previous one that displays the products with Lock in the name.
 - c) Change the last query so that the products without Lock in the name are displayed.
6. Write a query that displays all the rows from the **Person.Person** table where the rows were modified after December 29, 2000. Display the business entity ID number, the name columns, and the modified date.
7. Write a query displaying the ProductID, Name, and Color columns from rows in the **Production.Product** table. Display only those rows where no color has been assigned.
8. Write a query that returns the business entity ID and name columns from the **Person.Person** table.

Sort the results by **LastName**, **FirstName**, and **MiddleName**.

9. Write a query using the **Sales.SpecialOffer** table. Display the difference between the **MinQty** and **MaxQty** columns along with the **SpecialOfferID** and **Description** columns.
10. Write a query using the **Production.ProductReview** table. Use **CONTAINS** to find all the rows that have the word **socks** in the **Comments** column. Return the **ProductID** and **Comments** columns.

Session - 9

Advanced Queries and Joins

Welcome to the session, **Advanced Queries and Joins**.

This session explains various techniques to group and aggregate data and describes the concept of subqueries, table expressions, joins, and explores various set operators. The session also covers pivoting and grouping set operations.

In this Session, you will learn to:

- Explain grouping and aggregating data
- Describe subqueries
- Describe table expressions
- Explain Joins
- Describe various types of Joins
- Explain the use of various set operators to combine data
- Describe pivoting and grouping set operations

9.1 Introduction

SQL Server 2019 includes several powerful query features that help you to retrieve data efficiently and quickly. Data can be grouped and/or aggregated together in order to present summarized information. Using the concept of subqueries, a resultset of a `SELECT` can be used as criteria for another `SELECT` statement or query. Joins help you to combine column data from two or more tables based on a logical relationship between the tables. On the other hand, set operators such as `UNION` and `INTERSECT` help you to combine row data from two or more tables. The `PIVOT` and `UNPIVOT` operators are used to transform the orientation of data from column-oriented to row-oriented and vice versa. The `GROUPING SET` subclause of the `GROUP BY` clause helps to specify multiple groupings in a single query.

9.2 Grouping Data

The `GROUP BY` clause partitions the resultset into one or more subsets. Each subset has values and expressions in common. The `GROUP BY` keyword is followed by a list of columns, known as grouped columns. Every grouped column restricts the number of rows of the resultset. For every grouped column, there is only one row. The `GROUP BY` clause can have more than one grouped column.

Following is the syntax of the GROUP BY clause:

Syntax:

```
SELECT select_list FROM table_name  
GROUP BY column_name1, column_name2 ,....;
```

where,

`column_name`: is the name of the column according to which the resultset should be grouped.

Consider the `WorkOrderRouting` table in the `AdventureWorks2019` database. The total resource hours per work order must be calculated. To achieve this, the records should be grouped by work order number, that is, `WorkOrderID`.

Code Snippet 1 retrieves and displays the total resource hours per work order along with the work order number. In this query, a built-in function named `SUM()` is used to calculate the total. `SUM()` is an aggregate function.

Code Snippet 1:

```
SELECT WorkOrderID, SUM(ActualResourceHrs) AS TotalHoursPerWorkOrder FROM  
Production.WorkOrderRouting GROUP BY WorkOrderID
```

Executing this query will return all the work order numbers along with the total number of resource hours per work order.

A part of the output is shown in figure 9.1.

WorkOrderID	TotalHoursPerWorkOrder
1	13
2	14
3	15
4	16
5	17
6	18
7	19
8	20
9	21
10	22

Figure 9.1: Using GROUP BY Clause

The `GROUP BY` clause can also be used in combination with various other clauses. These clauses are as follows:

➤ GROUP BY with WHERE

The `WHERE` clause can also be used with `GROUP BY` clause to restrict the rows for grouping. The rows that satisfy the search condition are considered for grouping. The rows that do not meet the conditions in the `WHERE` clause are eliminated before any grouping is done.

Code Snippet 2 shows a query that is similar to Code Snippet 1 but limits the rows displayed, by considering only those records with **WorkOrderID** less than 50.

Code Snippet 2:

```
SELECT WorkOrderID, SUM(ActualResourceHrs) AS TotalHoursPerWorkOrder  
FROM Production.WorkOrderRouting WHERE WorkOrderID < 50 GROUP BY WorkOrderID
```

As the number of records returned is more than 25, a part of the output is shown in figure 9.2.

	WorkOrderID	TotalHoursPerWorkOrder
1	13	17.6000
2	14	17.6000
3	15	4.0000
4	16	4.0000
5	17	4.0000
6	18	4.0000
7	19	4.0000
8	20	4.0000

Figure 9.2: GROUP BY with WHERE

➤ GROUP BY with NULL

If the grouping column contains a **NULL** value, that row becomes a separate group in the resultset. If the grouping column contains more than one **NULL** value, the **NULL** values are put into a single row. Consider the **Production.Product** table. There are some rows in it that have **NULL** values in the **Class** column.

Using a **GROUP BY** on a query for this table will take into consideration the **NULL** values too. For example, Code Snippet 3 retrieves and displays the average of the list price for each **Class**.

Code Snippet 3:

```
SELECT Class, AVG(ListPrice) AS 'AverageListPrice' FROM Production.Product GROUP  
BY Class
```

As shown in figure 9.3, the **NULL** values are grouped into a single row in the output.

	Class	AverageListPrice
1	NULL	16.314
2	H	1679.4964
3	L	370.6887
4	M	635.5816

Figure 9.3: GROUP BY with NULL

➤ GROUP BY with ALL

The `ALL` keyword can also be used with the `GROUP BY` clause. It is significant only when the `SELECT` has a `WHERE` clause. When `ALL` is used, it includes all the groups that the `GROUP BY` clause produces. It even includes those groups that do not meet the search conditions.

Following is the syntax of using `GROUP BY` with `ALL`:

Syntax:

```
SELECT <column_name> FROM <table_name> WHERE <condition> GROUP BY ALL  
<column_name>
```

Consider the `Sales.SalesTerritory` table. This table has a column named `Group` indicating the geographic area to which the sales territory belongs to. Code Snippet 4 calculates and displays the total sales for each group. The output must display all the groups regardless of whether they had any sales or not. To achieve this, the code makes use of `GROUP BY` with `ALL`.

Code Snippet 4:

```
SELECT [Group], SUM(SalesYTD) AS 'TotalSales' FROM Sales.SalesTerritory WHERE  
[Group] LIKE 'N%' OR [Group] LIKE 'E%' GROUP BY ALL [Group]
```

Apart from the rows that are displayed in Code Snippet 4, it will also display the group 'Pacific' with null values as shown in figure 9.4. This is because the Pacific region did not have any sales.

	Group	TotalSales
1	Europe	13590506.0212
2	North America	33182889.0168
3	Pacific	NULL

Figure 9.4: GROUP BY with ALL

➤ GROUP BY with HAVING

`HAVING` clause is used only with `SELECT` statement to specify a search condition for a group. The `HAVING` clause acts as a `WHERE` clause in places where the `WHERE` clause cannot be used against aggregate functions such as `SUM()`. Once you have created groups with a `GROUP BY` clause, you may wish to filter the results further. The `HAVING` clause acts as a filter on groups, similar to how the `WHERE` clause acts as a filter on rows returned by the `FROM` clause. Following is the syntax of `GROUP BY` with `HAVING`:

Syntax:

```
SELECT <column_name> FROM <table_name> GROUP BY <column_name> HAVING  
<search_condition>
```

Code Snippet 5 displays the row with the group 'Pacific' as it has total sales less than 6000000.

Code Snippet 5:

```
SELECT [Group], SUM(SalesYTD) AS 'TotalSales' FROM Sales.SalesTerritory WHERE  
[Group] LIKE 'P%' GROUP BY ALL [Group] HAVING SUM(SalesYTD) < 6000000
```

The output of this is only 1 row, with Group name Pacific and total sales, 5977814.9154.

9.3 Summarizing Data

The `GROUP BY` clause also uses operators such as `CUBE` and `ROLLUP` to return summarized data. The number of columns in the `GROUP BY` clause determines the number of summary rows in the resultset. The operators are described as follows:

➤ **CUBE:**

`CUBE` is an aggregate operator that produces a super-aggregate row. In addition to the usual rows provided by the `GROUP BY`, it also provides the summary of rows that the `GROUP BY` clause generates.

The summary row is displayed for every possible combination of groups in the resultset. The summary row displays `NULL` in the resultset, but at the same time returns all the values for those.

Following is the syntax of `CUBE`:

Syntax:

```
SELECT <column_name> FROM <table_name> GROUP BY <column_name> WITH CUBE
```

Code Snippet 6 demonstrates the use of `CUBE`.

Code Snippet 6:

```
SELECT Name, CountryRegionCode, SUM(SalesYTD) AS TotalSales FROM Sales.  
SalesTerritory WHERE Name <> 'Australia' AND Name <> 'Canada' GROUP BY  
Name, CountryRegionCode WITH CUBE
```

`CUBE` is like an extension of the `GROUP BY` clause. `CUBE` allows you to generate subtotals for all combinations of grouping columns specified in the `GROUP BY` clause.

Code Snippet 6 retrieves and displays the total sales of each country (except Australia and Canada) and also, the total of the sales of all the countries' regions.

The output will return more than 20 rows. A part of the output is shown in figure 9.5.

	Name	CountryRegionCode	TotalSales
1	Germany	DE	3805202.3478
2	NULL	DE	3805202.3478
3	France	FR	4772398.3078
4	NULL	FR	4772398.3078
5	United Kingdom	GB	5012905.3656
6	NULL	GB	5012905.3656
7	Central	US	3072175.118
8	Northeast	US	2402176.8476
9	Northwest	US	7887186.7882
10	Southeast	US	2538667.2515

Figure 9.5: Using GROUP BY with CUBE

➤ **ROLLUP:**

In addition to the usual rows that are generated by the `GROUP BY`, it also introduces summary rows into the resultset. It is similar to `CUBE` operator, but generates a resultset that shows groups arranged in a hierarchical order. It arranges the groups from the lowest to the highest. Group hierarchy in the result is dependent on the order in which the columns that are grouped are specified.

When generating grouping sets, `ROLLUP` assumes a hierarchy among the dimension columns and only generates grouping sets based on this hierarchy. `ROLLUP` is often used to generate subtotals and totals for reporting purposes. `ROLLUP` is commonly used to calculate the aggregates of hierarchical data such as sales by year > quarter > month.

Following is the syntax of `ROLLUP`:

Syntax:

```
SELECT <column_name> FROM <table_name> GROUP BY <column_name> WITH ROLLUP
```

Code Snippet 7 demonstrates the use of `ROLLUP`.

Code Snippet 7:

```
SELECT Name, CountryRegionCode, SUM(SalesYTD) AS TotalSales FROM
Sales.SalesTerritory
WHERE Name <> 'Australia' AND Name <> 'Canada' GROUP BY
Name, CountryRegionCode
WITH ROLLUP
```

The output is shown in figure 9.6. As seen in the output, the query retrieves and displays total sales of each country (except Australia and Canada), total of the sales of these countries' regions, and then, arranges them in order from lowest to highest.

	Name	CountryRegionCode	TotalSales
1	Central	US	3072175.118
2	Central	NULL	3072175.118
3	France	FR	4772398.3078
4	France	NULL	4772398.3078
5	Germany	DE	3805202.3478
6	Germany	NULL	3805202.3478
7	Northeast	US	2402176.8476
8	Northeast	NULL	2402176.8476
9	Northwest	US	7887186.7882
10	Northwest	NULL	7887186.7882
11	Southeast	US	2538667.2515
12	Southeast	NULL	2538667.2515
13	Southwest	US	10510853.8739
14	Southwest	NULL	10510853.8739
15	United Kingdom	GB	5012905.3656
16	United Kingdom	NULL	5012905.3656
17	NULL	NULL	40001565.9004

Figure 9.6: Using GROUP BY with ROLLUP

9.4 Aggregate Functions

Occasionally, developers may also require to perform analysis across rows, such as counting rows, meeting specific criteria, or summarizing total sales for all orders. Aggregate functions enable to accomplish it.

Since aggregate functions return a single value, they can be used in `SELECT` statements where a single expression is used, such as `SELECT`, `HAVING`, and `ORDER BY` clauses. Aggregate functions ignore `NULLS`, except when using `COUNT(*)`.

Aggregate functions in a `SELECT` list do not generate a column alias. You may wish to use the `AS` clause to provide one.

Aggregate functions in a `SELECT` clause operate on all rows passed to the `SELECT` phase. If there is no `GROUP BY` clause, all rows will be summarized.

SQL Server provides many built-in aggregate functions. Commonly used functions are included in table 9.1.

Function Name	Syntax	Description
AVG	<code>AVG (<expression>)</code>	Calculates the average of all the non- <code>NULL</code> numeric values in a column.

Function Name	Syntax	Description
COUNT or COUNT_BIG	COUNT(*) or COUNT(<expression>)	When (*) is used, this function counts all rows, including those with NULL. The function returns count of non-NULL rows for the column when a column is specified as <expression>. The return value of COUNT function is an int. The return value of COUNT_BIG is a big_int.
MAX	MAX(<expression>)	Returns the largest number, latest date/time, or last occurring string.
MIN	MIN(<expression>)	Returns the smallest number, earliest date/time, or first occurring string.
SUM	SUM(<expression>)	Calculates the sum of all the non-NULL numeric values in a column.

Table 9.1: Commonly Used Aggregate Functions

To use a built-in aggregate in a `SELECT` clause, consider the query in Code Snippet 8.

Code Snippet 8:

```
SELECT AVG([UnitPrice]) AS AvgUnitPrice, MIN([OrderQty]) AS MinQty,
MAX([UnitPriceDiscount]) AS MaxDiscount
FROM Sales.SalesOrderDetail;
```

Since the query does not use a `GROUP BY` clause, all rows in the table will be summarized by the aggregate formulas in the `SELECT` clause. The output is shown in figure 9.7.

	Results	Messages
	AvgUnitPrice	MinQty
1	465.0934	1

	Results	Messages
	MaxDiscount	
1	0.40	

Figure 9.7: Using Aggregate Functions

When using aggregates in a `SELECT` clause, all columns referenced in the `SELECT` list must be used as inputs for an aggregate function or must be referenced in a `GROUP BY` clause. Failing this, there will be an error. For example, the query in Code Snippet 9 will return an error.

Code Snippet 9:

```
SELECT SalesOrderID, AVG(UnitPrice) AS AvgPrice FROM Sales.SalesOrderDetail;
```

This returns an error stating that the column `Sales.SalesOrderDetail.SalesOrderID` is invalid in the `SELECT` list because it is not contained in either an aggregate function or the `GROUP BY` clause. As the query is not using a `GROUP BY` clause, all rows will be treated as a single group. All columns, therefore, must be used as inputs to aggregate functions. To correct or prevent the error, one must remove `SalesOrderID` from the query.

Besides using numeric data, aggregate expressions can also include date, time, and character data for

summarizing.

Code Snippet 10 returns the earliest and latest order date, using MIN and MAX.

Code Snippet 10:

```
SELECT MIN(OrderDate) AS Earliest, MAX(OrderDate) AS Latest FROM Sales.SalesOrderHeader;
```

Figure 9.8 shows the output.

Results		Messages
	Earliest	Latest
1	2011-05-31 00:00:00.000	2014-06-30 00:00:00.000

Figure 9.8: Using Aggregate Functions with Non-Numeric Data

Other functions may also be used in combination with aggregate functions.

9.5 Spatial Aggregates

SQL Server provides several methods that help to aggregate two individual items of geometry or geography data. These methods are listed in table 9.2.

Method	Description
STUnion	Returns an object that represents the union of a geometry/geography instance with another geometry/geography instance.
STIntersection	Returns an object that represents the points where a geometry/geography instance intersects another geometry/geography instance.
STConvexHull	Returns an object representing the convex hull of a geometry/geography instance. A set of points is called convex if for any two points, the entire segment is contained in the set. The convex hull of a set of points is the smallest convex set containing the set. For any given set of points, there is only one convex hull.

Table 9.2: Spatial Aggregate Methods

Figures 9.9, 9.10, and 9.11 visually depict an example of these methods. Such output can be seen by clicking the 'spatial results' tab. When SSMS sees any spatial data in the results, it adds a tab to visualize it.

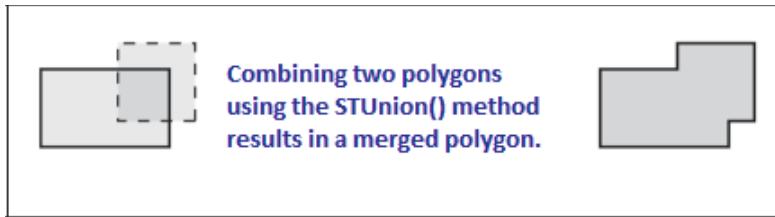


Figure 9.9: STUnion()



Figure 9.10: STIntersection()

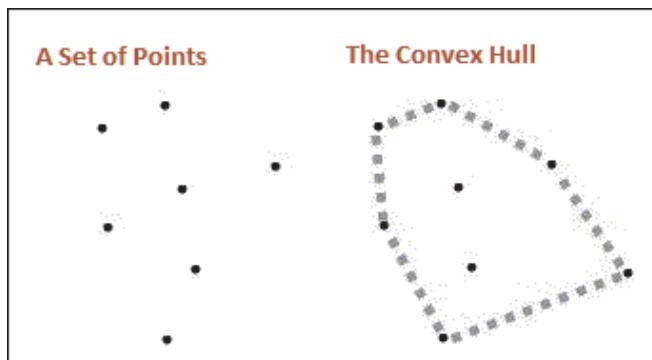


Figure 9.11: STConvexHull()

Code Snippet 11 demonstrates the use of STUnion().

Code Snippet 11:

```
SELECT geometry::Point(251, 1, 4326) .STUnion(geometry::Point(252, 2, 4326));
```

The output is shown in figure 9.12. It shows two points.

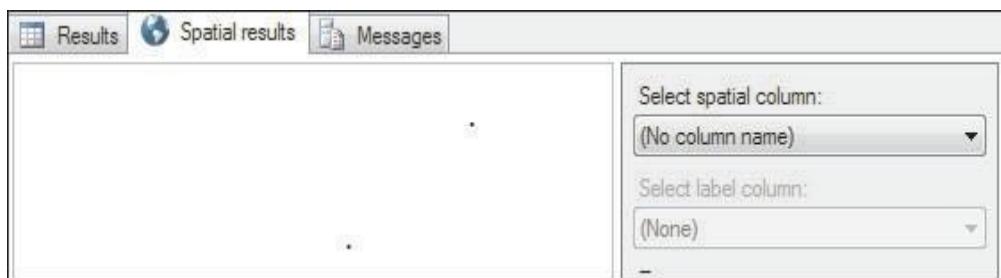


Figure 9.12: Using STUnion() with a geometry Type

Another example is given in Code Snippet 12.

Code Snippet 12:

```
DECLARE @City1 geography  
SET @City1=geography::STPolyFromText(  
    'POLYGON((175.3-41.5, 178.3-37.9, 172.8-34.6, 175.3-41.5))', 4326)  
DECLARE @City2 geography  
SET @City2=geography::STPolyFromText(  
    'POLYGON((169.3-46.6, 174.3-41.6, 172.5-40.7, 166.3-45.8, 169.3-46.6))',  
    4326)  
DECLARE @CombinedCity geography=@City1.STUnion(@City2)  
SELECT @CombinedCity
```

Here, two variables are declared of the `geography` type and appropriate values are assigned to them. Then, they are combined into a third variable of `geography` type by using the `STUnion()` method.

The output of the code is shown in figure 9.13.

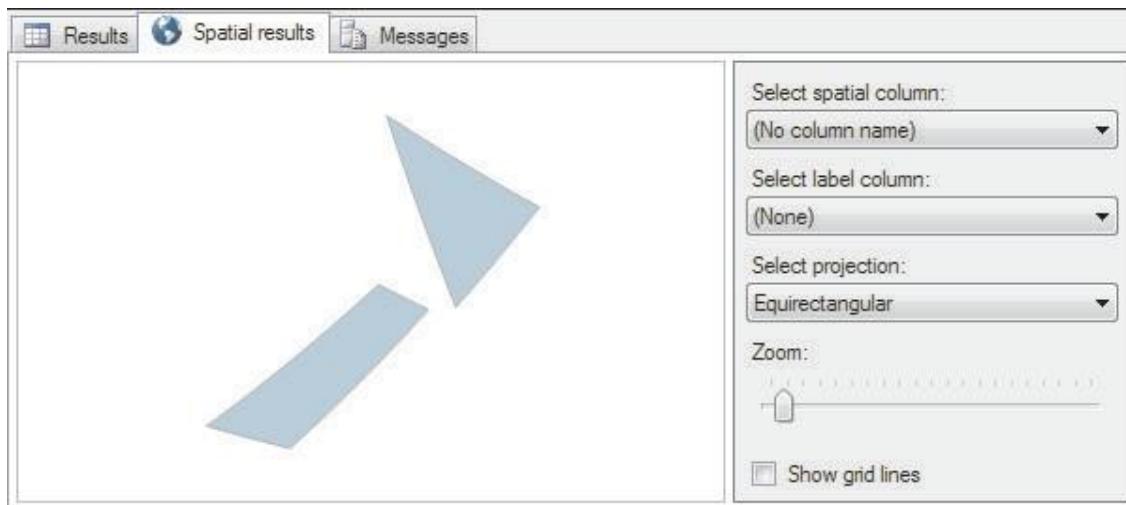
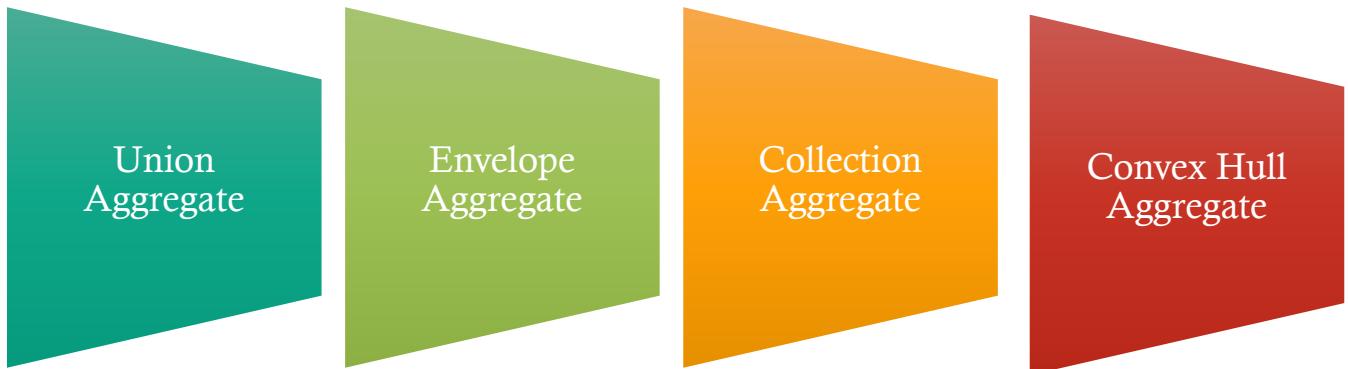


Figure 9.13: Using `STUnion()` with a `geography` Type

9.5.1 More Spatial Aggregates

Consider that you require to compute union of a set of geometry/geography objects or all elements in a spatial column or determine average of a set of Point elements. It would not be possible to use the `AVG()` function here. In such a case, you will make use of advanced spatial aggregate functions in SQL Server.

Following four additional aggregates were first introduced in SQL Server 2012 to the suite of spatial operators in SQL Server and are applicable to SQL Server 2019 as well:



These aggregates are implemented as static methods, which work for either `geography` or `geometry` data types. Although, aggregates are applicable to all classes of spatial data, they can be best described with polygons.

➤ Union Aggregate

It performs a union operation on a set of `geometry` objects. It combines multiple spatial objects into a single spatial object, removing interior boundaries, where applicable. Following is the syntax of `UnionAggregate`:

Syntax:

```
UnionAggregate (geometry_operand or geography_operand)
```

where,

`geometry_operand`: is a `geometry` type table column comprising the set of `geometry` objects on which a union operation will be performed.

`geography_operand`: is a `geography` type table column comprising the set of `geography` objects on which a union operation will be performed.

Code Snippet 13 demonstrates a simple example of using the `UnionAggregate`. It uses the `Person.Address` table in the `AdventureWorks2019` database.

Code Snippet 13:

```
SELECT Geography::UnionAggregate(SpatialLocation) AS AVGLocation  
FROM Person.Address  
WHERE City = 'London';
```

The output of this will be as shown in figure 9.14.

	AVGLocation
1	0xE6100000104A0010003DA82EC605C249407D37109CAD...

Figure 9.14: Using Spatial Aggregates

To view a visual representation of the spatial data, you can click **Spatial results** tab in the output window. This will display the output as shown in figure 9.15.

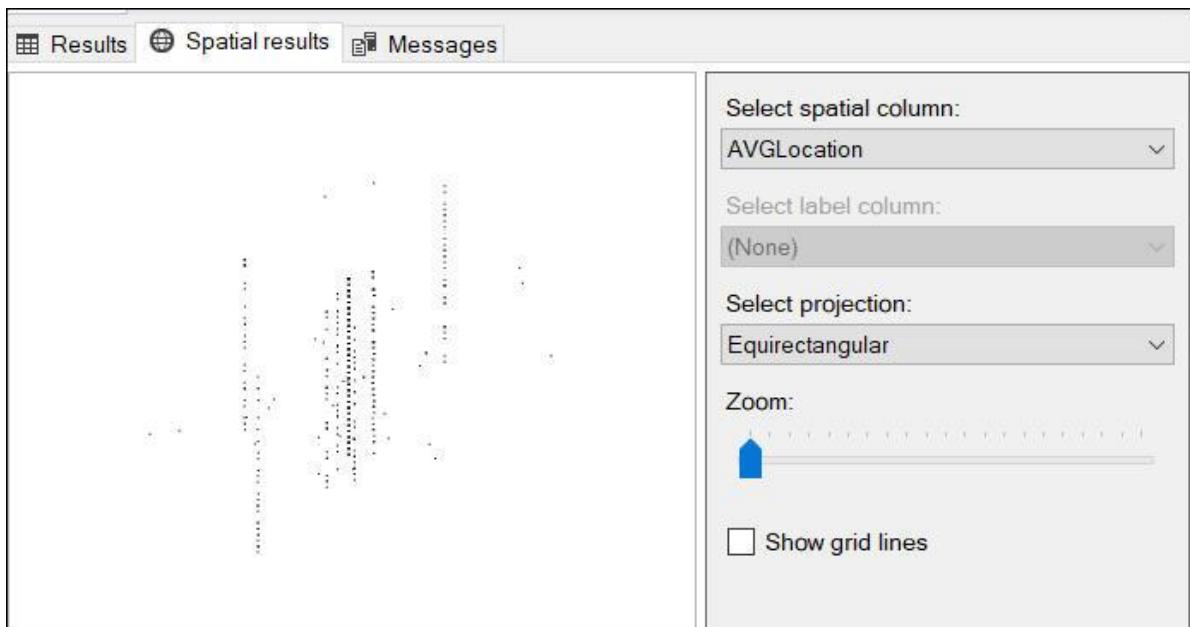


Figure 9.15: Viewing Spatial Results

➤ Envelope Aggregate

The Envelope Aggregate returns a bounding area for a given set of geometry or geography objects. It exhibits different behaviors for geography and geometry types. Based on the type of object it is applied to, it returns different results. For the geometry type, the result is a 'traditional' rectangular polygon, which closely bounds the selected input objects. For the geography type, the result is a circular object, which loosely bounds the selected input objects. Furthermore, the circular object is defined using the new CurvePolygon feature.

Following is the syntax of EnvelopeAggregate:

Syntax:

```
EnvelopeAggregate (geometry operand or geography operand)
```

where,

geometry_operand: is a geometry type table column comprising the set of geometry objects.

geography_operand: is a geography type table column comprising the set of geography objects.

Code Snippet 14 returns a bounding box for a set of objects in a table variable column.

Code Snippet 14:

```
SELECT Geography::EnvelopeAggregate(SpatialLocation) AS Location  
FROM Person.Address WHERE City = 'London'
```

The visual representation of the output is shown in figure 9.16.

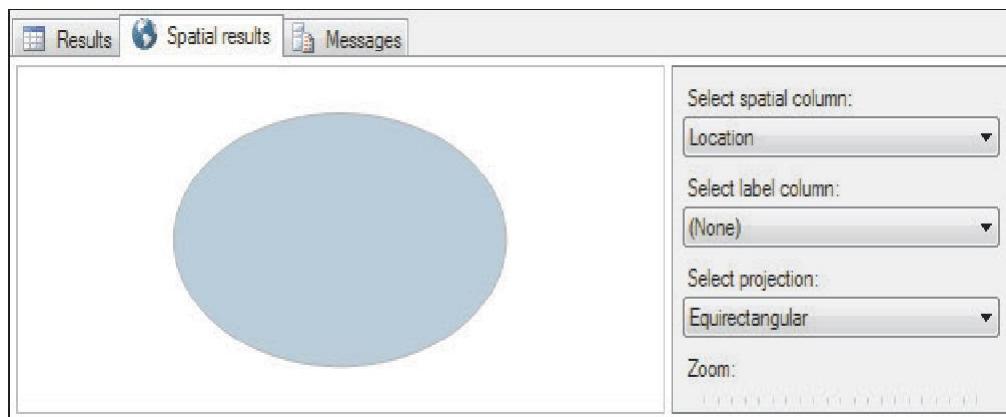


Figure 9.16: EnvelopeAggregate

➤ Collection Aggregate

It returns a GeometryCollection/GeographyCollection instance with one geometry/geography part for each spatial object(s) in the selection set. Following is the syntax of CollectionAggregate:

Syntax:

```
CollectionAggregate (geometry_operand or geography_operand)
```

where,

geometry_operand: is a geometry type table column comprising the set of geometry objects.
geography_operand: is a geography type table column comprising the set of geography objects.

Code Snippet 15 returns a GeometryCollection instance that contains a CurvePolygon and a Polygon.

Code Snippet 15:

```
DECLARE @CollectionDemo TABLE (
shape geometry, shapeType nvarchar(50)
)
INSERT INTO @CollectionDemo(shape, shapeType) VALUES ('CURVEPOLYGON(
CIRCULARSTRING(2 3, 4 1, 6 3, 4 5, 2 3))', 'Circle'),
('POLYGON((1 1, 4 1, 4 5, 1 5, 1 1))', 'Rectangle');

SELECT geometry::CollectionAggregate(shape) FROM @CollectionDemo;
```

The output of the code will be as shown in figure 9.17.

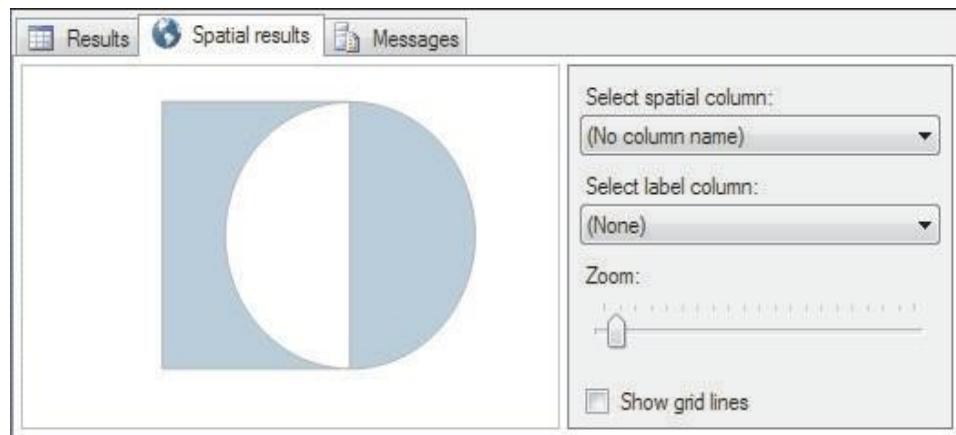


Figure 9.17: Using CollectionAggregate

➤ Convex Hull Aggregate

It returns a convex hull polygon, which encloses one or more spatial objects for a given set of geometry/geography objects. Following is the syntax of ConvexHullAggregate:

Syntax:

```
ConvexHullAggregate (geometry_operand or geography_operand)
```

where,

geometry_operand: is a geometry type table column comprising the set of geometry objects.
geography_operand: is a geography type table column comprising the set of geography objects.

Code Snippet 16 demonstrates the use of ConvexHullAggregate.

Code Snippet 16:

```
SELECT Geography::ConvexHullAggregate(SpatialLocation) AS Location
FROM Person.Address
WHERE City = 'London'
```

The output is shown in figure 9.18.



Figure 9.18: Using ConvexHullAggregate

9.6 Subqueries

You can use a `SELECT` statement or a query to return records that will be used as criteria for another `SELECT` statement or query. The outer query is called parent query and the inner query is called a subquery. The purpose of a subquery is to return results to the outer query. In other words, the inner query statement should return the column or columns used in the criteria of the outer query statement.

The simplest form of a subquery is one that returns just one column. The parent query can use the results of this subquery using an `=` sign. The syntax for the most basic form of a subquery using just one column with an `=` sign is as shown:

Syntax:

```
SELECT <ColumnName> FROM <table>
WHERE <ColumnName> = (SELECT <ColumnName> FROM <Table> WHERE <ColumnName> =
<Condition> )
```

In a subquery, the innermost `SELECT` statement is executed first and its result is passed as criteria to the outer `SELECT` statement.

Consider a scenario where it is required to determine the due date and ship date of the most recent orders.

Code Snippet 17 shows the code to achieve this.

Code Snippet 17:

```
SELECT DueDate, ShipDate FROM Sales.SalesOrderHeader WHERE
Sales.SalesOrderHeader.OrderDate= (SELECT MAX(OrderDate)
FROM Sales.SalesOrderHeader)
```

Here, a subquery has been used to achieve the desired output. The inner query or subquery retrieves the most recent order date. This is then passed to the outer query, which displays due date and ship date for all the orders that were made on that particular date.

A part of the output of the code is shown in figure 9.19.

	DueDate	ShipDate
1	2014-07-12 00:00:00.000	2014-07-07 00:00:00.000
2	2014-07-12 00:00:00.000	2014-07-07 00:00:00.000
3	2014-07-12 00:00:00.000	2014-07-07 00:00:00.000
4	2014-07-12 00:00:00.000	2014-07-07 00:00:00.000
5	2014-07-12 00:00:00.000	2014-07-07 00:00:00.000
6	2014-07-12 00:00:00.000	2014-07-07 00:00:00.000
7	2014-07-12 00:00:00.000	2014-07-07 00:00:00.000
8	2014-07-12 00:00:00.000	2014-07-07 00:00:00.000
9	2014-07-12 00:00:00.000	2014-07-07 00:00:00.000
10	2014-07-12 00:00:00.000	2014-07-07 00:00:00.000

Figure 9.19: Using a Simple Subquery

Based on the results returned by the inner query, a subquery can be classified as a scalar subquery or a multi-valued subquery.

These are described as follows:

- Scalar subqueries return a single value. Here, the outer query must be written to process a single result.
- Multi-valued subqueries return a result similar to a single-column table. Here, the outer query must be written to handle multiple possible results.

9.6.1 Working with Multi-valued Queries

If an = operator is used with the subquery, the subquery must return a single scalar value. If more than one value is returned, there will be an error and the query will not be processed. In such scenarios, the keywords ANY, ALL, IN, and EXISTS can be used with the WHERE clause of a SELECT statement when the query returns one column but one or more rows.

These keywords, also called predicates, are used with multi-valued queries. For example, consider that all the first names and last names of employees whose job title is 'Research and Development Manager' have to be displayed. Here, the inner query may return more than one row, as there may be more than one employee with that job title. To ensure that the outer query can use the results of the inner query, the IN keyword will have to be used.

Code Snippet 18 demonstrates this.

Code Snippet 18:

```
SELECT FirstName, LastName FROM Person.Person  
WHERE Person.Person.BusinessEntityID IN (SELECT BusinessEntityID  
FROM HumanResources.Employee WHERE JobTitle='Research and Development Manager');
```

Here, the inner query retrieves the **BusinessEntityID** from the **HumanResources.Employee** table for those records having job title 'Research and Development Manager'. These results are then passed to the outer query, which matches the **BusinessEntityID** with that in the **Person.Person** table. Finally, from the records that are matching, the first and last names are extracted and displayed.

The output is displayed in figure 9.20.

	FirstName	LastName
1	Dylan	Miller
2	Michael	Raheem

Figure 9.20: Output of Subquery with IN keyword

The SOME or ANY keywords evaluate to true if the result is an inner query containing at least one row that satisfies the comparison. They compare a scalar value with a column of values. SOME and ANY are equivalent; both return the same result. They are rarely used.

There are some guidelines to be followed when working with subqueries. You should remember the following points when using subqueries:

The ntext, text, and image data types cannot be used in the SELECT list of subqueries.

The SELECT list of a subquery introduced with a comparison operator can have only one expression or column name.

Subqueries that are introduced by a comparison operator not followed by the keyword ANY or ALL cannot include GROUP BY and HAVING clauses.

You cannot use DISTINCT keyword with subqueries that include GROUP BY.

You can specify ORDER BY only when TOP is also specified.

Besides scalar and multi-valued subqueries, you can also choose between self-contained subqueries and correlated subqueries. These are defined as follows:

- Self-contained subqueries are written as standalone queries, without any dependencies on the outer query. A self-contained subquery is processed once when the outer query runs and passes its results to the outer query.
- Correlated subqueries reference one or more columns from the outer query and therefore, depend on the outer query. Correlated subqueries cannot be run separately from the outer query.

The EXISTS keyword is used with a subquery to check the existence of rows returned by the subquery. The subquery does not actually return any data; it returns a value of TRUE or FALSE.

Following is the syntax of a subquery containing the word EXISTS:

Syntax:

```
SELECT <ColumnName> FROM <table> WHERE [NOT] EXISTS  
(  
    <Subquery_Statement>  
)
```

where,

Subquery_Statement: specifies the subquery.

The code in Code Snippet 18 can be rewritten as shown in Code Snippet 19 using EXISTS keyword to yield same output.

Code Snippet 19:

```
SELECT FirstName, LastName FROM Person.Person AS A WHERE EXISTS (SELECT *
FROM HumanResources.Employee AS B WHERE JobTitle = 'Research and Development Manager'
AND A.BusinessEntityID=B.BusinessEntityID);
```

Here, the inner subquery retrieves all those records that match job title as 'Research and Development Manager' and whose **BusinessEntityID** matches with that in the **Person** table. If there are no records matching both these conditions, the inner subquery will not return any rows. Thus, in that case, the **EXISTS** will return false and the outer query will also not return any rows. However, the code in Code Snippet 19 will return two rows because the given conditions are satisfied. The output will be the same as figure 9.20.

Similarly, one can use the **NOT EXISTS** keyword. The **WHERE** clause in which it is used is satisfied if there are no rows returned by the subquery.

9.6.2 Nested Subqueries

A subquery that is defined inside another subquery is called a nested subquery. Consider that you wanted to retrieve and display the names of persons from Canada. There is no direct way to retrieve this information since the **Sales.SalesTerritory** table is not related to **Person.Person** table. Hence, a nested subquery is used here as shown in Code Snippet 20.

Code Snippet 20:

```
SELECT LastName, FirstName FROM Person.Person WHERE BusinessEntityID IN (SELECT
BusinessEntityID FROM Sales.SalesPerson WHERE TerritoryID IN
(SELECT TerritoryID
FROM Sales.SalesTerritory
WHERE Name='Canada'))
)
```

The output is shown in figure 9.21.

Results		Messages	
	Last Name	First Name	
1	Vargas	Garrett	
2	Saraiva	José	

Figure 9.21: Output of Nested Subqueries

9.6.3 Correlated Queries

In many queries containing subqueries, the subquery must be evaluated only once to provide the values required by the parent query. This is because in most of the queries, the subquery makes no reference to the parent query, so the value in the subquery remains constant.

However, if the subquery refers to a parent query, the subquery must be reevaluated for every iteration in the parent query. This is because the search criterion in the subquery is dependent upon the value of a

particular record in the parent query.

When a subquery takes parameters from its parent query, it is known as Correlated subquery. Consider that you want to retrieve all the business entity ids of persons whose contact information was last modified not earlier than 2019. To do this, you can use a correlated subquery as shown in Code Snippet 21.

Code Snippet 21:

```
SELECT e.BusinessEntityID FROM Person.BusinessEntityContact e WHERE  
e.ContactTypeID IN (  
    SELECT c.ContactTypeID FROM  
    Person.ContactType c  
    WHERE YEAR(e.ModifiedDate) >= 2012  
)
```

In Code Snippet 21, the inner query retrieves contact type ids for all those persons whose contact information was modified on or before 2019. These results are then passed to the outer query, which matches these contact type ids with those in the **Person.BusinessEntityContact** table and displays the business entity IDs of those records. Figure 9.22 shows part of the output.

	BusinessEntityID
1	292
2	294
3	296
4	298
5	300
6	302
7	304

Figure 9.22: Output of Correlated Queries

9.7 Joins

Joins are used to retrieve data from two or more tables based on a logical relationship between tables. A join typically specifies foreign key relationship between the tables. It defines the manner in which two tables are related in a query by:

- Specifying the column from each table to be used for the join. A typical join specifies a foreign key from one table and its associated key in the other table.
- Specifying a logical operator such as =, <> to be used in comparing values from the columns.

Joins can be specified in either the **FROM** or **WHERE** clauses.

Following is the syntax of the JOIN statement:

Syntax:

```
SELECT <ColumnName1>, <ColumnName2>...<ColumnNameN> FROM  
Table_A AS Table_Alias_A  
JOIN  
Table_B AS Table_Alias_B ON  
Table_Alias_A.<CommonColumn> = Table_Alias_B.<CommonColumn>
```

where,

- <ColumnName1>, <ColumnName2>: Is a list of columns that must be displayed.
- Table_A: Is the name of the table on the left of the JOIN keyword.
- Table_B: Is the name of the table on the right of the JOIN keyword.
- AS Table_Alias: Is a way of giving an alias name to the table. An alias defined for the table in a query can be used to denote a table so that the full name of the table need not be used.
- <CommonColumn>: Is a column that is common to both the tables. In this case, the join succeeds only if the columns have matching values.

Consider that you want to list employee first names, last names, and their job titles from the HumanResources.Employee and Person.Person. To extract this information from the two tables, you must join them based on BusinessEntityID as shown in Code Snippet 22.

Code Snippet 22:

```
SELECT A.FirstName, A.LastName, B.JobTitle  
FROM Person.Person A  
JOIN  
HumanResources.Employee B ON  
A.BusinessEntityID = B.BusinessEntityID;
```

Here, the tables **HumanResources.Employee** and **Person.Person** are given aliases A and B. They are joined together based on their business entity ids. The SELECT statement then retrieves the desired columns through the aliases.

Figure 9.23 shows the output.

	FirstName	LastName	Job Title
1	Ken	Sánchez	Chief Executive Officer
2	Temi	Duffy	Vice President of Engineering
3	Roberto	Tamburello	Engineering Manager
4	Rob	Walters	Senior Tool Designer
5	Gail	Erickson	Design Engineer
6	Jossef	Goldberg	Design Engineer
7	Dylan	Miller	Research and Development Manager

Figure 9.23: Output of Join

There are three types of joins as follows:

Inner Joins

Outer Joins

Self-Joins

9.7.1 Inner Join

An inner join is formed when records from two tables are combined only if the rows from both the tables are matched based on a common column.

Following is the syntax of an inner join:

Syntax:

```
SELECT <ColumnName1>, <ColumnName2>...<ColumnNameN> FROM Table_A  
AS Table_Alias_A  
INNER JOIN  
Table_B AS Table_Alias_B ON  
Table_Alias_A.<CommonColumn>=Table_Alias_B.<CommonColumn>
```

Code Snippet 23 demonstrates the use of inner join. The scenario for this is similar to Code Snippet 22.

Code Snippet 23:

```
SELECT A.FirstName, A.LastName, B.JobTitle  
FROM Person.PersonA  
INNER JOIN HumanResources.EmployeeB  
ON  
A.BusinessEntityID = B.BusinessEntityID;
```

In Code Snippet 23, an inner join is constructed between `Person.Person` and `HumanResources.Employee` based on common business entity ids. Here again, the two tables are given aliases of A and B respectively. The output is the same as shown in figure 9.23.

9.7.2 Outer Join

Outer joins are join statements that return all rows from at least one of the tables specified in the `FROM` clause, as long as those rows meet any `WHERE` or `HAVING` conditions of the `SELECT` statement. The two types of commonly used outer joins are as follows:

- Left Outer Join
- Right Outer Join

Left Outer Join

Left outer join returns all the records from the left table and only matching records from the right table.

Following is the syntax of an outer join:

Syntax:

```
SELECT <ColumnList> FROM Table_A  
AS Table_Alias_A  
LEFT OUTER JOIN  
Table_B AS Table_Alias_B ON  
Table_Alias_A.<CommonColumn> = Table_Alias_B.<CommonColumn>
```

Consider that you want to retrieve all the customer ids from the **Sales.Customers** table and order information such as ship dates and due dates, even if the customers have not placed any orders. Since the record count would be very huge, it is to be restricted to only those orders that are placed before 2019. To achieve this, you perform a left outer join as shown in Code Snippet 24.

Code Snippet 24:

```
SELECT A.CustomerID, B.DueDate, B.ShipDate FROM Sales.Customer A LEFT OUTER JOIN  
Sales.SalesOrderHeader B  
ON  
A.CustomerID = B.CustomerID AND YEAR(B.DueDate) < 2019;
```

In Code Snippet 24, the left outer join is constructed between the tables **Sales.Customer** and **Sales.SalesOrderHeader**. The tables are joined based on customer ids. In this case, all records from the left table, **Sales.Customer** and only matching records from the right table, **Sales.SalesOrderHeader**, are returned. Figure 9.24 shows the output.

	CustomerID	DueDate	ShipDate
31...	16402	2014-07-12 00:00:00.000	2014-07-07 00:00:00.000
31...	18178	2014-07-12 00:00:00.000	2014-07-07 00:00:00.000
31...	13671	2014-07-12 00:00:00.000	2014-07-07 00:00:00.000
31...	11981	2014-07-12 00:00:00.000	2014-07-07 00:00:00.000
31...	18749	2014-07-12 00:00:00.000	2014-07-07 00:00:00.000
31...	15251	2014-07-12 00:00:00.000	2014-07-07 00:00:00.000
31...	15868	2014-07-12 00:00:00.000	2014-07-07 00:00:00.000
31...	18759	2014-07-12 00:00:00.000	2014-07-07 00:00:00.000
31...	215	NULL	NULL
31...	46	NULL	NULL
31...	169	NULL	NULL
31...	507	NULL	NULL
31...	630	NULL	NULL
31...	338	NULL	NULL

Figure 9.24: Output of Left Outer Join

As shown in the output, some records show the due dates and ship dates as NULL. This is because for some customers, no order is placed, hence, their records will show the dates as NULL.

Right Outer Join

The right outer join retrieves all records from the second table in the join, regardless of whether there is matching data in the first table or not.

Following is the syntax of a right outer join:

Syntax:

```
SELECT <ColumnList> FROM  
Left_Table_Name AS  
Table_A AS Table_Alias_A  
RIGHT OUTER JOIN  
Table_B AS Table_Alias_B ON  
Table_Alias_A.<CommonColumn>=Table_Alias_B.<CommonColumn>
```

Consider that you want to retrieve all the product names from **Product** table and all the corresponding sales order ids from the **SalesOrderDetail** table even if there is no matching record for the products in the **SalesOrderDetail** table. To do this, you will use a right outer join as shown in Code Snippet 25.

Code Snippet 25:

```
SELECT P.Name, S.SalesOrderID FROM Sales.SalesOrderDetails  
RIGHT OUTER JOIN  
Production.Product P  
ON P.ProductID=S.ProductID;
```

In the code, all the records from **Product** table are shown regardless of whether they have been sold or not.

9.7.3 Self-Join

A self-join is used to find records in a table that are related to other records in the same table. A table is joined to itself in a self-join.

Code Snippet 26 demonstrates how to use a self-join to retrieve the product details of those products that have the same color from the table **Production.Product**.

Code Snippet 26:

```
SELECT  
    p1. ProductID,  
    p1.Color,  
    p1.Name,  
    p2.Name
```

```

FROM
    Production.Product p1
INNER JOIN Production.Product p2 ON p1.Color = p2.Color
ORDER BY p1.ProductID

```

The output is shown in figure 9.25.

	ProductID	Color	Name	Name
1	317	Black	LL Crankarm	LL Crankarm
2	317	Black	LL Crankarm	ML Crankarm
3	317	Black	LL Crankarm	HL Crankarm
4	317	Black	LL Crankarm	Chainring
5	317	Black	LL Crankarm	HL Road Frame - Black, 58
6	317	Black	LL Crankarm	Sport-100 Helmet, Black
7	317	Black	LL Crankarm	Road-750 Black, 44
8	317	Black	LL Crankarm	Road-750 Black, 48
9	317	Black	LL Crankarm	Road-750 Black, 52
10	317	Black	LL Crankarm	Road-750 Black, 58

Figure 9.25: Self Join Example

9.7.4 MERGE Statement

The `MERGE` statement allows you to maintain a target table based on certain join conditions on a source table using a single statement. You can now perform the following actions in one `MERGE` statement:

Insert a new row from the source if the row is missing in the target table

Update a target row if a record already exists in the source table

Delete a target row if the row is missing in the source table

Consider that you want to:

- Compare last and first names of customers from both source and target tables
- Update customer information in target table if the last and first names match
- Insert new records in target table if the last and first names in source table do not exist in target table
- Delete existing records in target table if the last and first names do not match with those of source table

The `MERGE` statement accomplishes the tasks in a single statement. `MERGE` also allows you to display those records that were inserted, updated, or deleted by using an `OUTPUT` clause.

Syntax:

```

MERGE target_table USING source_table ON match_condition
WHEN MATCHED THEN UPDATE SET Col1=val1 [, Col2=val2...]
WHEN [TARGET] NOT MATCHED THEN INSERT (Col1 [, Col2...]) VALUES (Val1 [, , Val2...])
WHEN NOT MATCHED BY SOURCE THEN DELETE

```

```
[OUTPUT $action, Inserted.Col1, Deleted.Col1, ...] ;
```

where,

target_table: is the table WHERE changes are being made.

source_table: is the table from which rows will be inserted, updated, or deleted into the target table.

match_conditions: are the JOIN conditions and any other comparison operators.

MATCHED: true if a row in the target_table and source_table matches the match_condition.

NOT MATCHED: true if a row from the source_table does not exist in the target_table.

SOURCE NOT MATCHED: true if a row exists in the target_table but not in the source_table.

OUTPUT: An optional clause that allows to view those records that have been inserted/deleted/updated in target_table.

MERGE statements are terminated with a semi-colon (:).

Code Snippet 27 shows how to use MERGE statement.

Code Snippet 27:

```
SET IDENTITY_INSERT [Person].[AddressType] ON
MERGE INTO [Person].[AddressType] AS Target
USING (VALUES
    (1,'Billing')
    ,(2,'Home')
    ,(3,'Headquarters')
    ,(4,'Primary')
    ,(5,'Shipping')
    ,(6,'Archival')
    ,(7,'Contact')
    ,(8,'Alternative'))
) AS Source
    ([AddressTypeID], [Name]) ON (Target.AddressTypeID = Source.AddressTypeID)
WHEN MATCHED AND (Target.[Name] <> Source.[Name]) THEN
    UPDATE SET [Name] = Source.[Name]
WHEN NOT MATCHED BY TARGET THEN
    INSERT([AddressTypeID], [Name]) VALUES(Source.AddressTypeID, Source.[Name])
WHEN NOT MATCHED BY SOURCE THEN
    DELETE
OUTPUT $action, Inserted.AddressTypeID, Inserted.Name,
Deleted.AddressTypeID, Deleted.Name;
```

The **Person.AddressType** table is the target table and hard-coded data represents the source. The match condition is the column, **AddressTypeID** of both source and target. If the match condition evaluates to false (NOT MATCHED), then new records are inserted in the target table. If match condition evaluates to true (MATCHED), then records are updated into the target table from the source table.

If records present in the target table do not match with those of source table (NOT MATCHED BY SOURCE), then these are deleted from the target table. The last statement displays a report consisting of rows that were inserted/updated/deleted as shown in the output.

Figure 9.26 shows the output. Observe that one record has been updated, whereas one has been inserted.

	Action	AddressTypeID	Name	AddressTypeID	Name
1	UPDATE	3	Headquarters	3	Head Office
2	INSERT	8	Alternative	NULL	NULL

Figure 9.26: Using MERGE

9.8 Common Table Expressions (CTEs)

A CTE is similar to a temporary resultset defined within the execution scope of a single SELECT, INSERT, UPDATE, DELETE, or CREATE VIEW statement. A CTE is a named expression defined in a query. A CTE is defined at the start of a query and can be referenced several times in the outer query.

A CTE that include references to itself is called as a recursive CTE.

Note - CTEs were first introduced in SQL Server 2005.

Key advantages of CTEs are improved readability and ease in maintenance of complex queries.

Syntax:

```
WITH <CTE_name>
AS (<CTE_definition>)
```

For example, to retrieve and display the customer count year-wise for orders present in the **Sales.SalesOrderHeader** table, the code will be as given in Code Snippet 28.

Code Snippet 28:

```
WITH CTE_OrderYear AS
(
    SELECT YEAR(OrderDate) AS OrderYear, CustomerID
    FROM Sales.SalesOrderHeader
)
SELECT OrderYear, COUNT(DISTINCT CustomerID) AS CustomerCount FROM
CTE_OrderYear
GROUP BY OrderYear;
```

Here, **CTE_OrderYear** is specified as the CTE name. The **WITH...AS** keywords begins the CTE definition. Then, the CTE is used in the **SELECT** statement to retrieve and display the desired results.

Figure 9.27 shows the output.

	OrderYear	CustomerCount
1	2013	11095
2	2014	10354
3	2011	1406
4	2012	3162

Figure 9.27: Output of CTE

Following guidelines should be remembered while defining CTEs:

CTEs are limited in scope to the execution of the outer query. Hence, when the outer query ends, the lifetime of the CTE will end.

You must define a name for a CTE and also, define unique names for each of the columns referenced in the SELECT clause of the CTE.

It is possible to use inline or external aliases for columns in CTEs.

A single CTE can be referenced multiple times in the same query with one definition.

Multiple CTEs can also be defined in the same WITH clause. For example, consider Code Snippet 29. It defines two CTEs using a single WITH clause.

This snippet assumes that three tables named **Student**, **City**, and **Status** are created in the **AdventureWorks2019** database.

Code Snippet 29:

```
WITH CTE_Students
AS (
StudentCode, S.Name,C.CityName, St.Status FROM Student S
INNER JOIN City C
      ON S.CityCode = C.CityCode INNER
JOIN Status St
ON S.StatusId = St.StatusId)
,
StatusRecord -- This is the second CTE being defined
AS (
SELECT Status, COUNT(Name) AS CountofStudents FROM
CTE_Students
GROUP BY      Status
)
SELECT * FROM StatusRecord
```

Assuming some records are inserted in all three tables, the output may be as shown in figure 9.28.

	Status	CountofStudents
1	Failed	2
2	Passed	2

Figure 9.28: Using Multiple CTE with Single WITH

9.9 Combining Data Using SET Operators

SQL Server 2019 provides certain keywords, also called as operators, to combine data from multiple tables. These operators are as follows:

- UNION
- INTERSECT
- EXCEPT

9.9.1 UNION Operator

Results from two different query statements can be combined into a single resultset using the UNION operator. The query statements must have compatible column types and equal number of columns. The column names can be different in each statement, but the data types must be compatible. By compatible data types, it means that it should be possible to convert the contents of one of the columns into another. For example, if one of the query statements has an `int` data type and the other query statement has a `money` data type, they are compatible and a union can take place between them because the `int` data can be converted into `money` data.

Following is the syntax of the UNION operator.

Syntax:

```
Query_Statement1 UNION [ALL] Query_Statement2
```

where,

`Query_Statement1` and `Query_Statement2` are `SELECT` statements.

Code Snippet 30 demonstrates the UNION operator.

Code Snippet 30:

```
SELECT Product.ProductId FROM Production.Product UNION  
SELECT ProductId FROM Sales.SalesOrderDetail
```

This will list all the product ids of both tables that match with each other. If you include the `ALL` clause, all rows are included in the resultset including duplicate records.

Code Snippet 31 demonstrates the UNION ALL operator.

Code Snippet 31:

```
SELECT Product.ProductId FROM Production.Product UNION ALL  
SELECT ProductId FROM Sales.SalesOrderDetail
```

By default, the `UNION` operator removes duplicate records from the resultset. However, if you use the `ALL` clause with `UNION` operator, then all the rows are returned. Apart from `UNION`, the other operators that are used to combine data from multiple tables are `INTERSECT` and `EXCEPT`.

9.9.2 *INTERSECT Operator*

Consider again the two tables `Product` and `SalesOrderDetail` present in `AdventureWorks2019`. Suppose you want to display only the rows that are common to both the tables. To do this, you will use an operator named `INTERSECT`. The `INTERSECT` operator is used with two query statements to return a distinct set of rows that are common to both the query statements.

Syntax:

```
Query_statement1  
INTERSECT  
Query_statement2
```

where,

`Query_Statement1` and `Query_Statement2` are `SELECT` statements.

Code Snippet 32 demonstrates the `INTERSECT` operator.

Code Snippet 32:

```
SELECT Product.ProductId FROM Production.Product  
INTERSECT  
SELECT ProductId FROM Sales.SalesOrderDetail
```

The basic rules for using `INTERSECT` are as follows:

- Number of columns and order in which they are given must be same in both queries.
- Data types of the columns being used must be compatible.

The result of the intersection of the `Production.Product` and `Sales.SalesOrderDetail` tables would be only those product ids that have matching records in `Production.Product` table. In large enterprises, there are huge volumes of data stored in databases. Instead of storing the entire data in a single table, it can be spread over several tables that are related to one another. When data is stored in such tables, there must be some means to combine and retrieve the data from those tables. Using SQL Server, there are a number of ways to combine data from multiple tables.

9.9.3 *EXCEPT Operator*

The `EXCEPT` operator returns all of the distinct rows from the query given on left of the `EXCEPT` operator and removes all rows from the resultset that match rows on right of the `EXCEPT` operator.

Syntax:

```
Query_statement1  
EXCEPT  
Query statement2
```

where,

Query_Statement1 and Query_Statement2 are SELECT statements.

The two rules that apply to `INTERSECT` operator are also applicable for `EXCEPT` operator. Code Snippet 33 demonstrates the `EXCEPT` operator.

Code Snippet 33:

```
SELECT Product.ProductId FROM Production.Product  
EXCEPT  
SELECT ProductId FROM Sales.SalesOrderDetail
```

If the order of the two tables in this example is interchanged, only those rows are returned from `Production.Product` table which do not match with the rows present in `Sales.SalesOrderDetail`.

Thus, in simple terms, `EXCEPT` operator selects all the records from the first table except those that match with the second table. Hence, when you are using `EXCEPT` operator, the order of the two tables in the queries is important. Whereas, with the `INTERSECT` operator, it does not matter which table is specified first.

9.10 Pivoting and Grouping Set Operations

Consider a scenario where data must be displayed in a different orientation than it is stored in, in terms of row and column layout. The process of transforming data from a row-based orientation to a column-based orientation is called pivoting. The `PIVOT` and `UNPIVOT` operators of SQL Server help to change the orientation of data from column-oriented to row-oriented and vice versa. This is accomplished by consolidating values present in a column to a list of distinct values and then, projecting that list in the form of column headings.

9.10.1 PIVOT Operator

Following is the brief syntax for `PIVOT`:

Syntax:

```
SELECT <non-pivoted column>,  
[firstpivoted column] AS <column name>,  
[secondpivoted column] AS <column name>,  
...
```

```

[last pivoted column] AS <column name> FROM
(<SELECT query that produces the data>) AS
<alias for the source query>
PIVOT (
    <aggregation function>(<column being aggregated>) FOR
    <column that contains the values that will become column headers>]
IN ( [first pivoted column], [second pivoted column],
... [last pivoted column])
) AS <alias for the pivot table>
<optional ORDER BY clause>;

```

where,

`table_source`: is a table or table expression.

`aggregate_function`: is a user-defined or in-built aggregate function that accepts one or more inputs.

`value_column`: is the value column of the `PIVOT` operator.

`pivot_column`: is the pivot column of the `PIVOT` operator. This column must be of a type that can implicitly or explicitly be converted to `nvarchar()`.

`IN (column_list)`: are values in the `pivot_column` that will become the column names of the output table. The list must not include any column names that already exist in the input `table_source` being pivoted.

`table_alias`: is the alias name of the output table.

The output of this will be a table containing all columns of the `table_source` except the `pivot_column` and `value_column`. These columns of the `table_source`, excluding the `pivot_column` and `value_column`, are called the grouping columns of the pivot operator.

In simpler terms, to use the `PIVOT` operator, you supply three elements to the operator:

Grouping	Spreading	Aggregation
In the <code>FROM</code> clause, the input columns must be provided. The <code>PIVOT</code> operator uses those columns to determine which column(s) to use for grouping the data for aggregation.	Here, a comma-separated list of values that occur in the source data is provided that will be used as the column headings for the pivoted data.	An aggregation function, such as <code>SUM</code> , to be performed on the grouped rows.

Consider an example to understand the `PIVOT` operator. Code Snippet 34 is shown without the `PIVOT` operator and demonstrates a simple `GROUP BY` aggregation. As the number of records would be huge, the resultset is limited to five by specifying `TOP 5`.

Code Snippet 34:

```

SELECT TOP 5 SUM(SalesYTD) AS TotalSalesYTD, Name FROM Sales.SalesTerritory
GROUP BY Name

```

Figure 9.29 shows the output.

	TotalSalesYTD	Name
1	7887186.7882	Northwest
2	2402176.8476	Northeast
3	3072175.118	Central
4	10510853.8739	Southwest
5	2538667.2515	Southeast

Figure 9.29: Grouping without PIVOT

The top five year to date sales along with territory names grouped by territory names are displayed. Now, the same query is rewritten in Code Snippet 35 using a `PIVOT` so that the data is transformed from a row-based orientation to a column-based orientation.

Code Snippet 35:

```
-- Pivot table with one row and six columns
SELECT TOP 5 'TotalSalesYTD' AS GrandTotal,
[Northwest], [Northeast], [Central], [Southwest], [Southeast] FROM
(SELECT TOP 5 Name, SalesYTD FROM Sales.SalesTerritory) AS SourceTable PIVOT
(SUM(SalesYTD) FOR Name IN ([Northwest], [Northeast], [Central], [Southwest],
[Southeast])) AS PivotTable;
```

Figure 9.30 shows the output.

	GrandTotal	Northwest	Northeast	Central	Southwest	Southeast
1	TotalSalesYTD	7887186.7882	2402176.8476	3072175.118	10510853.8739	2538667.2515

Figure 9.30: Grouping with PIVOT

As shown in figure 9.30, the data is transformed and the territory names are now seen as columns instead of rows. This improves readability. A major challenge in writing queries using `PIVOT` is the requirement to provide a fixed list of spreading elements to the `PIVOT` operator, such as specific territory names given in Code Snippet 35. It would not be feasible or practical to implement this for large number of spreading elements. To overcome this, developers can use dynamic SQL. Dynamic SQL provides a means to build a character string that is passed to SQL Server, interpreted as a command, and then, executed.

9.10.2 UNPIVOT Operator

`UNPIVOT` performs almost the reverse operation of `PIVOT`, by rotating columns into rows. Unpivoting does not restore the original data. Detail-level data was lost during the aggregation process in the original pivot. `UNPIVOT` has no ability to allocate values to return to the original detail values. Instead of turning rows into columns, unpivoting results in columns being transformed into rows. SQL Server provides the `UNPIVOT` table operator to return a row-oriented tabular display from a pivoted data.

When unpivoting data, one or more columns are defined as the source to be converted into rows. The data in those columns is spread, or split, into one or more new rows, depending on how many columns are being unpivoted.

To use the UNPIVOT operator, you provide three elements as follows:

Source columns to be unpivoted

A name for the new column that will display the unpivoted values

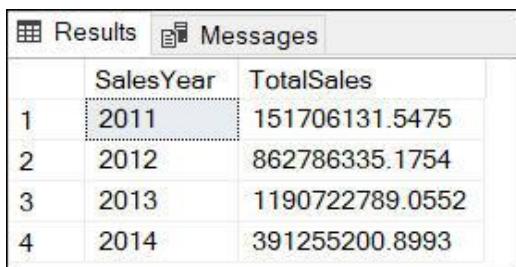
A name for the column that will display the names of the unpivoted values

Code Snippet 36 demonstrates the UNPIVOT operator.

Code Snippet 36:

```
SELECT SalesYear, TotalSales FROM
(
    SELECT * FROM
    (
        SELECT YEAR(SOH.OrderDate) AS SalesYear,
               SOH.SubTotal AS TotalSales
        FROM sales.SalesOrderHeader SOH
        JOIN sales.SalesOrderDetail SOD ON SOH.SalesOrderId =
SOD.SalesOrderId
    ) AS Sales PIVOT(SUM(TotalSales) FOR SalesYear IN([2011],
[2012],
[2013],
[2014])) AS PVT
) T UNPIVOT(TotalSales FOR SalesYear IN([2011],
[2012],
[2013],
[2014])) AS upvt;
```

The output is shown in figure 9.31.



	SalesYear	TotalSales
1	2011	151706131.5475
2	2012	862786335.1754
3	2013	1190722789.0552
4	2014	391255200.8993

Figure 9.31: UNPIVOT

9.11 Check Your Progress

1. Which of the following statements can be used with subqueries that return one column and many rows?

(A)	ANY	(C)	IN
(B)	ALL	(D)	=

2. The _____ operator is used to display only the rows that are common to both the tables.

(A)	INTERSECT	(C)	UNION
(B)	EXCEPT	(D)	UNION WITH ALL

3. A subquery that includes another subquery within it is called a ____.

(A)	Join	(C)	Correlated Subquery
(B)	Nested Subquery	(D)	Parent Subquery

4. _____ is formed when records from two tables are combined only if the rows from both the tables are matched based on a common column.

(A)	Inner join	(C)	Self-join
(B)	Left Outer join	(D)	Right Outer join

5. _____ return all rows from at least one of the tables in the `FROM` clause of the `SELECT` statement, as long as those rows meet any `WHERE` or `HAVING` conditions of the `SELECT` statement.

(A)	Inner join	(C)	Self-join
(B)	Outer join	(D)	Sub queries

6. Consider you have two tables, `Products` and `Orders` that are already populated. Based on new orders, you want to update `Quantity` in `Products` table. You write the following code:

```
MERGE Products AS T
USING Orders AS S
ON S.ProductID = T.ProductID
```

Which of the following code, when inserted into the blank space, will allow you to achieve this?

(A)	WHEN MATCHED THEN UPDATE SET T.Quantity = S.Quantity	(C)	WHEN NOT MATCHED THEN UPDATE SET T.Quantity = S.Quantity
(B)	WHEN MATCHED THEN UPDATE SET Quantity= Quantity	(D)	WHEN MATCHED THEN UPDATE SET S.Quantity = T.Quantity

9.11.1 Answers

1.	C
2.	A
3.	B
4.	A
5.	B
6.	A



Summary

- The GROUP BY clause and aggregate functions enable to group and/or aggregate data together in order to present summarized information.
- Spatial aggregate functions were first introduced in SQL Server 2012 and are supported in SQL Server 2019 as well.
- A subquery allows the resultset of one SELECT statement to be used as criteria for another SELECT statement.
- Joins help you to combine column data from two or more tables based on a logical relationship between the tables.
- Set operators such as UNION and INTERSECT help you to combine row data from two or more tables.
- The PIVOT and UNPIVOT operators help to change the orientation of data from column-oriented to row-oriented and vice versa.
- The GROUPING SET subclause of the GROUP BY clause helps to specify multiple groupings in a single query.



Try It Yourself

1. Write a query to display the employee names and their departments from the AdventureWorks2019 database.
2. Using the tables Sales.SalesPerson and Sales.SalesTerritory, retrieve IDs of all the sales persons who operate in Canada.
3. Using the tables Sales.SalesPerson and Sales.SalesTerritory, retrieve IDs of all the sales persons who operate in Northwest or Northeast.
4. Compare the bonus values of salespersons in the Sales.SalesPerson table to find out the sales persons earning more bonuses. Display the SalesPersonID and bonus values in descending order. (Hint: Use a self-join and ORDER BY ...DESC).
5. Retrieve all the values of SalesPersonID from Sales.SalesPerson table, but leave out those values, which are present in the Sales.Store table. (Hint: Use EXCEPT operator).
6. Combine all the SalesPersonIDs of the tables Sales.SalesPerson and Sales.Store.
7. Retrieve all the sales person IDs and territory IDs from Sales.SalesPerson table regardless of whether they have matching records in the Sales.SalesTerritory table. (Hint: Use a left outer join).
8. Retrieve a distinct set of Territory IDs that are present in both Sales.SalesPerson and Sales.SalesTerritory tables. (Hint: Use INTERSECT operator).

Session – 10

Views, Stored Procedures, and Querying Metadata

Welcome to the session, **Views, Stored Procedures, and Querying Metadata**.

This session explains about views and describes creating, altering, and dropping views. The session also describes stored procedures in detail. The session concludes with an explanation of the techniques to query metadata.

In this session, you will learn to:

- Define views
- Describe the technique to create, alter, and drop views
- Define stored procedures and its types
- Describe the procedure to create, alter, and execute stored procedures
- Describe nested stored procedures
- Describe querying SQL Server metadata System Catalog views and functions

10.1 Introduction

An SQL Server database has two main categories of objects: those that store data and those that access, manipulate, or provide access to data. Views and stored procedures belong to the latter category.

10.2 Views

A view is a virtual table that is made up of selected columns from one or more tables. The tables from which the view is created are referred to as base tables. These base tables can be from different databases. A view can also include columns from other views created in the same or a different database. A view can have a maximum of 1,024 columns. The data inside the view comes from the base tables that are referenced in the view definition. The rows and columns of views are created dynamically when the view is referenced.

Note - All the Code Snippets in the session are based on the AdventureWorks2019 database.

10.2.1 Creating Views

A user can create a view using columns from tables or other views only if the user has permission to access these tables and views. A view is created using the `CREATE VIEW` statement and it can be created only in the current database. SQL Server verifies the existence of objects that are referenced in the view definition.

Note - While creating a view, test the `SELECT` statement that defines the view to make sure that SQL Server returns the expected result. You create the view only after the `SELECT` statement is tested and the resultset has been verified.

Following syntax is used to create a view:

Syntax:

```
CREATE VIEW <view_name>
AS <select_statement>
```

where,

`view_name`: specifies the name of the view.

`select_statement`: specifies the `SELECT` statement that defines the view.

Code Snippet 1 creates a view from the `Production.Product` table to display only the product id, product number, name, and safety stock level of products.

Code Snippet 1:

```
CREATE VIEW vwProductInfo AS
SELECT ProductID, ProductNumber, Name, SafetyStockLevel
FROM Production.Product;
GO
```

Note - The words `vw` are prefixed to a view name as per recommended coding conventions.

Following code in Code Snippet 2 is used to display details of `vwProductInfo` view:

Code Snippet 2:

```
SELECT * FROM vwProductInfo
```

The result will show specified columns of all products from `Production.Product` table. A part of the output is shown in figure 10.1.

ProductID	ProductNumber	Name	SafetyStockLevel
1	AR-5381	Adjustable Race	1000
2	BA-8327	Bearing Ball	1000
3	BE-2349	BB Ball Bearing	800
4	BE-2908	Headset Ball Bearings	800
5	BL-2036	Blade	800
6	CA-5965	LL Crankarm	500
7	CA-6738	ML Crankarm	500
8	CA-7457	HL Crankarm	500
9	CB-2903	Chainring Bolts	1000
10	CN-6137	Chainring Nut	1000

Figure 10.1: Records from a View

10.2.2 Creating Views Using JOIN Keyword

The `JOIN` keyword can also be used to create views. The `CREATE VIEW` statement is used along with the `JOIN` keyword to create a view using columns from multiple tables.

Following syntax is used to create a view with the `JOIN` keyword:

Syntax:

```
CREATE VIEW<view_name> AS  
SELECT * FROM table_name1 JOIN  
table_name2  
ON table_name1.column_name = table_name2.column_name
```

where,

`view_name`: specifies the name of the view.

`table_name1`: specifies the name of first table.

`JOIN`: specifies that two tables are joined using `JOIN` keyword.

`table_name2`: specifies the name of the second table.

Code Snippet 3 creates a view named `vwPersonDetails` with specific columns from the `Person` and `Employee` tables of `HumanResources` schema. The `JOIN` and `ON` keywords join the two tables based on `BusinessEntityID` column.

Code Snippet 3:

```
CREATE VIEW vwPersonDetails AS  
SELECT  
    p.Title  
, p.[FirstName]  
, p.[MiddleName]  
, p.[LastName]  
, e.[JobTitle]  
FROM [HumanResources].[Employee] e  
    INNER JOIN [Person].[Person] p  
        ON p.[BusinessEntityID] = e.[BusinessEntityID]  
GO
```

This view will contain the columns `Title`, `FirstName`, `MiddleName`, and `LastName` from the `Person` table and `JobTitle` from the `Employee` table. Once the view is created, you can retrieve records from it, and manipulate and modify records as well.

Code Snippet 4 shows executing a query on this view.

Code Snippet 4:

```
SELECT * FROM vwPersonDetails
```

The output will be as shown in figure 10.2.

	Title	FirstName	MiddleName	LastName	Job Title
1	NULL	Ken	J	Sánchez	Chief Executive Officer
2	NULL	Temi	Lee	Duffy	Vice President of Engineering
3	NULL	Roberto	NULL	Tamburello	Engineering Manager
4	NULL	Rob	NULL	Walters	Senior Tool Designer
5	Ms.	Gail	A	Erickson	Design Engineer
6	Mr.	Jossef	H	Goldberg	Design Engineer
7	NULL	Dylan	A	Miller	Research and Development Manager
8	NULL	Diane	L	Margheim	Research and Development Engineer
9	NULL	Gigi	N	Matthew	Research and Development Engineer
10	NULL	Michael	NULL	Raheem	Research and Development Manager

Figure 10.2: Creating Views with a Join

As shown in figure 10.2, all the rows may not have values for the **Title** or **MiddleName** columns - some may have **NULL** in them. A person seeing this output may not be able to comprehend the meaning of the **NULL** values. Hence, to replace all the **NULL** values in the output with a null string, the **COALESCE()** function can be used as shown in Code Snippet 5.

Code Snippet 5:

```
CREATE VIEW vwPersonDetailsNew
AS
SELECT
    COALESCE(p.Title, '') AS Title
    ,p.[FirstName]
    ,COALESCE(p.MiddleName, '') AS MiddleName
    ,p.[LastName]
    ,e.[JobTitle]
FROM [HumanResources].[Employee] e
    INNER JOIN [Person].[Person] p
        ON p.BusinessEntityID = e.BusinessEntityID
GO
```

When this view is queried with a **SELECT** statement, the output will be as shown in figure 10.3.

	Title	FirstName	MiddleName	LastName	Job Title
1		Ken	J	Sánchez	Chief Executive Officer
2		Temi	Lee	Duffy	Vice President of Engineering
3		Roberto		Tamburello	Engineering Manager
4		Rob		Walters	Senior Tool Designer
5	Ms.	Gail	A	Erickson	Design Engineer
6	Mr.	Jossef	H	Goldberg	Design Engineer
7		Dylan	A	Miller	Research and Development Manager
8		Diane	L	Margheim	Research and Development Engineer
9		Gigi	N	Matthew	Research and Development Engineer
10		Michael		Raheem	Research and Development Manager

Figure 10.3: Using COALESCE Function in a View

10.2.3 Guidelines and Restrictions on Views

A view can be created using the `CREATE VIEW` command. Before creating a view, following guidelines and restrictions should be considered:

A view is created only in the current database. The base tables and views from which the view is created can be from other databases or servers.

View names must be unique and cannot be the same as the table names in the schema.

A view cannot be created on temporary tables.

A view cannot have a full-text index.

A view cannot contain the `DEFAULT` definition.

The `CREATE VIEW` statement can include the `ORDER BY` clause only if the `TOP` keyword is used.

Views cannot reference more than 1,024 columns.

The `CREATE VIEW` statement cannot include the `INTO` keyword.

The `CREATE VIEW` statement cannot be combined with other Transact-SQL statements in a single batch.

Code Snippet 6 reuses the code given in Code Snippet 5 with an `ORDER BY` clause.

Code Snippet 6:

```
CREATE VIEW vwSortedPersonDetails AS
SELECT TOP 10 COALESCE(p.Title, ' ') AS Title
,p.[FirstName]
,COALESCE(p.MiddleName, ' ') AS MiddleName
,p.[LastName]
,e.[JobTitle]
FROM [HumanResources].[Employee] e INNER JOIN [Person].[Person] p
ON p.[BusinessEntityID] = e.[BusinessEntityID] ORDER BY p.FirstName
GO
--Retrieve records from the view
SELECT * FROM vwSortedPersonDetails
```

The `TOP` keyword displays the name of the first ten employees with their first names in ascending order.

10.3 Modifying Data through Views

Views can be used to modify data in database tables. Data can be inserted, modified, or deleted through a view using following statements:

- INSERT
- UPDATE
- DELETE

10.3.1 INSERT with Views

The `INSERT` statement is used to add a new row to a table or a view. During the execution of the statement, if the value for a column is not provided, the SQL Server Database Engine must provide a value based on the definition of the column. If the Database Engine is unable to provide this value, then the new row will not be added.

The value for the column is provided automatically if the column:

Has an IDENTITY property

Has a default value specified

Has a timestamp data type

Takes null values

Is a computed column

While using the `INSERT` statement on a view, if any rules are violated, the record is not inserted.

In the following example, when data is inserted through the view, the insertion does not take place as the view is created from two base tables.

First, create a table `Employee_Personal_Details` as shown in Code Snippet 7.

Code Snippet 7:

```
CREATE TABLE Employee_Personal_Details (
    EmpID int NOT NULL,
    FirstName varchar(30) NOT NULL,
    LastName varchar(30) NOT NULL, Address
    varchar(30)
)
```

Then, create a table `Employee_Salary_Details` as shown in Code Snippet 8.

Code Snippet 8:

```
CREATE TABLE Employee_Salary_Details (
    EmpID
    int NOT NULL,
    Designation varchar(30),
    Salary int NOT NULL
)
```

Code Snippet 9 creates a view `vwEmployee_Details` using columns from the `Employee_Personal_Details` and `Employee_Salary_Details` tables by joining the two tables on the `EmpID` column.

Code Snippet 9:

```
CREATE VIEW vwEmployee_Personal_Details
AS
SELECT e1.EmpID, FirstName, LastName, Designation, Salary
FROM Employee_Personal_Details e1
JOIN Employee_Salary_Details e2
ON e1.EmpID = e2.EmpID
```

Code Snippet 10 uses the `INSERT` statement to insert data through the view `vwEmployee_Details`.

However, the data is not inserted as the view is created from two base tables.

Code Snippet 10:

```
INSERT INTO vwEmployee_Personal_Details VALUES (2, 'Jack', 'Wilson', 'Software
Developer', 16000)
```

Following error message is displayed when the `INSERT` statement is executed.

```
'Msg 4405, Level 16, State 1, Line 1
View      or function 'vwEmployee_Details' is not updatable because the
modification affects multiple base tables.'
```

Values can be inserted into user-defined data type columns by:

- Specifying a value of the user-defined type.
- Calling a user-defined function that returns a value of the user-defined type.

Following rules and guidelines must be followed when using the `INSERT` statement:

- The `INSERT` statement must specify values for all columns in a view in the underlying table that do not allow null values and have no `DEFAULT` definitions.
- When there is a self-join with the same view or base table, the `INSERT` statement does not work.

Code Snippet 11 creates a view `vwEmpDetails` using `Employee_Personal_Details` table. The `Employee_Personal_Details` table contains a column named `LastName` that does not allow null values to be inserted.

Code Snippet 11:

```
CREATE VIEW vwEmpDetails AS  
SELECT FirstName, Address  
FROM Employee_Personal_Details
```

Code Snippet 12 attempts to insert values into the **vwEmpDetails** view.

Code Snippet 12:

```
INSERT INTO vwEmpDetails VALUES ('Jack', 'NYC')
```

This insert is not allowed as the view does not contain the **Last Name** column from the base table and that column does not allow null values.

10.3.2 UPDATE with Views

The **UPDATE** statement can be used to change the data in a view. Updating a view also updates the underlying table.

Consider an example. Code Snippet 13 creates a table named **Product_Details**.

Code Snippet 13:

```
CREATE TABLE Product_Details (  
    ProductID int, ProductName  
    varchar(30), Rate money  
)
```

Assume some records are added in the table as shown in figure 10.4.

	ProductID	ProductName	Rate
1	5	DVD Writer	2250.00
2	4	DVD Writer	1250.00
3	6	DVD Writer	1250.00
4	2	External Hard Drive	4250.00
5	3	External Hard Drive	4250.00

Figure 10.4: Records in the Product_Details

Code Snippet 14 creates a view based on the **Product_Details** table.

Code Snippet 14:

```
CREATE VIEW vwProduct_Details  
AS  
SELECT ProductName, Rate FROM Product_Details
```

Code Snippet 15 updates the view to change all rates of DVD writers to 3,000.

Code Snippet 15:

```
UPDATE vwProduct_Details  
SET Rate=3000  
WHERE ProductName='DVD Writer'
```

The outcome of this code affects not only the view, **vwProduct_Details**, but also the underlying table from which the view was created.

Figure 10.5 shows the updated table which was automatically updated because of the view.

	ProductID	ProductName	Rate
1	5	DVD Writer	3000.00
2	4	DVD Writer	3000.00
3	6	DVD Writer	3000.00
4	2	External Hard Drive	4250.00
5	3	External Hard Drive	4250.00

Figure 10.5: Updated Table

Large value data types include `varchar(max)`, `nvarchar(max)`, and `varbinary(max)`. To update data having large value data types, the `.WRITE` clause is used. The `.WRITE` clause specifies that a section of the value in a column is to be modified. The `.WRITE` clause cannot be used to update a `NULL` value in a column. Also, it cannot be used to set a column value to `NULL`.

Syntax:

```
column_name.WRITE (expression, @Offset, @Length)
```

where,

`column_name`: specifies the name of the large value data-type column.

`expression`: specifies the value that is copied to the column.

`@Offset`: specifies the starting point in the value of the column at which the expression is written.

`@Length`: specifies the length of the section in the column.

`@Offset` and `@Length` are specified in bytes for `varbinary` and `varchar` data types and in characters for the `nvarchar` data type.

Assume that the table **Product_Details** is modified to include a column **Description** having data type `nvarchar(max)`.

A view is created based on this table, having the columns **ProductName**, **Description**, and **Rate** as shown in Code Snippet 16.

Code Snippet 16:

```
CREATEVIEWvwProduct_Details AS  
SELECT  
ProductName,  
Description,  
Rate FROM Product_Details
```

Code Snippet 17 uses the UPDATE statement on the view **vwProduct_Details**. The .WRITE clause is used with values 0 and 2 to change first two characters in the Description column. Thus, Internal will be changed to External.

Code Snippet 17:

```
UPDATE vwProduct_Details  
SET Description.WRITE(N'Ex', 0, 2)  
WHERE ProductName='Portable Hard Drive'
```

As a result of the code, all the rows in the view that had 'Portable Hard Drive' as product name will be updated with External instead of Internal in the **Description** column.

Figure 10.6 shows a sample output of the view after the updation.

	ProductName	Description	Rate
1	Hard Disk Drive	Internal 120 GB	3570.00
2	Portable Hard Drive	External Drive 500 GB	5580.00
3	Portable Hard Drive	External Drive 500 GB	5580.00
4	Hard Disk Drive	Internal 120 GB	3570.00
5	Portable Hard Drive	External Drive 500 GB	5580.00

Figure 10.6: Updating a View

Following rules and guidelines must be followed when using the UPDATE statement:

The value of a column with an IDENTITY property cannot be updated.

Records cannot be updated if the base table contains a TIMESTAMP column.

When there is a self-join with the same view or base table, the UPDATE statement does not work.

While updating a row, if a constraint or rule is violated, the statement is terminated, an error is returned, and no records are updated.

10.3.3 DELETE with Views

SQL Server enables you to delete rows from a view. Rows can be deleted from the view using the DELETE statement. When rows are deleted from a view, corresponding rows are also deleted from the base table.

For example, consider a view **vwCustDetails** that lists the account information of different customers. When a customer closes the account, the details of this customer need to be deleted. This is done using the DELETE statement.

Following syntax is used to delete data from a view.

Syntax:

```
DELETE FROM <view_name>
WHERE <search_condition>
```

Assume that a table named **Customer_details** and a view **vwCustDetails** based on the table are created.

Code Snippet 18 is used to delete the record from the view **vwCustDetails** that has **CustID C0004**.

Code Snippet 18:

```
DELETE FROM vwCustDetails WHERE CustID='C0004'
```

Figure 10.7 depicts the logic of deleting from views.

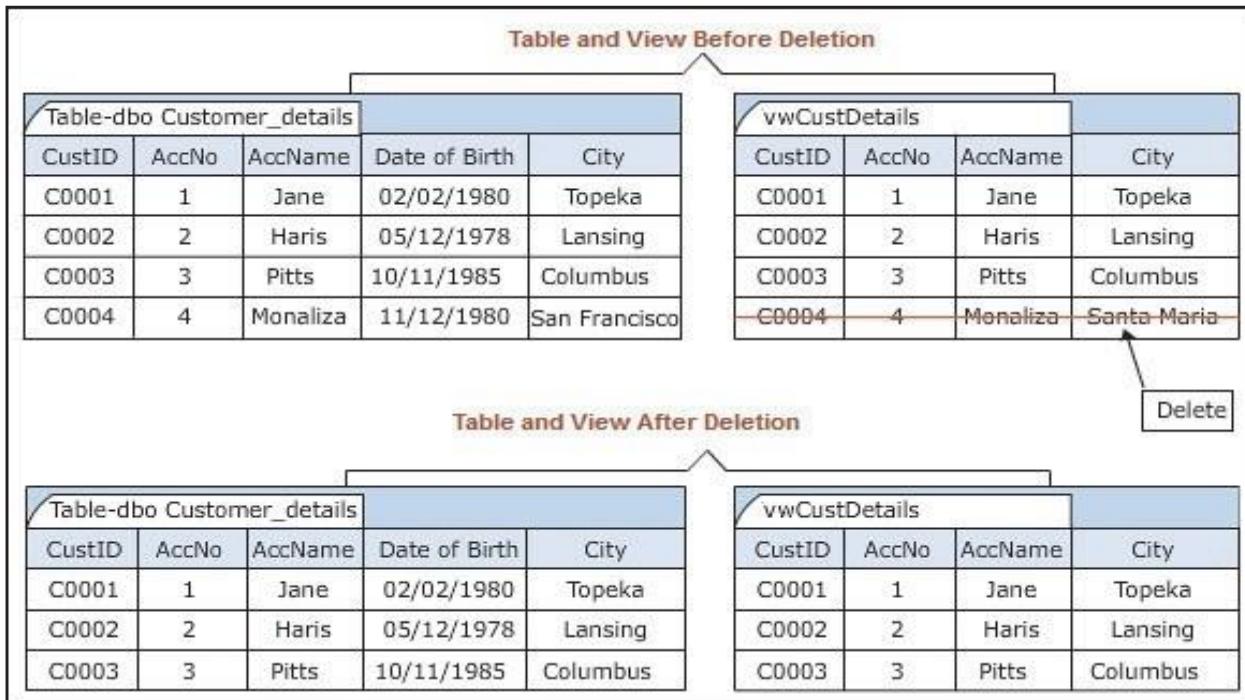


Figure 10.7: Deleting from Views

10.4 Altering Views

Besides modifying the data within a view, users can also modify the definition of a view. A view can be modified or altered by dropping and recreating it or executing the `ALTER VIEW` statement. The `ALTER VIEW` statement modifies an existing view without having to reorganize its permissions and other properties.

`ALTER VIEW` can be applied to indexed views; however, it unconditionally drops all indexes on the view. Views are often altered when a user requests for additional information or makes changes in the underlying table definition.

Note - After a view is created, if the structure of its underlying tables is altered by adding columns, the new columns do not appear in the view. This is because the column list is interpreted only when you first create the view. To see the new columns in the view, you must alter the view.

Following syntax is used to alter a view:

Syntax:

```
ALTERVIEW <view_name>
AS <select_statement>
```

Code Snippet 19 alters the view, `vwProductInfo` to include the `ReOrderPoint` column.

Code Snippet 19:

```
ALTERVIEW vwProductInfo AS
SELECT ProductID, ProductNumber, Name, SafetyStockLevel, ReOrderPoint
FROM Production.Product;
GO
```

10.5 Dropping Views

A view can be removed from the database if it is no longer required. This is done using the `DROP VIEW` statement. When a view is dropped, the data in the base tables remains unaffected. The definition of the view and other information associated with the view is deleted from the system catalog. All permissions for the view are also deleted. If a user queries any view that references the dropped view, the user receives an error message.

Note - The owner of the view has the permission to drop a view and this permission is nontransferable. However, the system administrator or database owner can drop any object by specifying the owner name in the `DROP VIEW` statement.

Following syntax is used to drop a view:

Syntax:

```
DROP VIEW <view_name>
```

Code Snippet 20 deletes the view, `vwProductInfo`.

Code Snippet 20:

```
DROP VIEW vwProductInfo
```

10.6 Definition of a View

The definition of a view helps to understand how its data is derived from the source tables. There are certain system stored procedures that help to retrieve view definitions. The `sp_helptext` stored procedure displays view related information when the name of the view is given as its parameter. Information about the definition of a view can be obtained if such information is not encrypted.

Following syntax is used to view the definition information of a view:

Syntax:

```
sp_helptext <view_name>
```

Code Snippet 21 displays information about the view, `vwEmployee_Personal_Details`.

Code Snippet 21:

```
EXEC sp_helptext vwEmployee_Personal_Details
```

The execution of the code will display the definition about the view as shown in figure 10.8.

Results	
	Text
1	CREATE VIEW vwEmployee_Personal_Details AS
2	SELECT e1.EmpID, FirstName, LastName, Designatio...
3	JOIN Employee_Salary_Details e2
4	ON e1.EmpID = e2.EmpID

Figure 10.8: Using `sp_helptext` to display View Definitions

10.7 Creating a View Using Built-in Functions

Views can be created using built-in functions of SQL Server. When functions are used, the derived column must include the column name in the `CREATE VIEW` statement.

Consider the view that was created in Code Snippet 16. It has been re-created in Code Snippet 22 to make use of the `AVG()` function. Make sure to delete the existing view before executing this snippet.

Code Snippet 22:

```
CREATE VIEW vwProduct_Details AS
SELECT
ProductName,
AVG(Rate) AS AverageRate FROM
Product_Details GROUP BY ProductName
```

Here, the `AVG()` function calculates the average rate of similar products by using a `GROUP BY` clause.

Figure 10.9 shows the result when the view is queried.

	ProductName	AverageRate
1	Hard Disk Drive	3570.00
2	Portable Hard Drive	5580.00

Figure 10.9: Using Built-in Functions with Views

10.8 CHECK OPTION

The `CHECK OPTION` is an option associated with the `CREATE VIEW` statement. It is used to ensure that all the updates in the view satisfy the conditions mentioned in the view definition. If the conditions are not satisfied, the database engine returns an error. Thus, the `CHECK OPTION` is used to enforce domain integrity; it checks the definition of the view to see that the `WHERE` conditions in the `SELECT` statement is not violated.

The `WITH CHECK OPTION` clause forces all the modification statements executed against the view to follow the condition set within the `SELECT` statement. When a row is modified, the `WITH CHECK OPTION` makes sure that the data remains visible through the view.

Following syntax creates a view using the `CHECK OPTION`:

Syntax:

```
CREATE VIEW <view_name>
AS select_statement [ WITH CHECK OPTION ]
```

where,

`WITH CHECK OPTION`: specifies that the modified data in the view continues to satisfy the view definition.

Code Snippet 23 re-creates the view `vwProductInfo` having `SafetyStockLevel` less than or equal to 1000.

Code Snippet 23:

```
CREATE VIEW vwProductInfo AS
SELECT ProductID, ProductNumber, Name, SafetyStockLevel,
ReOrderPoint
FROM Production.Product
WHERE SafetyStockLevel <=1000
WITH CHECK OPTION;
GO
```

In Code Snippet 24, the `UPDATE` statement is used to modify the view `vwProductInfo` by changing the value of the `SafetyStockLevel` column for the product having id 321 to 2500.

Code Snippet 24:

```
UPDATE vwProductInfo SET SafetyStockLevel=2500  
WHERE ProductID=321
```

The UPDATE statement fails to execute as it violates the view definition, which specifies that **SafetyStockLevel** must be less than or equal to 1000. Thus, no rows are affected in the view **vwProductInfo**.

Note - Any updates performed on the base tables are not verified against the view, even if **CHECK OPTION** is specified.

10.9 SCHEMABINDING Option

A view can be bound to the schema of the base table using the **SCHEMABINDING** option. This option can be used with **CREATE VIEW** or **ALTER VIEW** statements. When **SCHEMABINDING** option is specified, the base table or tables cannot be modified that would affect the view definition. The view definition must be first modified or deleted to remove dependencies on the table that is to be modified.

While using the **SCHEMABINDING** option in a view, you must specify the schema name along with the object name in the **SELECT** statement.

Following syntax is used to create a view with the **SCHEMABINDING** option:

Syntax:

```
CREATE VIEW <view_name> WITH SCHEMABINDING  
AS <select_statement>
```

where,

view_name: specifies the name of the view.

WITH SCHEMABINDING: specifies that the view must be bound to a schema.

select_statement: Specifies the **SELECT** statement that defines the view.

Code Snippet 25 creates a view **vwNewProductInfo** with **SCHEMABINDING** option to bind the view to the **Production** schema, which is the schema of the table **Product**.

Code Snippet 25:

```
CREATE VIEW vwNewProductInfo WITH SCHEMABINDING AS  
SELECT ProductID, ProductNumber, Name, SafetyStockLevel  
FROM Production.Product;  
GO
```

10.10 Using sp_refreshview

During the creation of a view, the **SCHEMABINDING** option is used to bind the view to the schema of the tables that are included in the view. However, a view can also be created without selecting the **SCHEMABINDING** option. In such a case, if changes are made to the underlying objects (tables or views) on

which the view depends, the `sp_refreshview` stored procedure should be executed. The `sp_refreshview` stored procedure updates the metadata for the view. If the `sp_refreshview` procedure is not executed, the metadata of the view is not updated to reflect the changes in the base tables. This results in the generation of unexpected results when the view is queried.

The `sp_refreshview` stored procedure returns code value zero if the execution is successful or returns a non-zero number in case the execution has failed.

Following syntax is used to run the `sp_refreshview` stored procedure:

Syntax:

```
sp_refreshview '<view_name>'
```

Code Snippet 26 creates a table **Customers** with the **CustID**, **CustName**, and **Address** columns.

Code Snippet 26:

```
CREATE TABLE Customers  
(  
    CustID int,  
    CustName varchar(50),  
    Address varchar(60)  
)
```

Code Snippet 27 creates a view **vwCustomers** based on the table **Customers**.

Code Snippet 27:

```
CREATE VIEW vwCustomers  
AS  
SELECT * FROM Customers
```

Code Snippet 28 executes the `SELECT` query on the view.

Code Snippet 28:

```
SELECT * FROM vwCustomers
```

The output of Code Snippet 28 shows three columns, **CustID**, **CustName**, and **Address**.

Code Snippet 29 uses the `ALTER TABLE` statement to add a column **Age** to the table **Customers**.

Code Snippet 29:

```
ALTER TABLE Customers ADD Age int
```

Code Snippet 30 executes the `SELECT` query on the view.

Code Snippet 30:

```
SELECT * FROM vwCustomers
```

The updated column **Age** is not seen in the view.

To resolve this, the `sp_refreshview` stored procedure must be executed on the view `vwCustomers` as shown in Code Snippet 31.

Code Snippet 31:

```
EXEC sp_refreshview 'vwCustomers'
```

When a `SELECT` query is run again on the view, the column **Age** is seen in the output. This is because the `sp_refreshview` procedure refreshes the metadata for the view `vwCustomers`.

Tables that are schema-bound to a view cannot be dropped unless the view is dropped or changed such that it no longer has schema binding. If the view is not dropped or changed and you attempt to drop the table, the Database Engine returns an error message.

Also, when an `ALTER TABLE` statement affects the view definition of a schema-bound view, the `ALTER TABLE` statement fails.

Consider the schema-bound view that was created in Code Snippet 25. It is dependent on the `Production.Product` table.

Code Snippet 32 tries to modify the data type of `ProductID` column in the `Production.Product` table from `int` to `varchar(7)`.

Code Snippet 32:

```
ALTER TABLE Production.Product ALTER COLUMN ProductID varchar(7)
```

The Database Engine returns an error message as the table is schema-bound to the `vwNewProductInfo` view and hence, cannot be altered such that it violates the view definition of the view.

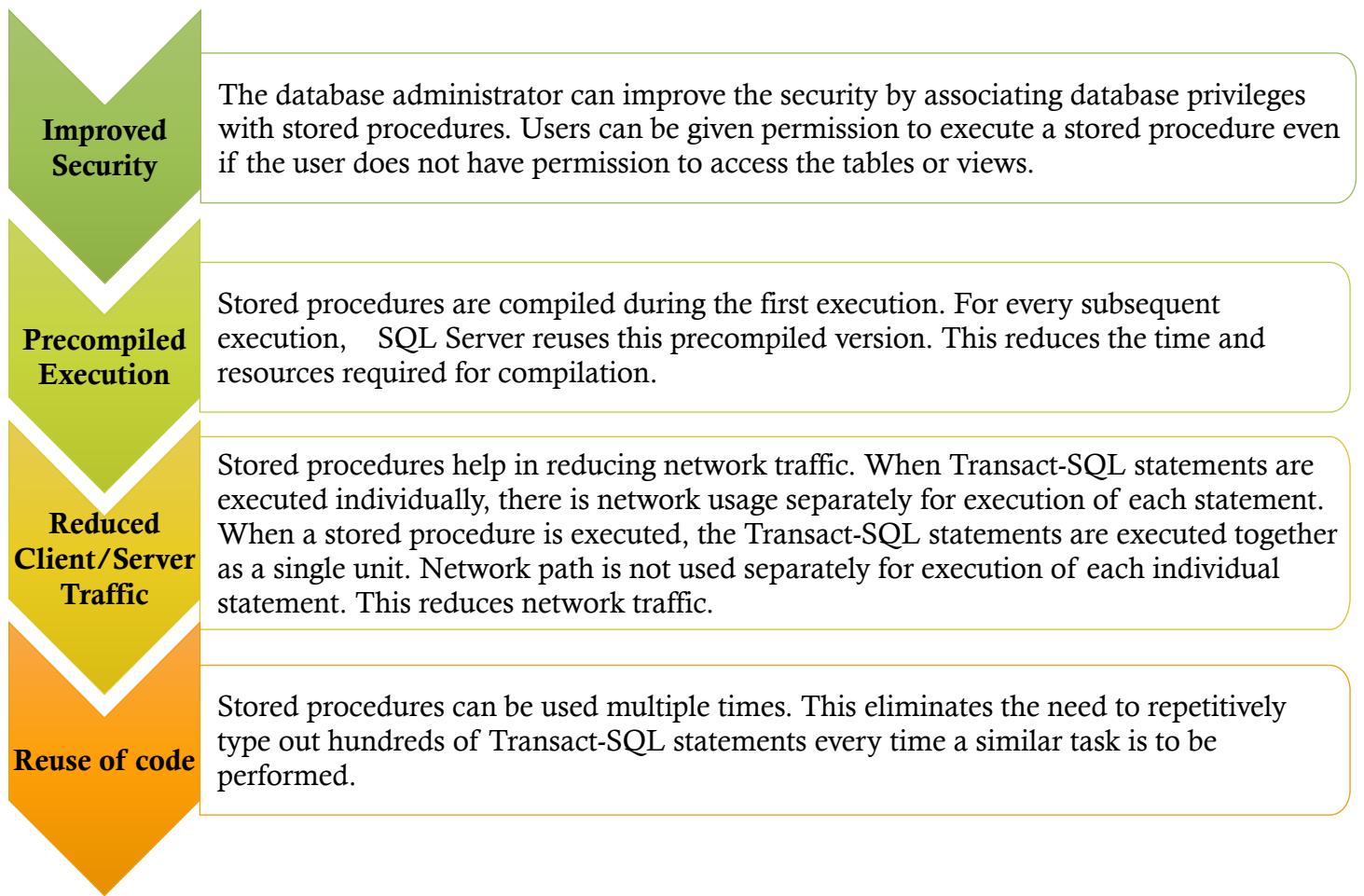
10.11 Stored Procedures

A stored procedure is a group of Transact-SQL statements that act as a single block of code that performs a specific task. This block of code is identified by an assigned name and is stored in the database in a compiled form. A stored procedure may also be a reference to a .NET Framework Common Language Runtime (CLR) method.

Stored procedures are useful when repetitive tasks have to be performed. This eliminates the need for repetitively typing out multiple Transact-SQL statements and then repetitively compiling them.

Stored procedures can accept values in the form of input parameters and return output values as defined by the output parameters.

Using stored procedures offers numerous advantages over using Transact-SQL statements. These are as follows:



10.11.1 Types of Stored Procedures

SQL Server supports various types of stored procedures. These are described as follows:

➤ **User-Defined Stored Procedures**

User-defined stored procedures are also known as custom stored procedures. These procedures are used for reusing Transact-SQL statements for performing repetitive tasks. There are two types of user-defined stored procedures, the Transact-SQL stored procedures and the Common Language Runtime (CLR) stored procedures.

Transact-SQL stored procedures consist of Transact-SQL statements whereas the CLR stored procedures are based on the .NET framework CLR methods. Both the stored procedures can take and return user-defined parameters.

➤ **Extended Stored Procedures**

Extended stored procedures help SQL Server in interacting with the operating system. Extended stored procedures are not resident objects of SQL Server. They are procedures that are implemented as Dynamic-Link Libraries (DLL) executed outside the SQL Server environment. The application interacting with SQL

Server calls the DLL at run-time. The DLL is dynamically loaded and run by SQL Server. SQL Server allots space to run the extended stored procedures. Extended stored procedures use the 'xp' prefix. Tasks that are complicated or cannot be executed using Transact-SQL statements are performed using extended stored procedures.

➤ System Stored Procedures

System stored procedures are commonly used for interacting with system tables and performing administrative tasks such as updating system tables. The system stored procedures are prefixed with 'sp_'. These procedures are located in the **Resource** database. These procedures can be seen in the sys schema of every system and user-defined database. System stored procedures allow GRANT, DENY, and REVOKE permissions.

A system stored procedure is a set of pre-compiled Transact-SQL statements executed as a single unit. System procedures are used in database administrative and informational activities. These procedures provide easy access to the metadata information about database objects such as system tables, user-defined tables, views, and indexes.

System stored procedures logically appear in the sys schema of system and user-defined databases. When referencing a system stored procedure, the **sys** schema identifier is used. System stored procedures are stored physically in the hidden **Resource** database and have the **sp_** prefix. System stored procedures are owned by the database administrator.

Note - System tables are created by default at the time of creating a new database. These tables store the metadata information about user-defined objects such as tables and views. Users cannot access or update the system tables using system stored procedures except through permissions granted by a database administrator.

10.11.2 Classification of System Stored Procedures

System stored procedures can be classified into different categories depending on the tasks they perform.

Some of the important categories are as follows:

Catalog Stored Procedures

All information about tables in the user database is stored in a set of tables called the system catalog. Information from the system catalog can be accessed using catalog procedures. For example, the sp_tables catalog stored procedure displays the list of all the tables in the current database.

Security Stored Procedures

Security stored procedures are used to manage the security of the database. For example, the sp_changedbowner security stored procedure is used to change the owner of the current database.

Cursor Stored Procedures

Cursor procedures are used to implement the functionality of a cursor. For example, the sp_cursor_list cursor stored procedure lists all the cursors opened by the connection and describes their attributes.

Distributed Query Stored Procedures

Distributed stored procedures are used in the management of distributed queries. For example, the sp_indexes distributed query stored procedure returns index information for the specified remote table.

Database Mail and SQL Mail Stored Procedures

Database Mail and SQL Mail stored procedures are used to perform e-mail operations from within the SQL Server. For example, the sp_send_dbmail database mail stored procedure sends e-mail messages to specified recipients. The message may include a query resultset or file attachments or both.

10.11.3 Temporary Stored Procedures

Stored procedures created for temporary use within a session are called temporary stored procedures. These procedures are stored in the `tempdb` database. The `tempdb` system database is a global resource available to all users connected to an instance of SQL Server. It holds all temporary tables and temporary stored procedures.

SQL Server supports two types of temporary stored procedures namely, local and global. The differences between the two types are given in table 10.1.

Local Temporary Procedure	Global Temporary Procedure
Visible only to the user that created it	Visible to all users
Dropped at the end of the current session	Dropped at the end of the last session
Local Temporary Procedure	Global Temporary Procedure
Can only be used by its owner	Can be used by any user
Uses the # prefix before the procedure name	Uses the ## prefix before the procedure name

Table 10.1: Differences Between Local and Global Temporary Procedures

Note - A session is established when a user connects to the database and is ended when the user disconnects. The complete name of a global temporary stored procedure including the prefix # # cannot exceed 128 characters. The complete name of a local temporary stored procedure including the prefix # cannot exceed 116 characters.

10.11.4 Remote Stored Procedures

Stored procedures that run on remote SQL Servers are known as remote stored procedures. Remote stored procedures can be used only when the remote server allows remote access. When a remote stored procedure is executed from a local instance of SQL Server to a client computer, a statement abort error might be encountered. When such an error occurs, the statement that caused the error is terminated but the remote procedure continues to be executed.

10.11.5 Extended Stored Procedures

Extended stored procedures are used to perform tasks that are unable to be performed using standard Transact-SQL statements. Extended stored procedures use the 'xp_' prefix. These stored procedures are

contained in the `dbo` schema of the master database.

Syntax:

```
EXECUTE <procedure_name>
```

Code Snippet 33 executes the extended stored procedure `xp_fileexist` to check whether the `MyTest.txt` file exists or not.

Code Snippet 33:

```
EXECUTE xp_fileexist 'c:\MyTest.txt'
```

Note - When you execute an extended stored procedure, either in a batch or in a module, qualify the stored procedure name with `master.dbo`.

10.12 Custom or User-defined Stored Procedures

In SQL Server, users are allowed to create customized stored procedures for performance of various tasks. Such stored procedures are referred to as user-defined or custom stored procedures.

For example, consider a table `Customer_Details` that stores details about all the customers. You would need to type out Transact-SQL statements every time you wished to view the details about the customers. Instead, you could create a custom stored procedure that would display these details whenever the procedure is executed.

Creating a custom stored procedure requires `CREATE PROCEDURE` permission in the database and `ALTER` permission on the schema in which the procedure is being created.

Following syntax is used to create a custom stored procedure:

Syntax:

```
CREATE { PROC | PROCEDURE } procedure_name  
[ { @parameter data_type } ]  
AS <sql_statement>
```

where,

`procedure_name`: specifies the name of the procedure.

`@parameter`: specifies the input/output parameters in the procedure.

`data_type`: specifies the data types of the parameters.

`sql_statement`: specifies one or more Transact-SQL statements to be included in the procedure.

Code Snippet 34 creates and then executes a custom stored procedure, `uspGetCustTerritory`, which will display the details of customers such as customer id, territory id, and territory name.

Code Snippet 34:

```
CREATE PROCEDURE uspGetCustTerritory
AS
SELECT TOP 10 CustomerID, Customer.TerritoryID, Sales.SalesTerritory.Name
FROM Sales.Customer JOIN Sales.SalesTerritory ON Sales.Customer.TerritoryID =
Sales.SalesTerritory.TerritoryID
```

To execute the stored procedure, the `EXEC` command is used as shown in Code Snippet 35.

Code Snippet 35:

```
EXEC uspGetCustTerritory
```

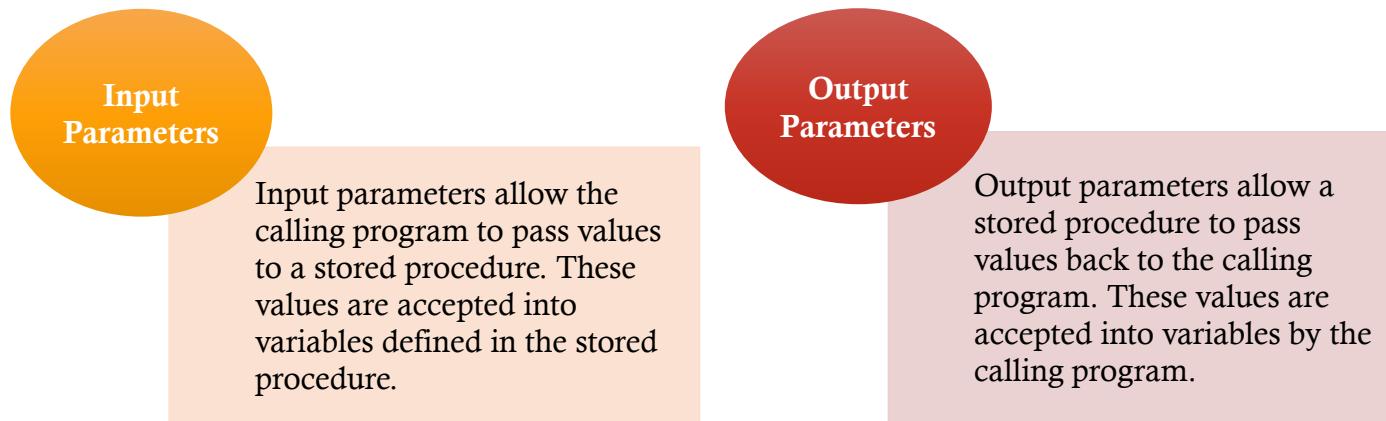
The output is shown in figure 10.10.

	CustomerID	TerritoryID	Name
1	15	9	Australia
2	33	9	Australia
3	51	9	Australia
4	69	9	Australia
5	87	9	Australia
6	105	9	Australia
7	123	9	Australia
8	141	9	Australia
9	159	9	Australia
10	177	9	Australia

Figure 10.10: Output of a Simple Stored Procedure

10.12.1 Using Parameters

The real advantage of a stored procedure comes into picture only when one or more parameters are used with it. Data is passed between the stored procedure and the calling program when a call is made to a stored procedure. This data transfer is done using parameters. Parameters are of two types that are as follows:



These are now described in detail.

Input Parameters

Values are passed from the calling program to the stored procedure and these values are accepted into the input parameters of the stored procedure. The input parameters are defined at the time of creation of the

stored procedure. The values passed to input parameters could be either constants or variables. These values are passed to the procedure at the time of calling the procedure. The stored procedure performs specified tasks using these values.

Following syntax is used to create a stored procedure:

Syntax:

```
CREATE PROCEDURE <procedure_name>
@parameter <data_type>
AS <sql statement>
```

where,

`data_type`: specifies the system defined data type.

Following syntax is used to execute a stored procedure and pass values as input parameters:

Syntax:

```
EXECUTE <procedure_name> <parameters>
```

Code Snippet 36 creates a stored procedure, `uspGetSales` with a parameter `territory` to accept the name of a territory and display the sales details and salesperson id for that territory. Then, the code executes the stored procedure with `Northwest` being passed as the input parameter.

Code Snippet 36:

```
CREATE PROCEDURE uspGetSales @territory varchar(40) AS
SELECT BusinessEntityID, B.SalesYTD, B.SalesLastYear FROM
Sales.SalesPersonA
JOIN Sales.SalesTerritoryB
ON A.TerritoryID=B.TerritoryID WHERE
B.Name = @territory;
--Execute the stored procedure EXEC
uspGetSales 'Northwest'
```

The output is shown in figure 10.11.

	BusinessEntityID	SalesYTD	SalesLastYear
1	280	7887186.7882	3298694.4938
2	283	7887186.7882	3298694.4938
3	284	7887186.7882	3298694.4938

Figure 10.11: Using Stored Procedure with Parameters

Output Parameters

Stored procedures occasionally need to return output back to the calling program. This transfer of data from the stored procedure to the calling program is performed using output parameters. Output parameters are defined at the time of creation of the procedure. To specify an output parameter, the `OUTPUT` keyword is used while declaring the parameter. Also, the calling statement has to have a variable specified with the

`OUTPUT` keyword to accept the output from the called procedure.

Following syntax is used to pass output parameters in a stored procedure and then, execute the stored procedure with the `OUTPUT` parameter specified:

Syntax:

```
EXECUTE <procedure_name> <parameters>
```

Code Snippet 37 creates a stored procedure, `uspGetTotalSales` with input parameter `@territory` to accept the name of a territory and output parameter `@sum` to display the sum of sales year to date in that territory.

Code Snippet 37:

```
CREATE PROCEDURE uspGetTotalSales @territory  
varchar(40), @sum int OUTPUT AS  
SELECT @sum=SUM(B.SalesYTD) FROM  
Sales.SalesPerson A JOIN  
Sales.SalesTerritoryB  
ON A.TerritoryID=B.TerritoryID  
WHERE B.Name=@territory
```

Code Snippet 38 declares a variable `sumsales` to accept the output of the procedure `uspGetTotalSales`.

Code Snippet 38:

```
DECLARE @sumsales money  
  
EXEC uspGetTotalSales 'Northwest', @sumsales OUTPUT  
  
PRINT 'The year-to-date sales figure for this territory is ' + convert (varchar(100),  
@sumsales);
```

The code passes `Northwest` as the input to the `uspGetTotalSales` stored procedure and accepts the output in the variable `sumsales`. The output is printed using the `PRINT` command.

`OUTPUT` parameters have following characteristics:

- The parameter cannot be of text and image data type.
- The calling statement must contain a variable to receive the return value.
- The variable can be used in subsequent Transact-SQL statements in the batch or the calling procedure.
- Output parameters can be cursor placeholders.

The `OUTPUT` clause returns information from each row on which the `INSERT`, `UPDATE`, and `DELETE` statements have been executed. This clause is useful to retrieve the value of an identity or computed column after an `INSERT` or `UPDATE` operation.

10.12.2 Using SSMS to Create Stored Procedures

You can also create a user-defined stored procedure using SSMS. Steps to perform this are as follows:

1. Launch **Object Explorer**.
2. In **Object Explorer**, connect to an instance of Database Engine and after successfully connection, expand the instance.
3. Expand the Databases node and then, expand the **AdventureWorks2019** database.

Expand **Programmability**, right-click **Stored Procedures**, and then, click **Stored Procedure** as shown in figure 10.12.

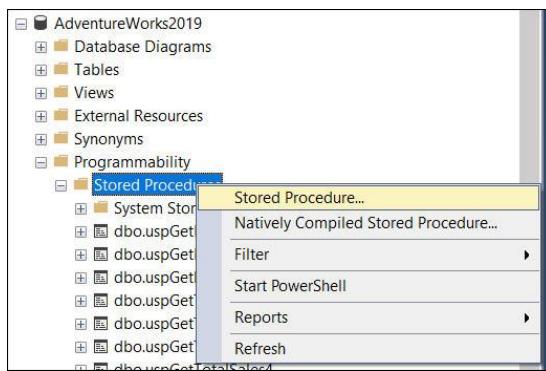


Figure 10.12: Creating a Stored Procedure

The Query Editor displays a default template in Transact SQL for the stored procedure, as shown in figure 10.13.

```
1 -- =====
2 -- Template generated from Template Explorer using:
3 -- Create Procedure (New Menu).SQL
4 --
5 -- Use the Specify Values for Template Parameters
6 -- command (Ctrl-Shift-M) to fill in the parameter
7 -- values below.
8 --
9 -- This block of comments will not be included in
10 -- the definition of the procedure.
11 -- =====
12 SET ANSI_NULLS ON
13 GO
14 SET QUOTED_IDENTIFIER ON
15 GO
16 -- =====
17 -- Author:      <Author,,Name>
18 -- Create date: <Create Date,,>
19 -- Description: <Description,,>
20 -- =====
21 CREATE PROCEDURE <Procedure_Name, sysname, ProcedureName>
22     -- Add the parameters for the stored procedure here
23     <@Param1, sysname, @p1> <Datatype_For_Param1, , int> = <Default_Value_For_Param1, ,
24     <@Param2, sysname, @p2> <Datatype_For_Param2, , int> = <Default_Value_For_Param2, ,
25 AS
26 BEGIN
27     -- SET NOCOUNT ON added to prevent extra result sets from
28     -- interfering with SELECT statements.
29     SET NOCOUNT ON;
```

Figure 10.13: Default Template for Stored Procedure

4. On the **Query** menu, click **Specify Values for Template Parameters**, as shown in figure 10.14.

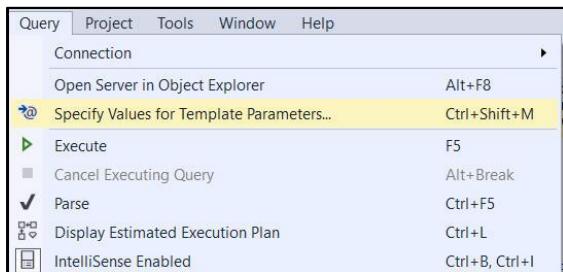


Figure 10.14: Selecting Query Menu Option

The **Specify Values for Template Parameters** dialog box is displayed as shown in figure 10.15.

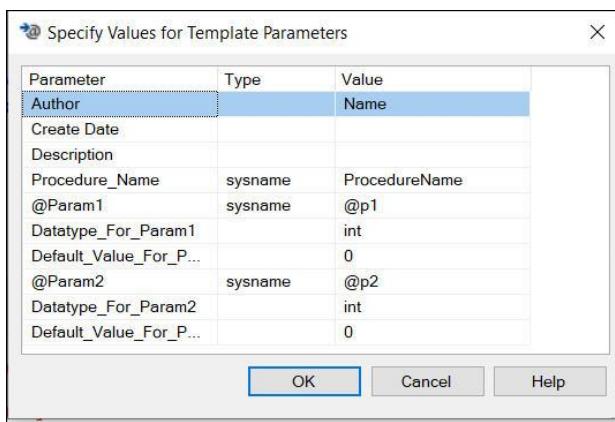


Figure 10.15: Specify Values for Template Parameters Dialog Box

5. In the **Specify Values for Template Parameters** dialog box, enter the values for the parameters as shown in table 10.2.

Parameter	Value
Author	Your name
Create Date	Today's date
Description	Returns year to sales data for a territory
Procedure_Name	uspGetTotals
@Param1	@territory
@Datatype_For_Param1	varchar(50)
Default_Value_For_Param1	NULL
@Param2	
@Datatype_For_Param2	
Default_Value_For_Param2	

Table 10.2: Parameter Values

6. After entering these details, click **OK**.

7. In the Query Editor, replace the `SELECT` statement (on line 31) with following statement:


```
SELECT BusinessEntityID, B.SalesYTD, B.SalesLastYear FROM Sales.SalesPerson
A JOIN Sales.SalesTerritory B
ON A.TerritoryID = B.TerritoryID WHERE B.Name = @territory;
```
8. Comment following line:


```
SELECT 'Return Value' = @return_value
```
9. To test the syntax, on the **Query** menu, click **Parse**. If an error message is returned, compare the statements with the information and correct as required.
10. To create the procedure, from the **Query** menu, click **Execute**. The procedure is created as an object in the database.
11. To see the procedure listed in **Object Explorer**, right-click **Stored Procedures** and select **Refresh**.

The procedure name will be displayed in the **Object Explorer** tree as shown in figure 10.16.

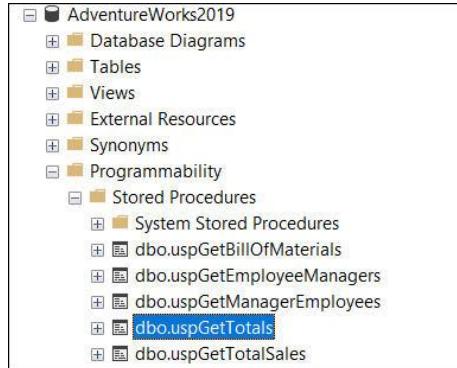


Figure 10.16: Stored Procedure Seen in Object Explorer

12. To run the procedure, in **Object Explorer**, right-click the stored procedure name `uspGetTotals` and select **Execute Stored Procedure**.
13. In the **Execute Procedure** window, enter `Northwest` as the value for the parameter `@territory` and click **OK**. The procedure will be executed and the output will be displayed, as shown in figure 10.17.

Results			
	BusinessEntityID	SalesYTD	SalesLastYear
1	280	7887186.7882	3298694.4938
2	283	7887186.7882	3298694.4938
3	284	7887186.7882	3298694.4938

Figure 10.17: Output of Stored Procedure Created Through SSMS

10.13 Modifying and Dropping Stored Procedures

The permissions associated with the stored procedure are lost when a stored procedure is re-created. However, when a stored procedure is altered, the permissions defined for the stored procedure remain the

same even though the procedure definition is changed.

A procedure can be altered using the `ALTER PROCEDURE` statement.

Following syntax is used to modify a stored procedure:

Syntax:

```
ALTER PROCEDURE <procedure_name>
@parameter <data_type> [ OUTPUT ]  [
WITH { ENCRYPTION | RECOMPILE } ]
AS <sql_statement>
```

where,

`ENCRYPTION`: encrypts the stored procedure definition.

`RECOMPILE`: indicates that the procedure is compiled at run-time.

`sql_statement`: specifies Transact-SQL statements to be included in the body of the procedure.

Code Snippet 39 modifies the definition of the stored procedure named `uspGetTotals` to add a new column `CostYTD` to be retrieved from `Sales.SalesTerritory`.

Code Snippet 39:

```
ALTER PROCEDURE [dbo].[uspGetTotals]
    @territory varchar = 40
AS
    SELECT BusinessEntityID, B.SalesYTD, B.CostYTD, B.SalesLastYear FROM
        Sales.SalesPersonA JOIN Sales.SalesTerritoryB
        ON A.TerritoryID = B.TerritoryID
    WHERE B.Name = @territory;
GO
```

Note - When you change the definition of a stored procedure, the dependent objects may fail when executed. This happens if the dependent objects are not updated to reflect the changes made to the stored procedure.

Dropping stored procedures

Stored procedures can be dropped if they are no longer required. If another stored procedure calls a deleted procedure, an error message is displayed.

If a new procedure is created using the same name as well as the same parameters as the dropped procedure, all calls to the dropped procedure will be executed successfully. This is because they will now refer to the new procedure, which has the same name and parameters as the deleted procedure.

Before dropping a stored procedure, execute the `sp_depends` system stored procedure to determine which objects depend on the procedure.

A procedure is dropped using the `DROP PROCEDURE` statement.

Following syntax is used to drop a stored procedure:

Syntax:

```
DROP PROCEDURE <procedure_name>
```

Code Snippet 40 drops (removes) the stored procedure, **uspGetTotals**.

Code Snippet 40:

```
DROP PROCEDURE uspGetTotals
```

10.14 Nested Stored Procedures

SQL Server 2019 enables stored procedures to be called inside other stored procedures. The called procedures can in turn call other procedures. This architecture of calling one procedure from another procedure is referred to as nested stored procedure architecture.

When a stored procedure calls another stored procedure, the level of nesting is said to be increased by one. Similarly, when a called procedure completes its execution and passes control back to the calling procedure, the level of nesting is said to be decreased by one. The maximum level of nesting supported by SQL Server 2019 is 32.

Code Snippet 41 is used to create a stored procedure **NestedProcedure** that calls two other stored procedures that were created earlier through Code Snippets 34 and 36.

Code Snippet 41:

```
CREATE PROCEDURE NestedProcedure AS
BEGIN
EXEC uspGetCustTerritory
EXEC uspGetSales 'France'
END
```

When the procedure **NestedProcedure** is executed, this procedure in turn invokes the **uspGetCustTerritory** and **uspGetSales** stored procedures and passes the value France as the input parameter to the **uspGetSales** stored procedure.

Note - Although there can be a maximum of 32 levels of nesting, there is no limit as to the number of stored procedure that can be called from a given stored procedure.

10.15 Querying System MetaData

The properties of an object such as a table or a view are stored in special system tables. These properties are referred to as metadata. All SQL objects produce metadata. This metadata can be viewed using system views, which are predefined views of SQL Server.

There are over 230 different system views and these are automatically inserted into the user created database. These views are grouped into several different schemas.

System Catalog Views

These contain information about the catalog in a SQL Server system. A catalog is similar to an inventory of objects. These views contain a wide range of metadata. In earlier versions of SQL Server, users were required to query a large number of system tables, system views, and system functions. In SQL Server 2012, all user-accessible catalog metadata can easily be found by querying just the catalog views.

Code Snippet 42 retrieves a list of user tables and attributes from the system catalog view `sys.tables`.

Code Snippet 42:

```
SELECT name, object_id, type, type_desc  
FROM sys.tables;
```

Information Schema Views

Users can query information schema views to return system metadata. These views are useful to third-party tools that may not be specific for SQL Server. Information schema views provide an internal, system table-independent view of the SQL Server metadata. Information schema views enable applications to work correctly although significant changes have been made to the underlying system tables.

Following points in table 10.3 will help to decide whether one should query SQL Server-specific system views or information schema views:

Information Schema Views	SQL Server System Views
They are stored in their own schema, <code>INFORMATION_SCHEMA</code> .	They appear in the <code>sys</code> schema.
They use standard terminology instead of SQL Server terms. For example, they use catalog instead of database and domain instead of user-defined data type.	They adhere to SQL Server terminology.
They may not expose all the metadata available to SQL Server's own catalog views. For example, <code>sys.columns</code> includes attributes for identity property and computed column property, while <code>INFORMATION_SCHEMA.columns</code> does not.	They can expose all the metadata available to SQL Server's catalog views.

Table 10.3: Information Schema Views and SQL Server System Views

Code Snippet 43 retrieves data from the `INFORMATION_SCHEMA.TABLES` view in the `AdventureWorks2019` database.

Code Snippet 43:

```
SELECT TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME, TABLE_TYPE  
FROM INFORMATION_SCHEMA.TABLES;
```

System Metadata Functions

In addition to views, SQL Server provides a number of built-in functions that return metadata to a query. These include scalar functions and table-valued functions, which can return information about system

settings, session options, and a wide range of objects.

SQL Server metadata functions come in a variety of formats. Some appear similar to standard scalar functions, such as `ERROR_NUMBER()`. Others use special prefixes, such as `@@VERSION` or `$PARTITION`.

Table 10.4 shows some common system metadata functions.

Function Name	Description	Example
<code>OBJECT_ID(<object_name>)</code>	Returns the object ID of a database object.	<code>OBJECT_ID('Sales.Customer')</code>
<code>OBJECT_NAME(<object_id>)</code>	Returns the name corresponding to an object ID.	<code>OBJECT_NAME(197575742)</code>
<code>@@ERROR</code>	Returns 0 if the last statement succeeded; otherwise returns the error number.	<code>@@ERROR</code>
<code>SERVERPROPERTY(<property>)</code>	Returns the value of the specified server property.	<code>SERVERPROPERTY('Collation')</code>

Table 10.4: Common System Metadata Functions

Code Snippet 44 uses a `SELECT` statement to query a system metadata function.

Code Snippet 44:

```
SELECT SERVERPROPERTY('EDITION') AS EditionName;
```

One of the new features in SQL Server 2019 is memory-optimized `tempdb` metadata. The SQL Server team enhanced tempdb code with optimizations so that some of the metadata which could have been a bottleneck on tempdb heavy systems can now rely on memory and be optimized for RAM access.

Large volume, large scale environments that use a lot of tempdb usually run into this type of bottleneck. Earlier this would require some sort of workaround to reduce the use of tempdb. However, with this new feature in place, it is possible to enable metadata to remain in memory and be optimally accessed.

10.16 Querying Dynamic Management Objects

First introduced in SQL Server 2005, Dynamic Management Views (DMVs) and Dynamic Management Functions (DMFs) are dynamic management objects that return server and database state information. DMVs and DMFs are collectively referred to as dynamic management objects. They provide useful insight into the working of software and can be used for examining the state of SQL Server instance, troubleshooting, and performance tuning.

Both DMVs and DMFs return data in tabular format but the difference is that while a DMF normally accepts at least one parameter, a DMV does not accept parameters. SQL Server 2019 provides nearly 200 dynamic management objects. In order to query DMVs, it is required to have `VIEW SERVER STATE` or `VIEW DATABASE STATE` permission, depending on the scope of the DMV.

10.16.1 Categorizing and Querying DMVs

Table 10.5 lists the naming convention that helps organize the DMVs by function.

Naming Pattern	Description
db	Database related
io	I/O statistics
Os	SQL Server Operating System Information
'tran'	Transaction-related
'exec'	Query execution-related metadata

Table 10.5: Organizing DMVs by Function

To query a dynamic management object, you use a `SELECT` statement as you would with any user-defined view or table-valued function. For example, Code Snippet 45 returns a list of current user connections from the `sys.dm_exec_sessions` view.

`sys.dm_exec_sessions` is a server-scoped DMV that displays information about all active user connections and internal tasks. This information includes login user, current session setting, client version, client program name, client login time, and more. The `sys.dm_exec_sessions` can be used to identify a specific session and find information about it.

Code Snippet 45:

```
SELECT session_id, login_time, program_name FROM sys.dm_exec_sessions  
WHERE login_name = 'sa' and is_user_process = 1;
```

Here, `is_user_process` is a column in the view that determines if the session is a system session or not. A value of 1 indicates that it is not a system session but rather a user session. The `program_name` column determines the name of client program that initiated the session. The `login_time` column establishes the time when the session began. The output of Code Snippet 45 is shown in figure 10.18.

	session_id	login_time	program_name
1	54	2020-11-11 08:27:27.390	Microsoft SQL Server Management Studio
2	55	2020-11-11 08:27:37.180	Microsoft SQL Server Management Studio - Query
3	56	2020-11-11 08:27:42.287	Microsoft SQL Server Management Studio - Transa...

Figure 10.18: Querying the `sys.dm_exec_sessions` DMV

10.17 Check Your Progress

1. Which of these statements about views are true?

a.	VIEWS enable you to see and manipulate selected parts of a table.
b.	Only columns from a table can be selected for a view, rows cannot be.
c.	System views display information about the system or the machine.
d.	Views have a maximum of 1024 columns.
e.	When data in a view is changed, it is not reflected in the underlying table.

(A)	a, c, d, e	(C)	a, b, d
(B)	a, d	(D)	All of these

2. You are creating a view **vwSupplier** with **FirstName**, **LastName**, and **City** columns from the **Supplier_Details** table. Which of the following code is violating the definition of a view?

(A)	CREATE VIEW vwSupplier AS SELECT FirstName, LastName, City FROM Supplier_Details WHERE City IN('New York', 'Boston', 'Orlando')	(C)	CREATE VIEW vwSupplier AS SELECT FirstName, LastName, City FROM Supplier_Details ORDER BY FirstName
(B)	CREATE VIEW vwSupplier AS SELECT TOP 100 FirstName, LastName, City FROM Supplier_Details WHERE FirstName LIKE 'A%' ORDER BY FirstName	(D)	CREATE VIEW vwSupplier AS SELECT TOP 100 FirstName, LastName, City FROM Supplier_Details

3. Which of these statements about CHECK OPTION and SCHEMABINDING options are true?

a.	The CHECK OPTION ensures entity integrity.
b.	The SCHEMABINDING option binds the view to the schema of the base table.
c.	When a row is modified, the WITH CHECK OPTION makes sure that the data remains visible through the view.
d.	SCHEMABINDING option ensures the base table cannot be modified in a way that would affect the view definition.
e.	SCHEMABINDING option cannot be used with ALTER VIEW statements.

(A)	a, b, c	(C)	b, c, d
(B)	b, c	(D)	c, d, e

4. You want to create a view **vwAccount_Details** with the SCHEMABINDING option. Which of the following code will achieve this objective?

(A)	CREATE VIEW vwAccount_Details AS SELECT AccNo, City FROM dbo.Customer_Details WITH SCHEMABINDING	(C)	CREATE VIEW vwAccount_Details WITH SCHEMABINDING AS SELECT AccNo, City FROM dbo.Customer_Details
(B)	CREATE VIEW vwAccount_Details SCHEMABINDING AS SELECT AccNo, City FROM Customer_Details	(D)	CREATE VIEW vwAccount_Details WITH SCHEMABINDING AS SELECT AccNo, City FROM Customer_Details

5. A table **Item_Details** is created with **ItemCode**, **ItemName**, **Price**, and **Quantity** columns. The **ItemCode** column is defined as the PRIMARY KEY, **ItemName** is defined with UNIQUE and NOT NULL constraints, **Price** is defined with the NOT NULL constraint, and **Quantity** is defined with the NOT NULL constraint and having a default value specified. Which of the following views created using columns from the **Item_Details** table can be used to insert records in the table?

(A)	CREATE VIEW vwItemDetails AS SELECT ItemCode, ItemName, Price FROM Item_Details	(C)	CREATE VIEW vwItemDetails AS SELECT ItemName, Price, Quantity FROM Item_Details
(B)	CREATE VIEW vwItemDetails AS SELECT ItemCode, Price, Quantity FROM Item_Details	(D)	CREATE VIEW vwItemDetails AS SELECT ItemCode, ItemName, Quantity FROM Item_Details

6. Which of these statements about stored procedures are true?

a.	A stored procedure is a group of Transact-SQL statements that act as a block of code used to perform a particular task.
b.	All system stored procedures are identified by the 'xp_' prefix.
c.	A distributed stored procedure is used in the management of distributed queries.
d.	Database Mail and SQL mail procedures are used to perform e-mail operations within SQL Server.
e.	User-defined stored procedures are also known as custom stored procedures.

(A)	a, d	(C)	a, c, d, e
(B)	b, c, e	(D)	d

10.17.1 Answers

1.	A
2.	C
3.	C
4.	B
5.	A
6.	C

Summary

- A view is a virtual table that is made up of selected columns from one or more tables and is created using the CREATE VIEW command in SQL Server.
- Users can manipulate the data in views, such as inserting into views, modifying the data in views, and deleting from views.
- A stored procedure is a group of Transact-SQL statements that act as a single block of code that performs a specific task.
- SQL Server supports various types of stored procedures, such as User-Defined Stored Procedures, Extended Stored Procedures, and System Stored Procedures.
- System stored procedures can be classified into different categories such as Catalog Stored Procedures, Security Stored Procedures, and Cursor Stored Procedures.
- Input and output parameters can be used with stored procedures to pass and receive data from stored procedures.
- The properties of an object such as a table or a view are stored in special system tables and are referred to as metadata.
- DMVs and DMFs are dynamic management objects that return server and database state information. DMVs and DMFs are collectively referred to as dynamic management objects.

Try It Yourself

1. In SQL Server Management Studio, locate the extended stored procedures defined under the master database and execute the following procedures in a query window:

```
sys.xp_readerrorlog sys.xp_getnetname sys.xp_fixeddrives
```

2. ShoezUnlimited is a trendy shoe store based in Miami. It stocks various kinds of footwear in its store and sells them for profits. ShoezUnlimited maintains the details of all products in an SQL Server 2019 database. The management wants their developer to make use of stored procedures for commonly performed tasks. Assuming that you are the developer, perform the following tasks:

- a) Create the Shoes table having structure as shown in table 10.6 in the database, ShoezUnlimited.

Field Name	Data Type	Key Field	Description
ProductCode	varchar(5)	Primary Key	Product Code that uniquely identifies each shoe
BrandName	varchar(30)		Brand name of the shoe
Category	varchar(30)		Category of the shoe, such as for example, sports shoe, casual wear, party wear, and so forth
UnitPrice	money		Price of the shoe in dollars
QtyOnHand	int		Quantity available

Table 10.6: Shoes Table

- b) Add at least five records to the table. Ensure that the value of the column **QtyOnHand** is more than 20 for each of the shoes.
 - c) Write statements to create a stored procedure named **sp_PriceIncrease** that will increment the **unitprice** of all shoes by 10 dollars.
 - d) Write statements to create a stored procedure **sp_QtyOnHand** that will decrease the quantity on hand of specified brands by 25. The brand name should be supplied as input.
 - e) Execute the stored procedures **sp_PriceIncrease** and **sp_QtyOnHand**.
3. Create a view called **dbo.vw_OrderDetails** that displays details of sales orders for the product whose id is 777. Use the **Sales.SalesOrderDetail** table of **AdventureWorks2019** database. Test the view by creating a query that retrieves data from the view.
 4. Create a view called **dbo.vw_Products** that displays a list of the products from the **Production.Product** table joined to the **Production.ProductCostHistory** table of **AdventureWorks2019** database. Include columns that describe the product and show the cost history for each product. Test the view by creating a query that retrieves data from the view.

Session - 11

Indexes

Welcome to the Session, **Indexes**. This session introduces indexes with their internal data storage. Then, this session explains index structures for different indexes along with their types. Finally, the session describes the procedures for Index management to implement types of indexes in actual environment.

In this session, you will learn to:

- Define and explain Indexes
- Explain Storage Structure
- Explain types of Indexes
- Understand Index Management

11.1 Introduction

Indexes are special data structures associated with tables or views that help speed up the query. Table 11.1 lists commonly used indexes in SQL Server.

Index Type	Description
Clustered	It sorts and stores the data rows of a table or view in order based on the clustered index key. Clustered index is implemented as a B-tree index structure that supports fast retrieval of the rows, based on their clustered index key values.
Nonclustered	Non-clustered index is defined on a table or view that has data in either a clustered structure or on a heap. Each index row in the non-clustered index contains nonclustered key value and a row locator. Locator points to data row in the clustered index or heap having key value. Rows in index are stored in order of the index key values, but the data rows are not guaranteed to be in any particular order unless a clustered index is created on the table.
Unique	Unique index ensures that index key contains no duplicate values and therefore, each row in the table or view is in some way unique. Uniqueness can be a property of both clustered and nonclustered indexes.
Columnstore	Columnstore index stores and manages data by using column-based data storage and column-based query processing in in-memory. Columnstore indexes work well for data warehousing workloads that primarily perform bulk loads and read-only queries. Use the columnstore index to achieve up to 10x query performance gains over traditional row-oriented storage, and up to 7x data compression over the uncompressed data size.
Filtered	Optimized non-clustered index is suited to cover queries that select from a well-defined subset of data. It uses a filter predicate to index a portion of rows in a table. A well-designed filtered index can improve query performance, reduce index maintenance

Index Type	Description
	costs, and reduce index storage costs compared with full-table indexes.
Spatial	It provides the ability to perform certain operations more efficiently on spatial objects in a column of geometry data type.
XML	Due to large size of XML columns, queries that search within these columns can be slow. You can speed up these queries by creating an XML index on each column. An XML index can be a clustered or a nonclustered index.

Table 11.1: Types of Indexes

There are other types of indexes such as Hash, Memory optimize nonclustered, Index with included column, Index on computed columns, and Full text.

SQL Server engine uses an index in the similar way as a student uses a book index. For example, consider that you need to find all references to `INSERT` statements in an SQL book. The immediate approach taken will be to scan each page of the book beginning from the starting page. You mark each time the word `INSERT` is found, until the end of the book is reached. This approach is time consuming and laborious. The second way is to use the index in the back of the book to find the page numbers for each occurrence of the `INSERT` statements. The second way produces the same results as the first, but by tremendously saving time.

When SQL Server has not defined any index for searching, then the process is similar to the first way in the example; the SQL engine must visit each row in a table. In database terminology, this behavior is called table scan or just scan.

A table scan is not always troublesome, but it is sometimes unavoidable. However, as a table grows up to thousands and millions of rows and beyond, scans become slower and more expensive. In such cases, indexes are strongly recommended.

11.1.1 Overview of Data Storage

A book contains pages, which contain paragraphs made up of sentences. Similarly, SQL Server stores data in storage units known as data pages. These pages contain data in the form of rows.

The pages in a book have the same physical size. Similarly, in SQL Server all data pages are the same size - 8 kilobytes (KB). This means that SQL Server databases have 128 data pages per megabytes (MB) of storage space.

A page begins with a 96-byte header, which stores system information about the page. This information includes the following:

- Page number
- Page type
- Amount of free space on the page
- Allocation unit ID of the object to which the page is allocated

Figure 11.1 shows data storage structure of a data page.

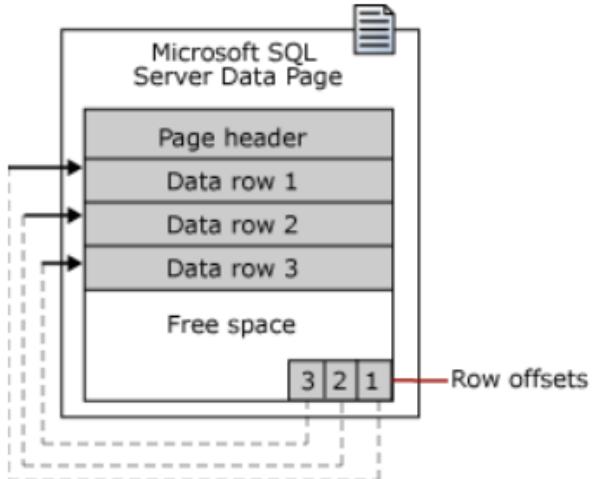


Figure 11.1: Data Storage

Note - A data page is the smallest unit of data storage. An allocation unit is a collection of data pages grouped together based on the page type. This grouping is done for efficient management of data.

11.1.2 Data Files

All input and output operations in the database are performed at the page level. This means that the database engine reads or writes data pages. A set of eight contiguous data pages is referred to as an extent.

SQL Server stores data pages in files known as data files. The space allotted to a data file is divided into sequentially numbered data pages. The numbering starts from zero as shown in figure 11.2.

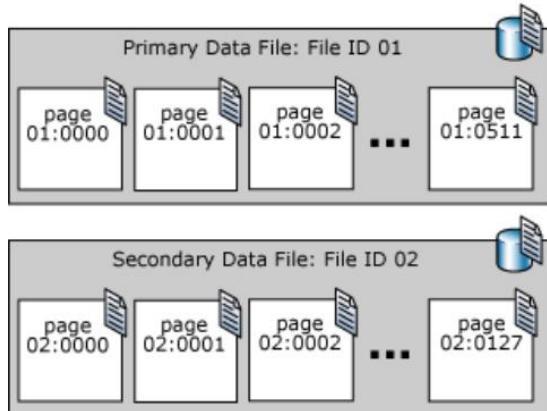


Figure 11.2: Data Files

There are three different types of data files explained as follows:

Primary

- A primary data file is automatically created at the time of creation of the database. This file has references to all other files in the database. The recommended file extension for primary data files is **.mdf**.

Secondary

- These are optional user-defined data files. Data can be spread across multiple disks by putting each file on a different disk drive. Recommended file name extension for secondary data files is **.ndf**.

Transaction Log

- Log files contain information about modifications carried out in the database. This information is useful in recovery of data in contingencies such as sudden power failure or the need to shift the database to a different server. There is at least one log file for each database. The recommended file extension for log files is **.ldf**.

11.1.3 Requirement for Indexes

To facilitate quick retrieval of data from a database, SQL Server provides the indexing feature. Similar to an index in a book, an index in SQL Server database contains information that allows you to find specific data without scanning through the entire table as shown in figure 11.3.

Index			
A			
Adapter	1	Border	19
Aggregate	10	Bullet	58
Analysis	13		
Average	23	C	
		Consistency	20
B		Connect	22
Board	17	Communication	24
Brilliant	18	Character	30

Figure 11.3: Index in a Book

11.1.4 Indexes

In a table, records are stored in the order in which they are inserted. Their storage in the database is unsorted. When data is to be retrieved from such tables, the entire table requires to be scanned. This slows down the retrieval process. To speed up data retrieval process, indexes are required.

When an index is created on a table, it creates an order for the data rows or records in the table as shown in figure 11.4. This assists in faster location and retrieval of data during searches.

Index		Employee_Details		
	EmployeeID	EmployeeID	EmployeeName	DepartmentID
	CN00012	CN00016	John Keena	Purchase
	CN00015	CN00015	Smith Jones	Accounts
	CN00016	CN00020	Albert Walker	Sales
	CN00020	CN00012	Rosa Stines	Administrator

Figure 11.4: Indexes

Note - Multiple indexes can be created on a single table.

Indexes are automatically created when PRIMARY KEY and UNIQUE constraints are defined on a table. Indexes reduce disk I/O operations and consume fewer system resources.

The CREATE INDEX statement is used to create an index. Following is the syntax for this statement:

Syntax

```
CREATE INDEX <index_name> ON <table_name> (<column_name>)
```

where,

- <index_name>: specifies the name of the index.
- <table_name>: specifies the name of the table.
- <column_name>: specifies the name of the column.

Indexes point to the location of a row on a data page instead of searching through the table. Consider the following facts and guidelines about indexes:

- Indexes increase the speed of queries that join tables or perform sorting operations.
- Indexes implement the uniqueness of rows if defined when you create an index.
- Indexes are created and maintained in ascending or descending order.

11.1.5 Scenario

In a telephone directory, where a large amount of data is stored and is frequently accessed, the storage of data is done in an alphabetical order. If such data were unsorted, it would be nearly impossible to search for a specific telephone number.

Similarly, in a database table having a large number of records that are frequently accessed, the data is to be sorted for fast retrieval. When an index is created on the table, the index either physically or logically sorts the records. Thus, searching for a specific record becomes faster and there is less strain on system resources.

11.1.6 Accessing Data Group-wise

Indexes are useful when data must be accessed group-wise. For example, you want to make modifications to the conveyance allowance for all employees based on the department they work in.

Here, you wish to make changes for all employees in one department before moving on to employees in another department. In this case, an index can be created as shown in figure 11.5 on the **DepartmentName** column before accessing records.

This index will create logical chunks of data rows based on the department. This again will limit the amount of data actually scanned during query retrieval.

Hence, retrieval will be faster and there will be less strain on system resources.

Department Name	Employee Name
Marketing	Jenny Woods
Marketing	Merry Thomas
Marketing	John Updeeke
Marketing	Robert Williamson
Sales	Smith Gordon
Sales	Albert Wang

Figure 11.5: Accessing Data Group-wise

11.2 Index Architecture

In SQL Server, data in the database can be stored either in a sorted manner or at random. If data is stored in a sorted manner, the data is said to be present in a clustered structure. If it is stored at random, it is said to be present in a heap structure.

Figure 11.6 shows an example demonstrating index architecture.

Employee_Details		
EmpID	EmpName	DeptID
CN00020	Rosa Stevens	BN0001
CN00018	John Updeeke	BN0020
CN00019	Smith Gordon	BN0021
CN00012	Robert Tyson	BN0011

Heap Structure

Employee_Details		
EmpID	EmpName	DeptID
CN00012	Robert Tyson	BN0011
CN00018	John Updeeke	BN0020
CN00019	Smith Gordon	BN0021
CN00020	Rosa Stevens	BN0001

Clustered Structure

Figure 11.6: Index Architecture

11.2.1 B-Tree

In SQL Server, indexes are organized as B-Trees. Each page in an index B-tree is called an index node. Top node of the B-tree is called the root node. Bottom nodes in the index are called the leaf nodes. Any index levels between the root and the leaf nodes are collectively known as intermediate nodes as shown in figure 11.7.

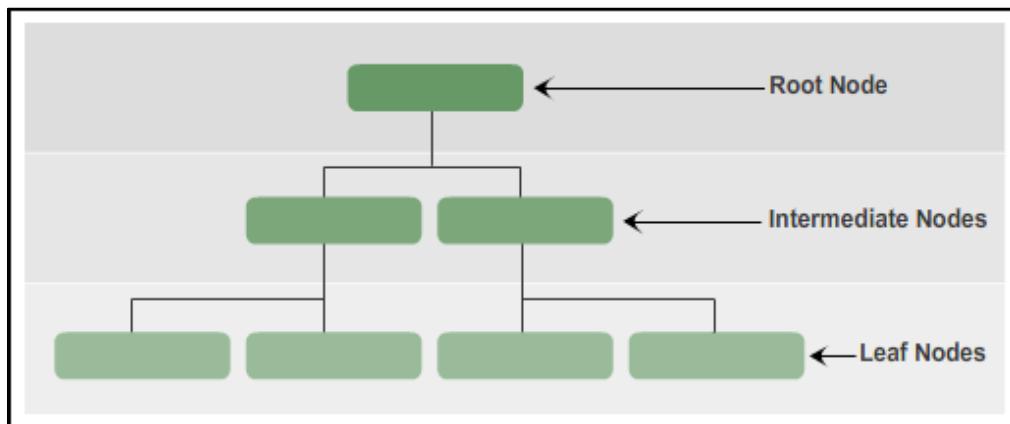


Figure 11.7: B-Tree

Note - A B-Tree index traverses from top to bottom by using pointers.

11.2.2 Index B-Tree Structure

In the B-Tree structure of an index, the root node consists of an index page. This index page contains pointers that point to the index pages present in the first intermediate level. These index pages in turn point to the index pages present in the next intermediate level. There can be multiple intermediate levels in an index B-Tree. The leaf nodes of the index B-Tree have either data pages containing data rows or index pages containing index rows that point to data rows as shown in figure 11.8.

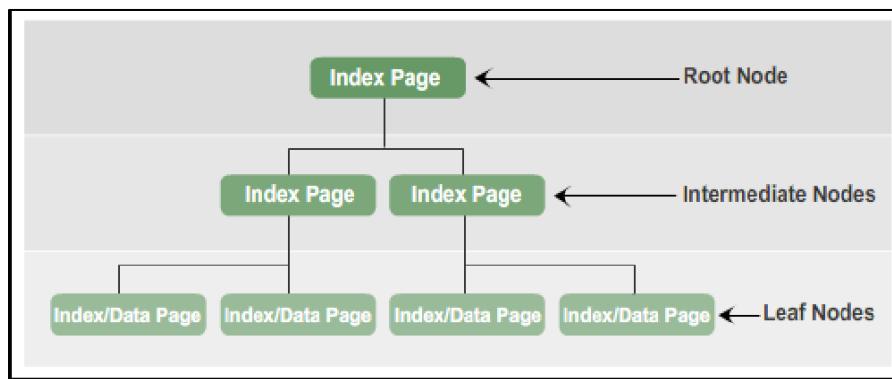


Figure 11.8: Index B-Tree Structure

Different types of nodes are as follows:

- **Root Node** - Contains an index page with pointers pointing to index pages at the first intermediate level.
- **Intermediate Nodes** - Contain index pages with pointers pointing either to index pages at the next intermediate level or to index or data pages at the leaf level.
- **Leaf Nodes** - Contain either data pages or index pages that point to data pages.

Note - A data page containing index entries is called an index page.

11.2.3 Heap Structures

A heap is a table without a clustered index. This means that, in a heap structure, the data pages and records are not arranged in sorted order. The only connection between the data pages is the information recorded in the Index Allocation Map (IAM) pages.

Heaps have one row in `sys.partitions`, with `index_id = 0` for each partition used by the heap. By default, a heap has a single partition. When a heap has multiple partitions, each partition has a heap structure that contains data for that specific partition as shown in figure 11.9. For example, if a heap has four partitions, there are four heap structures; one in each partition.

At a minimum, each heap will have one `IN_ROW_DATA` allocation unit per partition. The heap will also have one `LOB_DATA` allocation unit per partition, if it contains large object (LOB) columns. It will also have one `ROW_OVERFLOW_DATA` allocation unit per partition, if it contains variable length columns that exceed the 8,060 byte row size limit.

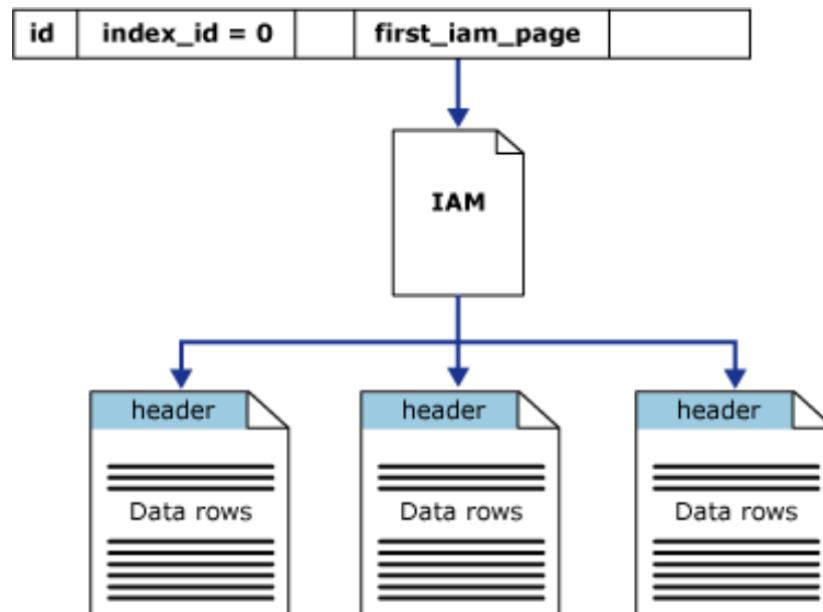


Figure 11.9: Heap Structures

Note - If an allocation unit contains extents from more than one file, there will be multiple IAM pages linked together in an IAM chain to map these extents.

11.2.4 Clustered Index Structures

Clustered indexes are organized as B-Trees. Each page in an index B-tree is called an index node.

Top node of the B-tree is root node and bottom nodes in the index are called the leaf nodes.

Clustered Index

- Leaf nodes contain data pages of the underlying table, root, and intermediate level nodes contain index pages holding index rows.
- Each index row contains a key value and a pointer to either an intermediate level page in the B-tree or a data row in the leaf level of the index.
- By default, a clustered index has a single partition. When a clustered index has multiple partitions, each partition has a B-tree structure that contains the data for that specific partition.
- The clustered index will also have one *LOB_DATA* allocation unit per partition if it contains large object (LOB) columns. It will also have one *ROW_OVERFLOW_DATA* allocation unit per partition if it contains variable length columns that exceed the 8,060 byte row size limit.

Figure 11.10 shows the structure of a clustered index in a single partition.

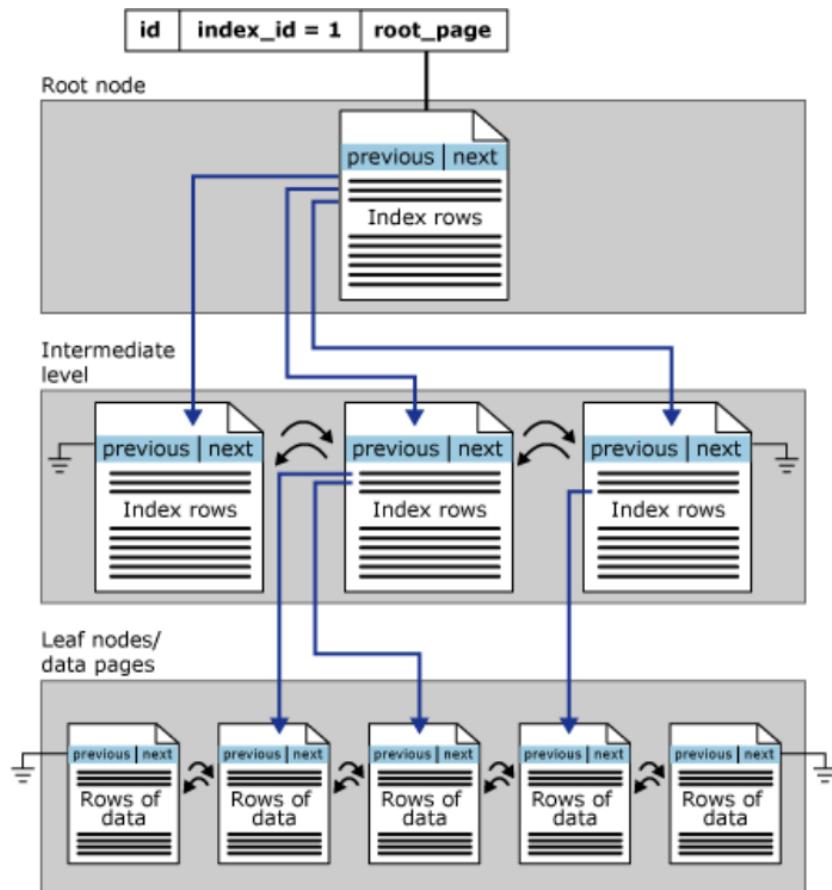


Figure 11.10: Clustered Indexes

11.2.5 Nonclustered Index Structures

A nonclustered index is defined on a table that has data in either a clustered structure or a heap.

Nonclustered index will be the default type if an index is not defined on a table. Each index row in the nonclustered index contains a nonclustered key value and a row locator. This row locator points to the data row corresponding to the key value in the table. Figure 11.11 shows nonclustered index structure.

Nonclustered indexes have a similar B-Tree structure as clustered indexes, but with the following differences:

- The data rows of the table are not physically stored in the order defined by their nonclustered keys.
- In a nonclustered index structure, the leaf level contains index rows.

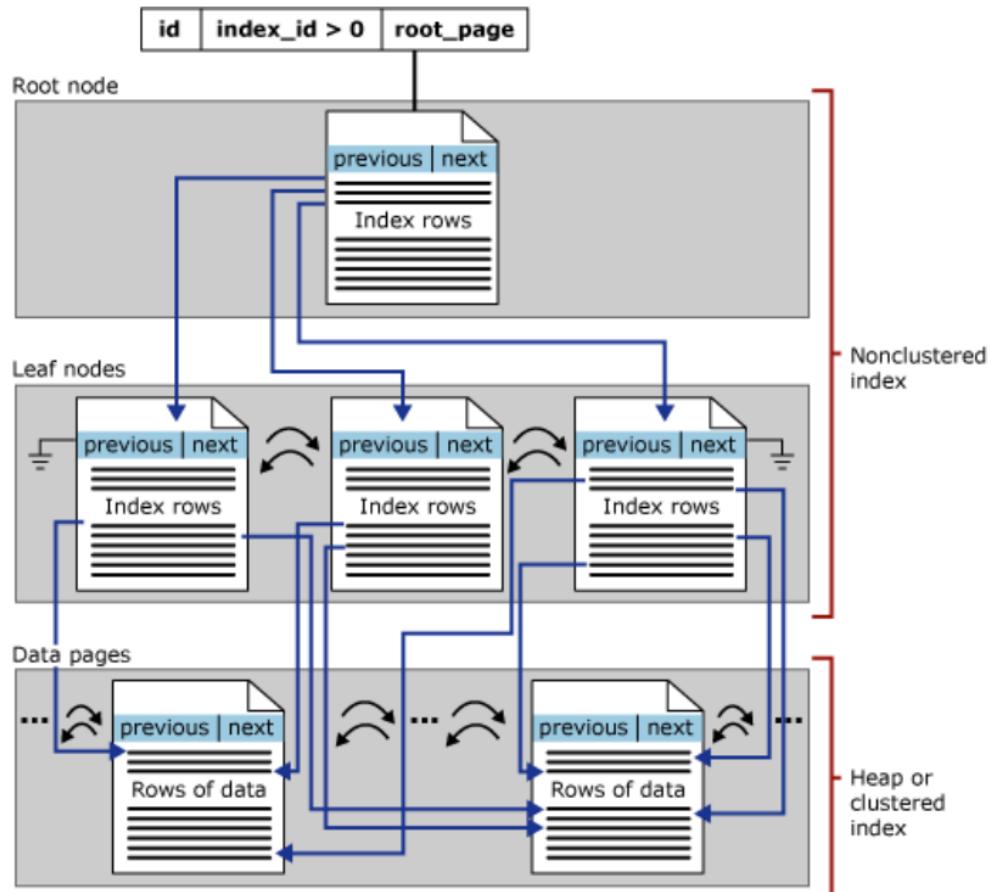


Figure 11.11: Nonclustered Index

- Nonclustered indexes are useful when you require multiple ways to search data. Some facts and guidelines to be considered before creating a nonclustered index are as follows:
- When a clustered index is re-created or the `DROP_EXISTING` option is used, SQL Server rebuilds the existing nonclustered indexes.
- A table can have up to 999 nonclustered indexes.
- Create clustered index before creating a nonclustered index.

11.2.6 Column Store Index

A columnstore index is a feature in SQL Server for storing, retrieving and managing data by using a **columnar** data format, called a columnstore as shown in figure 11.12.

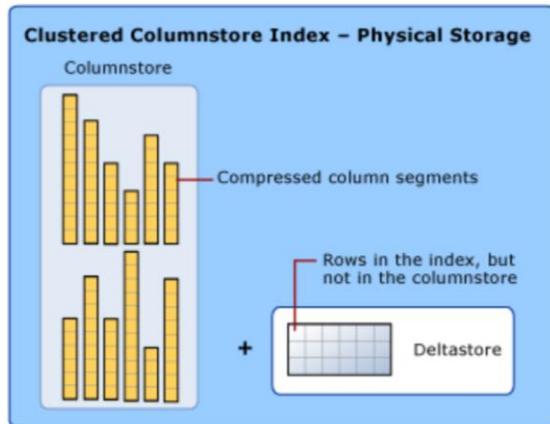


Figure 11.12: Clustered Column Store Index

Columnstore indexes use both types of data storage format rowstore and columnstore.

Columnstore indexes are mainly used for following reasons:

- To reduce storage costs
- Better performance

Details of columnstore, rowstore, and deltastore data storage formats are as follows:

Columnstore	It is logically organized data as a table with rows and columns and physically stored in a column-wise data format.
Rowstore	It is logically organized data as a table with rows and columns and then, physically stored in a row-wise data format.
Deltastore	It is a holding place for rows that are too few in number to be compressed into the columnstore. The deltastore stores the rows in rowstore format.

11.2.7 Hash Index

Hash index consists of an array of pointers and each element of the array is called a hash bucket.

- Each bucket is eight bytes, which are used to store the memory address of a link list of key entries.
- Each entry is a value for an index key, plus the address of its corresponding row in the underlying memory-optimized table.
- Each entry points to the next entry in a link list of entries, all chained to the current bucket.

The number of buckets must be specified at index definition time:

- The lower the ratio of buckets to table rows or to distinct values, the longer the average bucket link list will be.
- Short link lists perform faster than long link lists.

- The maximum number of buckets in hash indexes is 1,073,741,824.

Interplay of the hash index and the buckets is summarized in the figure 11.13.

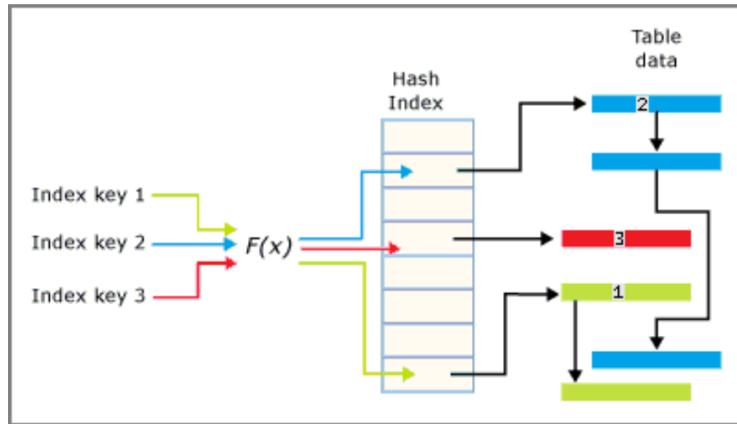


Figure 11.13: Hash Index Architecture

11.2.8 XML Indexes

XML indexes can be created on xml data type columns. They index all tags, values, and paths over the XML instances in the column and benefit query performance. Your application may benefit from an XML index in the following situations:

Queries on XML columns are common in your workload. XML index maintenance cost during data modification must be considered.

When XML values are relatively large and the retrieved parts are relatively small. Building the index avoids parsing the whole data at run time and benefits index lookups for efficient query processing.

XML indexes fall into the following categories:

- Primary XML index
- Secondary XML index

11.2.9 Spatial Indexes

In SQL Server, spatial indexes are built using B-trees, which means that the indexes must represent the 2-dimensional spatial data in the linear order of B-trees. Therefore, before reading data into a spatial index, SQL Server implements a hierarchical uniform decomposition of space. The index-creation process decomposes the space into a four-level grid hierarchy. These levels are referred to as level 1 (the top level), level 2, level 3, and level 4 as shown in figure 11.16.

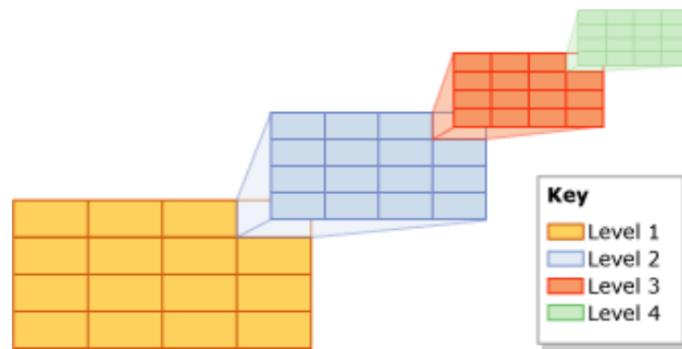


Figure 11.16: Four Levels of 4x4 Grids

11.2.10 Full-Text Indexes

Creating and maintaining a full-text index involves populating the index by using a process called a population also known as a crawl.

Types of population

A full-text index supports the following types of population:

- Full population
- Automatic or manual population based on change tracking
- Incremental population based on a timestamp

11.3 Index Management

Index management allow user to manage indexes in terms various operation such as CREATE, ALTER, DROP, and so on.

11.3.1 Create Clustered Index

CREATE CLUSTERED index statement allow users to create CLUSTERED index on specified columns and table.

Syntax

```
CREATE CLUSTERED INDEX index_name ON table_name (column1, column2, ...);
```

For creating clustered index, a new table name parts is created in AdventureWorks2019 database under production with column details as shown in Code Snippet 1.

Code Snippet 1:

```
CREATE TABLE Production.parts(
    part_id    INT NOT NULL,
    part_name  VARCHAR(100)
```

Code Snippet 2 displays how create clustered index named **ix_parts_id** on **Part_Id** column in **Production.parts** table.

Code Snippet 2:

```
CREATE CLUSTERED INDEX ix_parts_id ON Production.parts (Part_id);
```

The index is created as shown in figure 11.17.

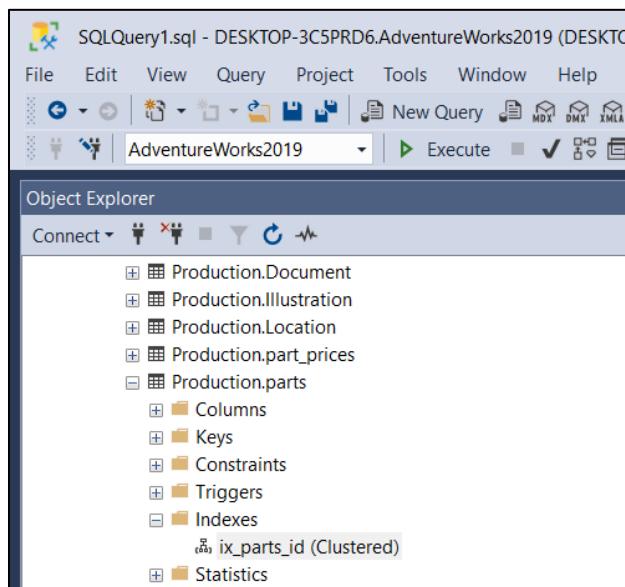


Figure 11.17: Clustered Index

In a similar way, following syntaxes are used to Rename, Disable, Enable, and Drop indexes:

RENAME INDEX

The **sp_rename** is a system stored procedure that allows you to rename any user-created object in the current database including table, index, and column.

Syntax

```
EXEC sp_rename
    index_name,
    new_index_name,
    N'INDEX';
```

Code Snippet 3 renames the index **ix_parts_id** of the **production.parts** table to **index_part_id** and output is shown in figure 11.18.

Code Snippet 3:

```
EXEC sp_rename
    N'production.parts.ix_parts_id',
    N'index_part_id',
    N'INDEX';
```

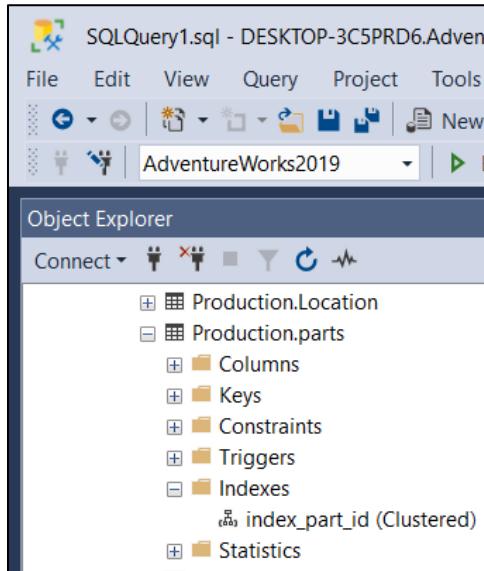


Figure 11.18: Renamed Index

Note: One can rename an index using the SQL Server Management Studio (SSMS), simply by right-clicking index name it can be done.

DISABLE INDEX

To disable an index, `ALTER INDEX` statement is used as follows:

Syntax

```
ALTER INDEX index_name  
ON table_name  
DISABLE;
```

Code Snippet 4 displays example for disable index. It disables `index_part_id` on `part_id` column in `production.parts` table.

Code Snippet 4:

```
ALTER INDEX INDEX_PART_ID  
ON Production.Parts  
DISABLE;
```

Figure 11.19 displays `index_part_id` is disabled successfully.

```
SQLQuery2.sql - DE...D6\Lenovo pc (51)* X
Use AdventureWorks2019;
ALTER INDEX index_part_id
ON production.parts
DISABLE;

100 % < >
Messages
Commands completed successfully.

Completion time: 2020-11-13T12:59:05.9584627+05:30
```

Figure 11.19: Disable Index

As a result of this, simply if user tried to display all records from Production.Parts table, it displays error as shown in following figure 11.20.

```
SQLQuery1.sql - DE...D6\Lenovo pc (52)* X
SELECT * FROM production.parts;

100 % < >
Messages
Msg 8655, Level 16, State 1, Line 12
The query processor is unable to produce a plan because the index 'index_part_id' on table or view 'parts' is disabled.

Completion time: 2020-10-20T14:56:42.5836922+05:30
```

Figure 11.20: Records of Production.Parts

In addition to this, user can able to disable all indexes on table as shown in Code Snippet 5.

Code Snippet 5:

```
ALTER INDEX ALL ON Production.Parts
DISABLE;
```

ENABLE INDEX

This statement uses the ALTER INDEX statement to ‘enable’ or rebuild an index on a table.

Syntax

```
ALTER INDEX index_name
ON table_name
REBUILD;
```

Figure 11.21 displays example for enabling index. It enables INDEX_PART_ID on PART_ID column on Production.Parts table.

The screenshot shows a SQL query window titled "SQLQuery1.sql - DE...D6\Lenovo pc (52)*". The query is:

```
ALTER INDEX index_part_id  
ON production.parts  
REBUILD;
```

The status bar at the bottom indicates "100 %". Below the query results, there is a "Messages" section with the message "Commands completed successfully." and a completion time of "Completion time: 2020-10-20T15:19:12.9363457+05:30".

Figure 11.21: Enable Index

Similarly, user can enable all indexes for `production.parts` table as shown in figure 11.22.

DROP INDEX

`DROP INDEX` statement removes one or more indexes from the current database. Following is the syntax for `DROP INDEX` statement:

Syntax

```
DROP INDEX [IF EXISTS] index_name  
ON table_name;
```

- Specify name of index that you want to remove after `DROP INDEX` clause.
- Then, specify name of table to which index belongs.

Code Snippet 6 shows code for dropping index `INDEX_PART_ID` on `Production.Parts` table.

Code Snippet 6:

```
DROP INDEX IF EXISTS  
INDEX_PART_ID ON Production.Parts;
```

Figure 11.22 displays output of Code Snippet 6.

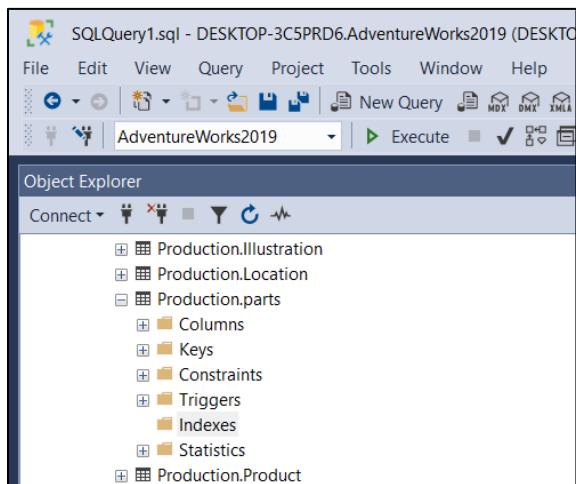


Figure 11.22: Drop Indexes

Removing a nonexisting index will result in an error. However, you can use the `IF EXISTS` option to conditionally drop the index and avoid the error as shown in figure 11.23.

```
SQLQuery1.sql - DE...D6\Lenovo pc (52)* + X
DROP INDEX
index_part_id ON production.parts;

100 % < >
Messages
Msg 3701, Level 11, State 7, Line 24
Cannot drop the index 'production.parts.index_part_id', because it does not exist or you do not have permission.

Completion time: 2020-10-20T16:45:32.9073678+05:30
```

Figure 11.23: Nonexisting index

11.3.2 NonClustered Indexes

A nonclustered index is a data structure that improves speed of data retrieval from tables. Unlike a clustered index, a nonclustered index sorts and stores data separately from the data rows in table.

Following syntax is used to create nonclustered index:

Syntax

```
CREATE [NONCLUSTERED] INDEX index_name
ON table_name(column_list);
```

- First, specify name of index followed by `CREATE NONCLUSTERED INDEX` clause. Note that the `NONCLUSTERED` keyword is optional.

- Then, specify the table name on which you want to create index and a list of columns of that table as the index key columns.

Code Snippet 7 displays how to create nonclustered index on `Sales.Customer` table from `Adventureworks2019` database.

Code Snippet 7:

```
CREATE NONCLUSTERED INDEX INDEX_CUSTOMER_STOREID
ON Sales.Customer(StoreID);
```

Figure 11.24, displays `index_customer_storeid` is created on `Sales.Customer` table.

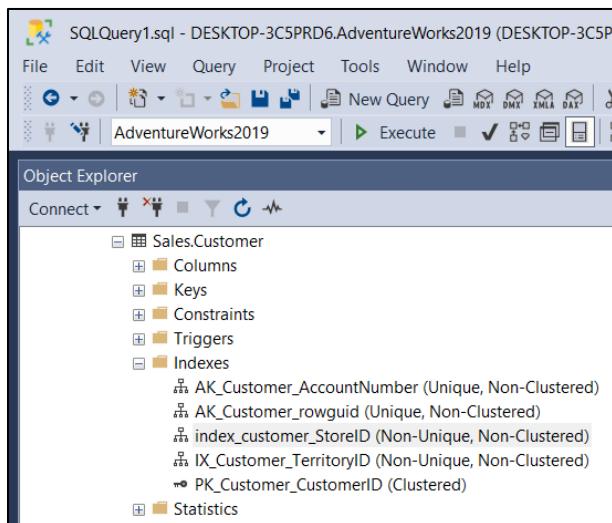


Figure 11.24: Nonclustered Index

11.3.3 Unique Indexes

Unique index ensures index key columns do not contain any duplicate values.

It may consist of one or many columns. If a unique index has one column, the values in this column will be unique. In case the unique index has multiple columns, the combination of values in these columns is unique.

Note: unique index can be clustered or non-clustered.

Following syntax can be used to create a unique index:

Syntax

```
CREATE UNIQUE INDEX index_name
ON table_name(column_list);
```

- Provide name of unique index after `CREATE UNIQUE INDEX` keywords.

- Then, specify name of table to which index associated and a list of columns that will be included in index.

Code Snippet 8, display how to create unique index on Sales.Customer table in AdventureWorks2019 database.

Code Snippet 8:

```
CREATE UNIQUE INDEX AK_Customer_rowguid
ON Sales.Customer(rowguid);
```

Figure 11.25 displays Unique index created in Code Snippet 8.

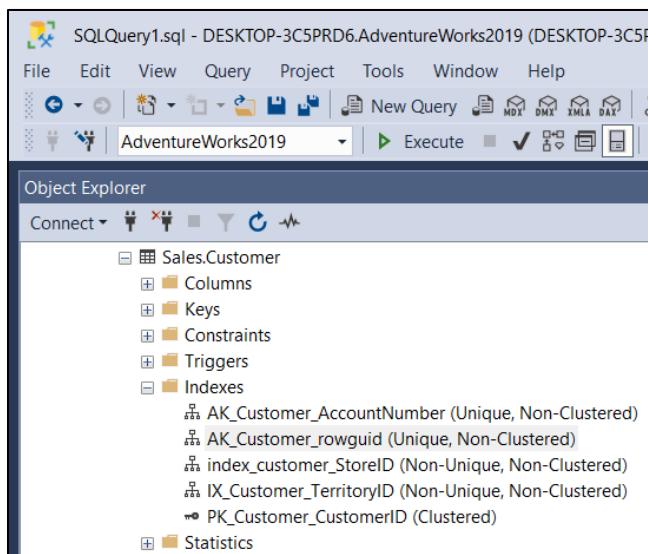


Figure 11.25: Unique Index

11.3.4 Filtered Indexes

Filtered index is a nonclustered index with a predicate that allows you to specify which rows should be added to the index.

Following syntax illustrates how to create a filtered index:

Syntax

```
CREATE INDEX index_name
ON table_name(column_list)
WHERE predicate;
```

- Provide name of filtered index after CREATE INDEX clause.
- Next, list the table name with a list of key columns that will be included in index.
- Then, use a WHERE clause with a predicate to specify which rows of the table should be included in index.

For creating filtered index Sales.Customer table from AdventureWorks2019 database is used.

In this table column `PersonID` having so many `NULL` values and it will be correct selection for filtered index. Let us understand this by using Code Snippet 9.

Code Snippet 9:

```
CREATE INDEX index_cust_personID  
ON Sales.Customer(PersonID)  
WHERE PersonID IS NOT NULL;
```

Figure 11.26 displays filtered index created using Code Snippet 9.

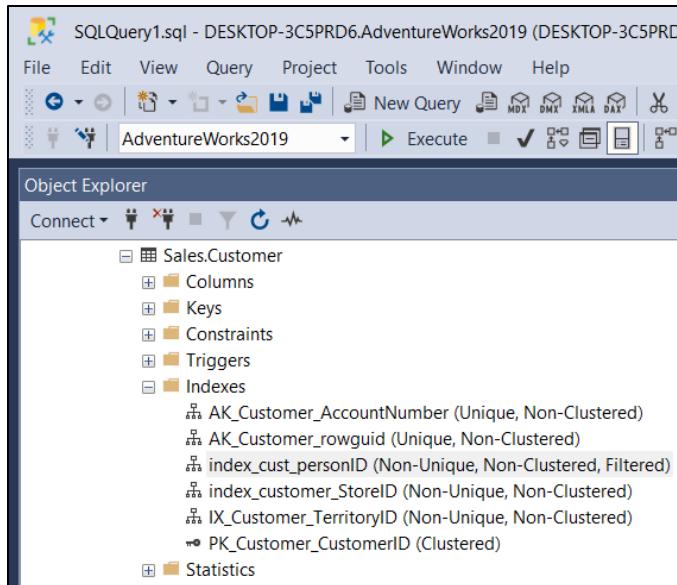


Figure 11.26: Filtered Index

Now, suppose user executed code from Code Snippet 10.

Code Snippet 10:

```
Select CustomerID, PersonID, StoreID from Sales.Customer Where PersonID=1700;
```

Here is the estimated execution plan as shown in figure 11.27.

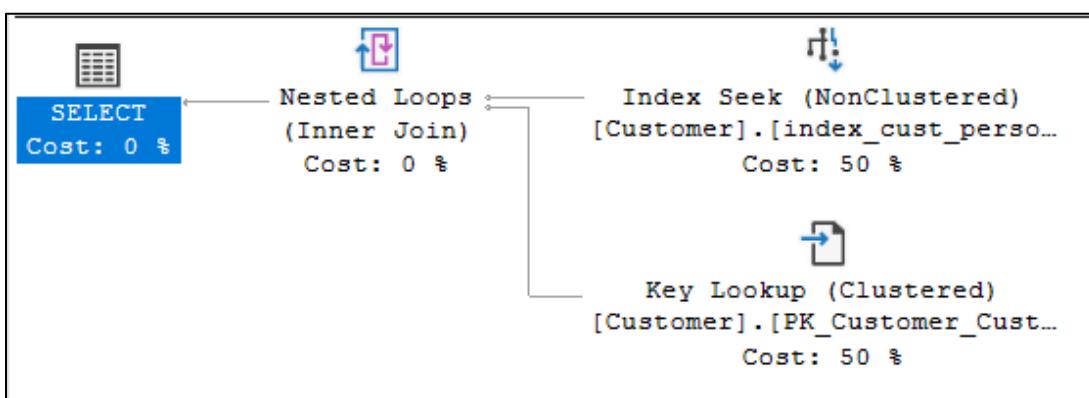


Figure 11.27: Execution Plan for index_cust_personID

In figure 11.27, if you noticed query optimizer leverage filtered index `index_cust_personID` for searching.

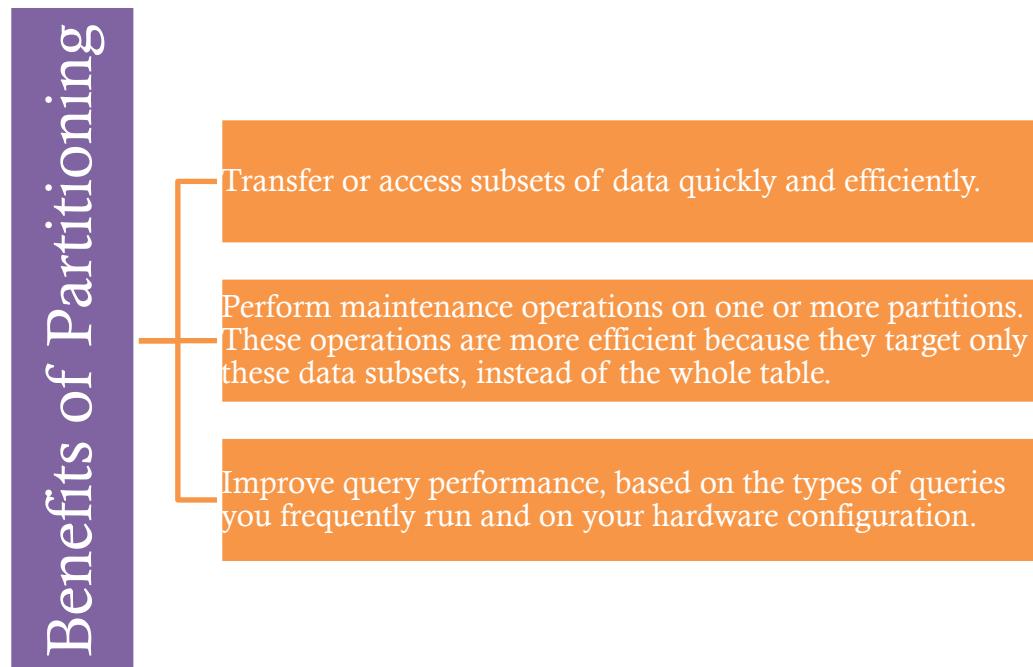
Note: To improve the key lookup, you can use an index with included columns.

11.3.5 Partitioned Table and Indexes

SQL Server supports both table and index partitioning. Data of partitioned tables and indexes is divided into units that may optionally be spread across more than one filegroup in a database. Data is partitioned horizontally, so that groups of rows are mapped into individual partitions. All partitions of a single index or table must reside in the same database. Table or index is treated as a single logical entity when queries or updates are performed on the data.

Note: SQL Server 2019 (15.x) supports up to 15,000 partitions by default. In versions earlier than SQL Server 2012 (11.x), the number of partitions was limited to 1,000 by default.

Partitioning large tables or indexes can have the following manageability and performance benefits.



For creating sample partition following table with mentioned detail is used.

```
CREATE TABLE testing_table(receipt_id BIGINT, date DATE);
```

Specify exactly how table is going to be partitioned by partitioning column. In this case, `date`, along with the range of values included in each partition. Regarding partition boundaries, you can specify either the `LEFT` or `RIGHT` as displayed in Code Snippet 11.

Code Snippet 11:

```
CREATE PARTITION FUNCTION partition_function (int)
RANGE LEFT FOR VALUES (20200630, 20200731, 20200831);
```

In this example, following four partitions are defined:

- Partition No. 1 ⇒ All records with date \leq 2020-06-30
- Partition No. 2 ⇒ All records with date $>$ 2020-06-30 & date \leq 2020-07-31
- Partition No. 3 ⇒ All records with date $>$ 2020-07-31 & date \leq 2020-08-31
- Partition No. 4 ⇒ All records with date $>$ 2020-08-31

Code Snippet 12 will allow to determine partition at which records are placed.

Code Snippet 12:

```
(SELECT 20200613 date, $PARTITION.partition_function(20200613)
PartitionNumber)
UNION
(SELECT 20200713 date, $PARTITION.partition_function(20200713)
PartitionNumber)
UNION
(SELECT 20200813 date, $PARTITION.partition_function(20200813)
PartitionNumber)
UNION
(SELECT 20200913 date, $PARTITION.partition_function(20200913));
```

Output of Code Snippet 12 is displayed in figure 11.28.

The screenshot shows a SQL query window titled "SQLQuery1.sql - DE...D6\Lenovo pc (63)*". The code creates a table "testing_table" and defines a partition function "partition_function" with four ranges. Below the code, a results grid displays the mapping of dates to partition numbers:

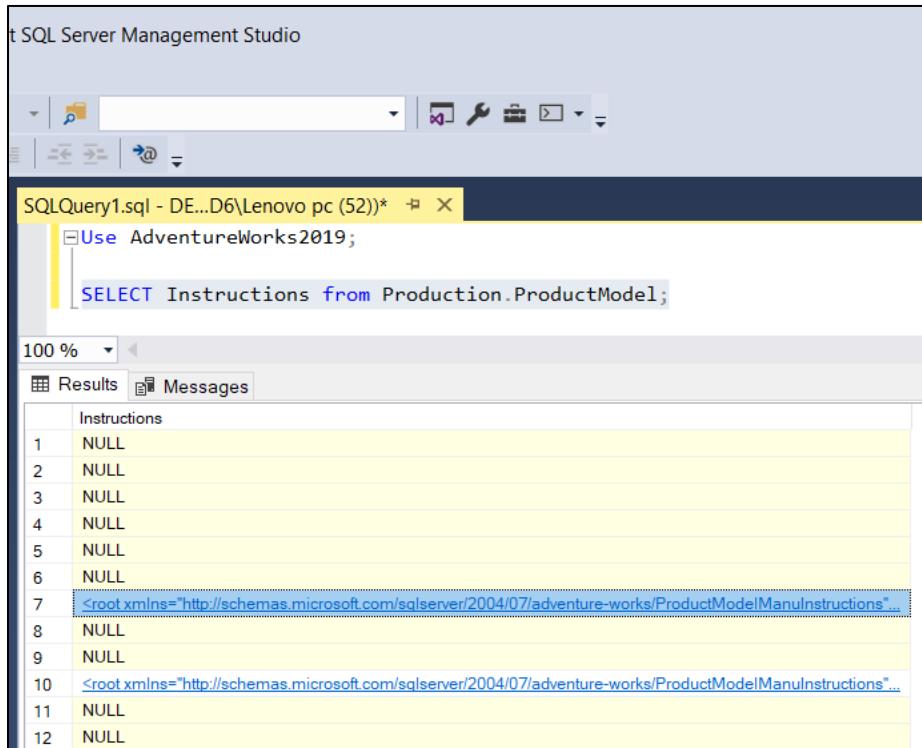
	date	PartitionNumber
1	20200613	1
2	20200713	2
3	20200813	3
4	20200913	4

Figure 11.28: Partition Table

11.3.6 XML Indexes

XML data is stored in xml type columns as large binary objects (BLOBs). These XML instances can be large, and the stored binary representation of xml data type instances can be up to two GB.

Following figure 11.29 displays how XML data is stored in Instructions column of Production.ProductModel table in AdventureWorks2019 database.



The screenshot shows the SQL Server Management Studio interface. A query window titled 'SQLQuery1.sql - DE...D6\Lenovo pc (52)*' contains the following code:

```
USE AdventureWorks2019;
SELECT Instructions from Production.ProductModel;
```

The results grid shows the 'Instructions' column for 12 rows. Rows 1 through 6 are NULL. Rows 7 and 10 are XML fragments starting with '<root xmlns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/ProductModelManu...' and ending with '...' (ellipsis). Rows 8 and 11 are NULL. Row 9 is also NULL.

	Instructions
1	NULL
2	NULL
3	NULL
4	NULL
5	NULL
6	NULL
7	<root xmlns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/ProductModelManu..."...
8	NULL
9	NULL
10	<root xmlns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/ProductModelManu..."...
11	NULL
12	NULL

Figure 11.29: XML Data

XML indexes are created on such XML data columns stored in tables and databases. Code Snippet 13 is used to create Primary XML index name PXML_ProductModel_CatalogDescription on CatalogDescription column of Production.ProductModel column.

Code Snippet 13:

```
CREATE PRIMARY XML INDEX PXML_ProductModel_CatalogDescription  
ON Production.ProductModel (CatalogDescription);
```

Figure 11.30 displays specified XML index in Code Snippet 13 is created.

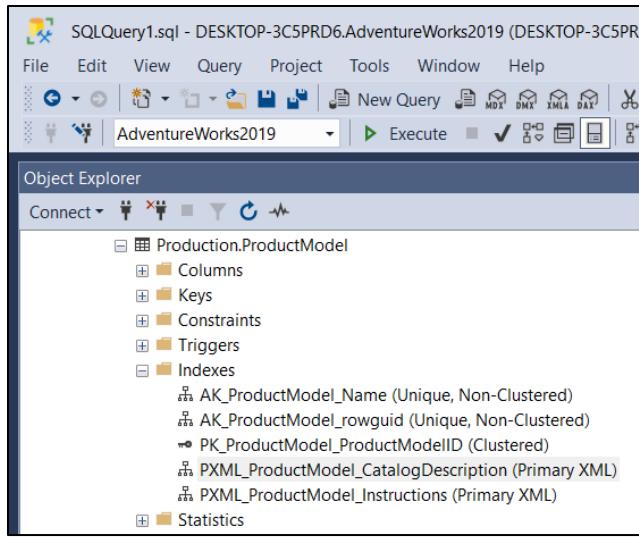


Figure 11.30: Primary XML Index

Primary XML index contains all the data in XML column. To provide an additional performance boost to XML queries, you can add specialized secondary XML indexes. Secondary XML index covers the same data as it is underlying primary index, but it creates a more specific index, based on the primary index as shown in Code Snippet 14.

Code Snippet 14:

```
CREATE XML INDEX IXML_ProductModel_CatalogDescription_Path
ON Production.ProductModel (CatalogDescription)
USING XML INDEX PXML_ProductModel_CatalogDescription
FOR PATH;
```

Figure 11.31, displays specified Secondary XML index in Code Snippet 14 is created.

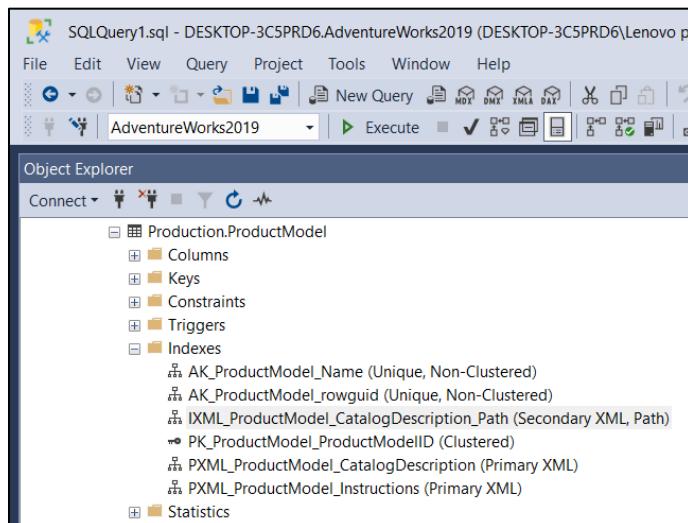


Figure 11.31: Secondary XML Index

11.3.7 Columnstore Indexes

Creating a column store index is done by using the `CREATE COLUMNSTORE INDEX` command and has

many of the same options as a regular index. For creating columnstore index Sales.SalesOrderDetail table from AdventureWorks2019 database will be used as shown in Code Snippet 15.

Code Snippet 15:

```
CREATE COLUMNSTORE INDEX IX_SalesOrderDetail_ProductIDOrderQty_ColumnStore  
ON Sales.SalesOrderDetail (ProductId, OrderQty);
```

Figure shows columnstore index IX_SalesOrderDetail_ProductIDOrderQty_ColumnStore created on Sales.SalesOrderDetail table.

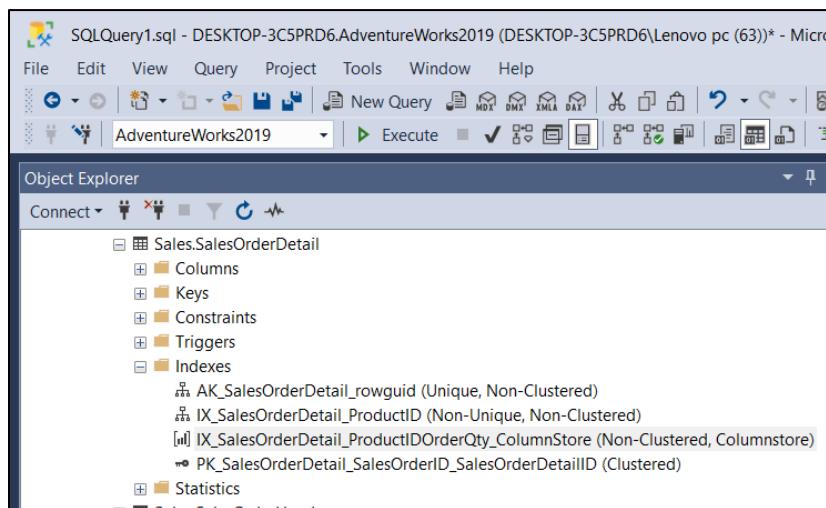


Figure 11.32: Columnstore Index

Once the index is created, a basic aggregation query in Code Snippet 16 is similar to what you would see in any data warehouse environment that is used to test created columnstore index.

Code Snippet 16:

```
SELECT ProductID, SUM(OrderQty)  
FROM Sales.SalesOrderDetail  
GROUP BY ProductId;
```

This query will simply sum of order quantity across each product. By looking at estimated execution plan one can understand that the entire query was satisfied by scanning the column store index and none of the table access was even required as shown in figure 11.33.

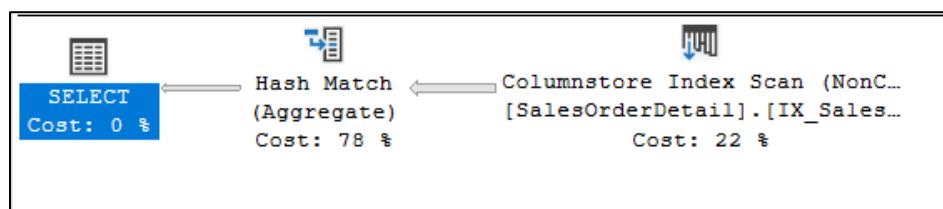


Figure 11.33: Estimated Execution Plan for Columnstore Index

11.4 Check Your Progress

1. Which of the following is the correct code to define an index?

(A)	USE CUST_DB CREATE INDEX IX_Country Customer_Details(Country); GO	(C)	USE CUST_DB CREATE INDEX IX_CountryWithCustomer_Details(Country); GO
(B)	USE CUST_DB CREATE INDEX IX_CountryFROMCustomer_Details(Country); GO	(D)	USE CUST_DB CREATE INDEX IX_Country ON Customer_Details(Country); GO

2. SQL Server stores data in storage units known as _____.

(A)	data pages	(C)	records
(B)	forms	(D)	columns

3. Creating and maintaining a full-text index involves populating the index by using a process called a _____ and is also called as _____.

(A)	Population, Crawl	(C)	Crawl, Trigger
(B)	Troll, Population	(D)	Partition, Population

4. Which of the following code is used to create a clustered INDEX on **CustID** in **Customer_Details** table?

(A)	USE CUST_DB CREATE CLUSTERED INDEX Customer_Details(CustID) GO	(C)	USE CUST_DB CREATE INDEX Customer_Details ON IX_CustID GO
(B)	USE CUST_DB CREATE INDEX IX_CustID ON Customer_Details GO	(D)	USE CUST_DB CREATE CLUSTERED INDEX IX_CustID ON Customer_Details(CustID) GO

5. A clustered index can be created on a table using a column without _____ values.

(A)	similar	(C)	transparent
(B)	duplicate	(D)	any

11.4.1 Answers

1.	D
2.	A
3.	A
4.	D
5.	B



Summary

- Indexes increase the speed of querying process by providing quick access to rows or columns in a data table.
- SQL Server stores data in storage units known as data pages.
- All input and output operations in a database are performed at the page level.
- A clustered index causes records to be physically stored in a sorted or sequential order.
- A nonclustered index is defined on a table that has data either in a clustered structure or a heap.
- XML indexes can speed up queries on tables that have XML data.
- Column Store Index enhances performance of data warehouse queries extensively.

Try It Yourself

1. Houston State Library is one of the renowned libraries in Houston, Texas. The library has a stock of around 10, 00000 books of different genres. The library issues books to students of the college nearby. With the inflow of students coming to the library growing exponentially, Houston State Library has decided to automate the entire process of issuing books to the students. The library has increased the quantity of each book by 10 copies, depending upon the demand made by the students.
- Create a database named **HoustonStateLibrary** to store details of books in the Library.
 - Create a table named **BooksMaster** to store the details of the books in the library as shown in table 11.2.

Field Name	Data Type	Key Field	Description
BookCode	varchar(50)	Primary Key	Stores book code of the book
Title	varchar(max)		Stores the book title
ISBN	varchar(50)		Stores the ISBN of the book
Author	char(30)		Stores author name of the book
Price	money		Stores price of the book
Publisher	char(30)		Stores publisher name of the book
NumPages	numeric(10,0)		Stores number of pages in the book

Table 11.2: BooksMaster Table

- Create a clustered index named IX _ Title on the Title column in the BooksMaster table.
- Create a table BooksMaster1 having field names BookCode, Title, and Book Details.
- Specify the data type for BookDetails as xml. Create an XML document with details of ISBN, Author, Price, Publisher, and NumPages.
- The library wants to retrieve the publisher name of the company which prints a specific author's book.
- Create a primary XML index PXML_Books on the BookCode column of the BooksMaster table.

Session - 12

Triggers

Welcome to the Session, **Triggers**.

This session explains the triggers and different types of triggers. The session also describes the procedure to create and alter DML triggers, nested triggers, update functions, handling multiple rows in a session, LOGON trigger, and performance implication of triggers.

In this Session, you will learn to:

- Explain triggers
- Explain the procedure to create and alter DML triggers
- Describe nested triggers
- Describe update functions
- Explain the handling of multiple rows in a session
- Explain the performance implication of triggers implication

12.1 Introduction

Trigger is a special type of stored procedure that is executed when an attempt is made to modify data in a table for which triggers are created. Unlike standard system stored procedures, triggers cannot be executed directly, nor do they pass or receive parameters.

If `INSERT`, `UPDATE`, or `DELETE` operations are performed on table for which triggers are defined, triggers will get executed automatically when such operations are performed. This automatic execution of the trigger cannot be circumvented. In SQL Server, triggers are created using the `CREATE TRIGGER` statement. Figure 12.1 displays an example of triggers.

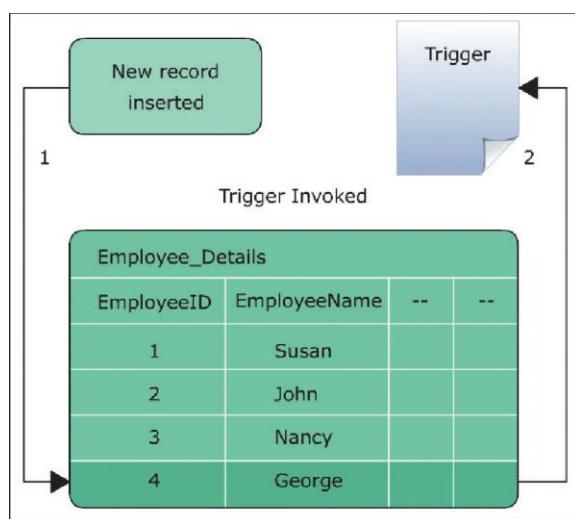


Figure 12.1: Triggers

12.2 Use of Triggers

Triggers can contain complex processing logic and are generally used for maintaining low-level data integrity. The primary uses of triggers can be classified as follows:

➤ Cascade changes in related tables

Users can use a trigger to cascade changes through related tables. For example, consider a table `Sales.Customer` in `AdventureWorks2019` database having a foreign key, `PersonID` referencing the primary key, `BusinessEntityID` of the `Person.Person` table. If an update or a delete event occurs in the `Person.Person` table, an update or delete trigger can be defined to cascade these changes to the `Sales.Customer` table.

➤ Enforce complex data integrity

DML Triggers	DDL Triggers	Logon Triggers
<ul style="list-style-type: none">DML triggers execute when data is inserted, modified, or deleted in a table or a view using the <code>INSERT</code>, <code>UPDATE</code>, or <code>DELETE</code> statements.	<ul style="list-style-type: none">DDL triggers execute when a table or a view is created, modified, or deleted using the <code>CREATE</code>, <code>ALTER</code>, or <code>DROP</code> statements.	<ul style="list-style-type: none">Logon triggers execute stored procedures when a session is established with a <code>LOGON</code> event. These triggers are invoked after the login authentication is complete and before the actual session is established. Logon triggers control server sessions by restricting invalid logins or limiting the number of sessions.

Unlike `CHECK` constraints, triggers can reference the columns in other tables. This feature can be used to apply complex data integrity checks. Data integrity can be enforced by:

- Checking constraints before cascading updates or deletes.
- Creating multi-row triggers for actions executed on multiple rows.
- Enforcing referential integrity between databases.

➤ Define custom error messages

Custom error messages are used for providing more suitable or detailed explanations in certain error situations. Triggers can be used to invoke such predefined custom error messages when the relevant error conditions occur.

➤ Maintain denormalized data

Low-level data integrity can be maintained in denormalized database environments using triggers. Denormalized data generally refers to redundant or derived data. Here, triggers are used for checks that do not require exact matches. For example, if the value of the year is to be checked against complete dates, a trigger can be used to perform the check.

➤ Compare before and after states of data

Triggers provide the option to reference changes that are made to data by `INSERT`, `UPDATE`, and `DELETE`

statements. This allows users to reference the affected rows when modifications are carried out through triggers.

12.3 Types of Triggers

Triggers are automatically executed when a language event occurs in a table or a view. Language events can be classified as DML and DDL events. Triggers associated with DML events are known as DML triggers, whereas triggers associated with DDL events are known as DDL triggers. Triggers in SQL Server can be classified into three basic types:

12.4 Creating DML Triggers

DML triggers are executed either on completion of the DML events or in place of the DML events. These triggers enforce referential integrity by cascading changes to related tables when a row is modified. They can perform multiple actions for each modification statement.

DML triggers are of three main types:

- INSERT trigger
- UPDATE trigger
- DELETE trigger

Figure 12.2 displays different types of DML events.

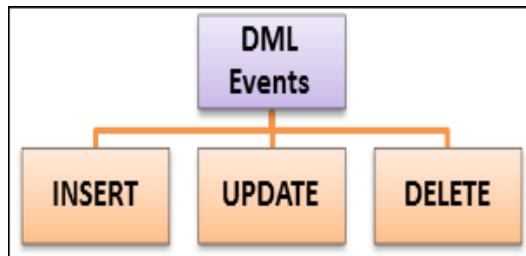


Figure 12.2: DML Events

12.5 Introduction to Inserted and Deleted Tables

SQL statements in DML triggers use two special types of tables to modify data in the database. When the data is inserted, updated, or deleted, SQL Server creates and manages these tables automatically. Tables temporarily store the original as well as the modified data. These tables are as follows:

- **Inserted Table**

Inserted table stores copies of the affected rows during `INSERT` and `UPDATE` statements. During an insert or update transaction, new rows are added to both the inserted table and the trigger table. The rows in the inserted table are copies of the new rows in the trigger table.

- **Deleted Table**

Deleted table stores copies of the affected rows during `DELETE` and `UPDATE` statements. During the execution of a `DELETE` or `UPDATE` statement, rows are deleted from the trigger table and transferred to the deleted table.

Note - Inserted and deleted tables are temporary, memory-resident tables to test the effects of certain data modifications and to set conditions for DML trigger actions. SQL Server 2019 (15.x) does not allow text, ntext, or image column references in the inserted and deleted tables for `AFTER` triggers.

12.6 Insert Triggers

An `INSERT` trigger is executed when a new record is inserted in a table. `INSERT` trigger ensures that the value being entered conforms to the constraints defined on that table.

When a user inserts a record in the table, `INSERT` trigger saves a copy of that record in inserted table. It then checks whether the new value in the inserted table conforms to the specified constraints or not.

If the record is valid, the `INSERT` trigger inserts the row in the trigger table otherwise, it displays an error message. An `INSERT` trigger is created using the `INSERT` keyword in the `CREATE TRIGGER` and `ALTER TRIGGER` statements.

Following is the general syntax for creating an `INSERT | UPDATE | DELETE` trigger:

Syntax:

```
CREATE TRIGGER [schema_name.] trigger_name ON [schema_name.] table_name  
[WITH ENCRYPTION]  
{FOR INSERT} AS  
[IF UPDATE (column_name)...]  
[{AND | OR} UPDATE (column_name)...]  
<sql_statements>
```

where,

- `schema_name`: specifies the name of the schema to which the table/trigger belongs.
- `trigger_name`: specifies the name of the trigger.
- `table_name`: specifies the table to which DML trigger to be applied.
- `WITH ENCRYPTION`: encrypts the text of the `CREATE TRIGGER` statement.
- `FOR`: specifies that the DML trigger executes after the modification operations are complete.
- `INSERT`: specifies that this DML trigger will be invoked by insert operations.
- `UPDATE`: Returns a Boolean value that indicates whether an `INSERT` or `UPDATE` attempt was made on a specified column.
- `column_name`: Is the name of the column to test for the `UPDATE` action.
- `AND`: Combines two Boolean expressions and returns `TRUE` when both expressions are `TRUE`.
- `OR`: Combines two Boolean expressions and returns `TRUE` if at least one expression is `TRUE`.
- `sql_statement`: specifies the SQL statements that are executed in the DML trigger.

Code Snippet 1 creates two tables with mentioned details into AdventureWorks2019 database.

Code Snippet 1:

```
CREATE TABLE Locations (LocationID int, LocName varchar(100));  
CREATE TABLE LocationHistory (LocationID int, ModifiedDate DATETIME);
```

Code Snippet 2 creates an `INSERT` trigger named `TRIGGER_INSERT_Locations` on `Locations` table.

Code Snippet 2:

```
CREATE TRIGGER TRIGGER_INSERT_Locations ON Locations
FOR INSERT
NOT FOR REPLICATION
AS
BEGIN
    INSERT INTO LocationHistory
    SELECT LocationID
    ,getdate()
    FROM inserted
END;
```

The trigger, `TRIGGER_INSERT_Locations` will get executed when a record is inserted in `Locations` table. It then inserts `LocationID` and current date from `inserted` table to `LocationHistory` table.

To test the created `INSERT` trigger, code from Code Snippet 3 is used.

Code Snippet 3:

```
INSERT INTO dbo.Locations (LocationID,LocName) VALUES (443101,'Alaska');
```

Then `SELECT` all records from `LocationHistory` table to check `TRIGGER_INSERT_Locations` is executed or not as shown in figure 12.3.

The screenshot shows a SQL query window titled "SQLQuery1.sql - DE...D6\Lenovo pc (64)*". The query is as follows:

```
INSERT INTO dbo.Locations (LocationID,LocName) VALUES (443101,'Alaska');
SELECT * FROM LocationHistory;
```

The results pane shows one row in the "LocationHistory" table:

	LocationID	ModifiedDate
1	443101	2020-10-22 14:24:23.233

Figure 12.3: Insert Trigger Test

12.7 Update Triggers

The `UPDATE` trigger copies the original record in the **Deleted** table and the new record into the **Inserted** table when a record is updated. It then evaluates the new record to determine if the values conform to the constraints specified in the trigger table.

If the new values are valid, the record from the **Inserted** table is copied to the trigger table. However, if the new values are invalid, an error message is displayed. Also, the original record is copied from the **Deleted** table back into the trigger table.

An `UPDATE` trigger is created using the `UPDATE` keyword in the `CREATE TRIGGER` and `ALTER TRIGGER` statements. Code Snippet 4 creates an `UPDATE` trigger at the table-level.

Code Snippet 4:

```
CREATE TRIGGER TRIGGER_UPDATE_Locations ON Locations
FOR UPDATE
NOT FOR REPLICATION
AS
BEGIN
    INSERT INTO LocationHistory
    SELECT LocationID
    ,getdate()
    FROM inserted
END;
```

The trigger, TRIGGER_UPDATE_Locations is created on Locations table and will get executed when a record is updated in Locations table. It then inserts the LocationID and current date from inserted table to LocationHistory table. The FOR UPDATE clause specifies that this DML trigger will be invoked after the update operations.

To test the created UPDATE trigger, code from Code Snippet 5 is used.

Code Snippet 5:

```
UPDATE dbo.Locations
SET LocName='Atlanta'
Where LocationID=443101;
```

Then, SELECT all records from LocationHistory table to check TRIGGER_UPDATE_Locations is executed or not as shown in figure 12.4.

LocationID	ModifiedDate
1	2020-10-22 14:24:23.233
2	2020-10-22 16:39:14.070

Figure 12.4: Update Trigger Test

12.8 Delete Triggers

The DELETE trigger can be created to restrict a user from deleting a particular record in a table. Following will happen if the user tries to delete the record:

- The record is deleted from the trigger table and inserted in the Deleted table.

- It is checked for constraints against deletion.
- If there is a constraint on the record to prevent deletion, the `DELETE` trigger displays an error message.
- The deleted record stored in the **Deleted table** is copied back to the trigger table.

A `DELETE` trigger is created using the `DELETE` keyword in the `CREATE TRIGGER` statement as shown in Code Snippet 6.

Code Snippet 6:

```
CREATE TRIGGER TRIGGER_DELETE_Locations ON Locations
FOR DELETE
NOT FOR REPLICATION
AS
BEGIN
    INSERT INTO LocationHistory
    SELECT LocationID
    ,getdate()
    FROM deleted
END;
```

The trigger `TRIGGER_DELETE_Locations` is created on `Locations` table and will get executed when a record is deleted from `Locations` table. It then inserts the `LocationID` and current date from deleted table to `LocationHistory` table.

To test the created `DELETE` trigger, code from Code Snippet 7 is used.

Code Snippet 7:

```
DELETE FROM dbo.Locations
Where LocationID=443101;
```

Then, `SELECT` all records from `LocationHistory` table to check `TRIGGER_DELETE_Locations` is executed or not as shown in figure 12.5.

	LocationID	ModifiedDate
1	443101	2020-10-22 14:24:23.233
2	443101	2020-10-22 16:39:14.070
3	443101	2020-10-22 17:10:58.517

Figure 12.5: Delete Trigger Test

12.9 AFTER Triggers

An AFTER trigger is executed on completion of INSERT, UPDATE, or DELETE operations. A table can have multiple AFTER triggers defined for each INSERT, UPDATE, and DELETE operation. If multiple AFTER triggers are created on the same table, the user must define the order in which the triggers must be executed. An AFTER trigger is executed when the constraint check in the table is completed. Also, the trigger is executed after the **Inserted** and **Deleted** tables are created.

Figure 12.6 displays different types of AFTER triggers.

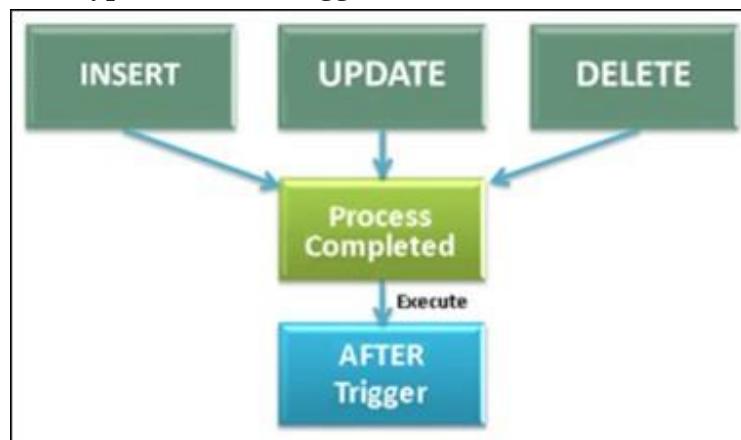


Figure 12.6: AFTER Triggers

Following is the syntax for creating an AFTER trigger:

Syntax

```
CREATE TRIGGER <trigger_name> ON <table_name>
    [WITH ENCRYPTION]
    {FOR | AFTER}
    { [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
    AS <sql_statement>
```

where,

FOR | AFTER: specifies that the DML trigger executes after the modification operations are complete.

{ [INSERT] [,] [UPDATE] [,] [DELETE] }: specifies the operations that invoke the DML trigger

Code Snippet 8 creates an AFTER INSERT trigger on the **Locations** table. If a new record is inserted in the table, the AFTER INSERT trigger activates. The trigger inserts the LocationID and current date from inserted table to **LocationHistory** table.

Code Snippet 8:

```
CREATE TRIGGER AFTER_INSERT_Locations ON Locations
AFTER INSERT
AS
BEGIN
    INSERT INTO LocationHistory
    SELECT LocationID
    ,getdate()
```

```
FROM inserted  
END;
```

To test the created AFTER INSERT trigger, code from Code Snippet 9 is used.

Code Snippet 9:

```
INSERT INTO dbo.Locations (LocationID,LocName) VALUES (443103,'SAN ROMAN');
```

Then, SELECT all records from **LocationHistory** table to check AFTER_INSERT_Locations is executed or not as shown in figure 12.7.

The screenshot shows a SQL query window with two statements. The first statement is an `INSERT` into the `dbo.Locations` table with `LocationID = 443103` and `LocName = 'SAN ROMAN'`. The second statement is a `SELECT * FROM LocationHistory`. Below the results pane, there is a table showing the data from the `LocationHistory` table. The table has two columns: `LocationID` and `ModifiedDate`. There are five rows, with the last row (LocationID 443103, ModifiedDate 2020-10-22 18:45:24.767) highlighted with a blue border.

LocationID	ModifiedDate
1	2020-10-22 14:24:23.233
2	2020-10-22 16:39:14.070
3	2020-10-22 17:10:58.517
4	2020-10-22 18:45:24.730
5	2020-10-22 18:45:24.767

Figure 12.7: After Insert Trigger Test

12.10 INSTEAD OF Triggers

An INSTEAD OF trigger is executed in place of the INSERT, UPDATE, or DELETE operations. INSTEAD OF triggers can be created on tables as well as views. A table or a view can have only one INSTEAD OF trigger defined for each INSERT, UPDATE, and DELETE operation.

The INSTEAD OF triggers are executed before constraint checks are performed on the table. These triggers are executed after the creation of the **Inserted** and **Deleted** tables.

Code Snippet 10 creates the trigger `INSTEADOF_DELETE_Locations` for delete event on **Locations** table.

Code Snippet 10:

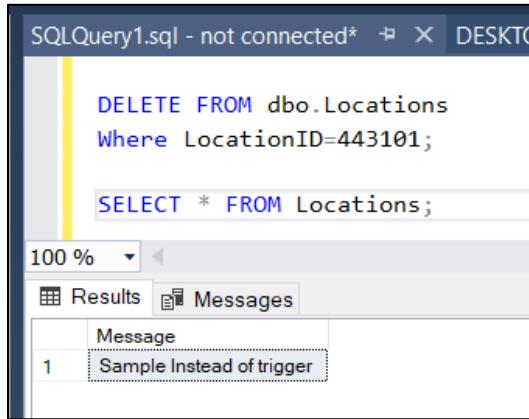
```
CREATE TRIGGER INSTEADOF_DELETE_Locations ON Locations  
INSTEAD OF DELETE  
AS  
BEGIN  
    SELECT 'Sample Instead of trigger' as [Message]  
END;
```

When delete statement is issued against the table, INSTEAD OF trigger is fired and the T-SQL block inside the triggers in SQL Server is executed, but the actual delete operation does not happen.

Code Snippet 11:

```
DELETE FROM dbo.Locations  
WHERE LocationID=443101;
```

The output of Code Snippet 11 is shown in figure 12.7.



```
SQLQuery1.sql - not connected* X DESKTOP-  
DELETE FROM dbo.Locations  
Where LocationID=443101;  
  
SELECT * FROM Locations;  
100 % ◀  
Results Messages  
Message  
1 Sample Instead of trigger
```

Figure 12.7: Instead of Trigger Test

12.11 Execution Order of DML Triggers

SQL Server allows users to specify which AFTER trigger is to be executed first and which is to be executed last. All AFTER triggers invoked between the first and last triggers have no definite order of execution.

All the triggering actions have a first and last trigger defined for them. However, no two triggering actions on a table can have the same first and last triggers.

Users can use the `sp_settriggerorder` stored procedure to define the order of DML AFTER triggers.

Figure 12.8 displays the execution order of DML Triggers.

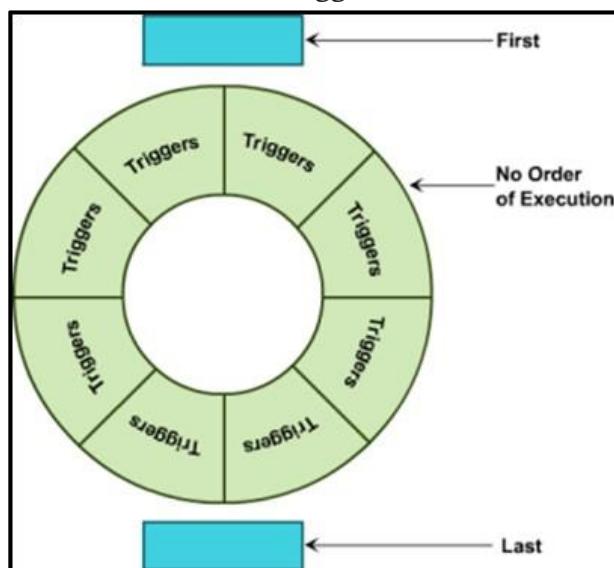


Figure 12.8: Execution Order of DML Triggers

Following is the syntax for specifying execution order of multiple AFTER DML triggers:

Syntax

```
sp_settriggerorder [@triggername = ] '[ triggerschema. ] triggername'  
, [@order = ] 'value'  
, [@stmttype = ] 'statement_type'
```

where,

[**triggerschema.**] **triggername**: is the name of the DML or DDL trigger and the schema to which it belongs and whose order needs to be specified.

value: specifies the execution order of the trigger as FIRST, LAST, or NONE. If FIRST is specified, then the trigger is fired first. If LAST is specified, the trigger is fired last. If NONE is specified, the order of the firing of the trigger is undefined.

statement_type: specifies the type of SQL statement (INSERT, UPDATE, or DELETE) that invokes the DML trigger.

Code Snippet 12 first executes the TRIGGER_DELETE_Locations trigger defined on Locations table when the DELETE operation is performed on the table row.

Code Snippet 12:

```
EXEC sp_settriggerorder @triggername = 'TRIGGER_DELETE_Locations', @order =  
'FIRST', @stmttype = 'DELETE'
```

12.12 Viewing Definitions of DML Triggers

A trigger definition includes the trigger name, the table on which the trigger is created, the triggering actions, and the SQL statements that are executed. SQL Server provides `sp_helptext` stored procedure to retrieve the trigger definitions.

The DML trigger name must be specified as the parameter when executing `sp_helptext`.

Note - Trigger definition cannot be viewed if the definition is encrypted.

Following is the syntax for viewing a DML trigger:

Syntax

```
sp_helptext '<DML_trigger_name>;'
```

where,

DML_trigger_name: specifies the name of the DML trigger whose definitions are to be displayed.

Code Snippet 13 displays the definition of the trigger, **TRIGGER_DELETE_Locations**, created on the Locations table.

Code Snippet 13:

```
sp_helptext TRIGGER_DELETE_Locations
```

12.13 Modifying Definitions of DML Triggers

Trigger parameters are defined at the time of creating a trigger. These parameters include the type of triggering action that invokes the trigger and the SQL statements that are executed.

If the user wants to modify any of these parameters for a DML trigger, a user can do so in any one of two ways:

- Drop and re-create the trigger with the new parameters.
- Change the parameters using the `ALTER TRIGGER` statement.

If the object referencing a DML trigger is renamed, the trigger must be modified to reflect the change in object name.

Note - A DML trigger can be encrypted to hide its definition.

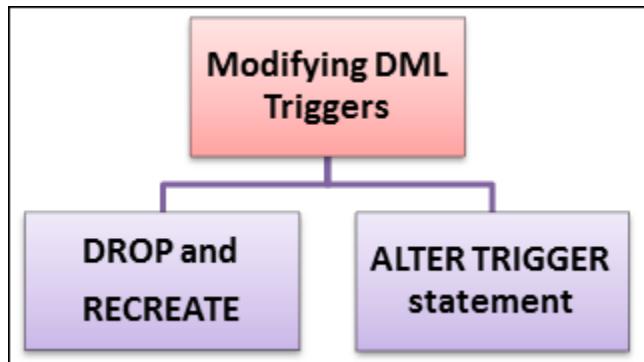


Figure 12.9: Modifying DML Triggers

Following is the syntax for modifying a DML trigger:

Syntax

```
ALTER TRIGGER <trigger_name>
ON {<table_name> | <view_name>}
    [WITH ENCRYPTION]
    {FOR | AFTER | INSTEAD OF}
    { [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
    AS <sql_statement>
```

where,

`WITH ENCRYPTION`: specifies that the DML trigger definitions are not displayed.

`FOR | AFTER`: specifies that the DML trigger executes after modification operations are complete.

`INSTEAD OF`: specifies that the DML trigger executes in place of modification operations.

Code Snippet 14 alters the `TRIGGER_UPDATE_Locations` trigger created on the `Locations` table using the `WITH ENCRYPTION` option.

Code Snippet 14:

```
ALTER TRIGGER TRIGGER_UPDATE_Locations ON Locations
WITH ENCRYPTION FOR INSERT
AS
IF '443101' IN (SELECT LocationID FROM inserted)
BEGIN
PRINT 'Location cannot be updated'
ROLLBACK TRANSACTION
END;
```

Now, if the user tries to view the definition of the TRIGGER_UPDATE_Locations trigger using the sp_helptext stored procedure, the following error message is displayed:

The text for object TRIGGER_UPDATE_Locations is encrypted as shown in figure 12.10.

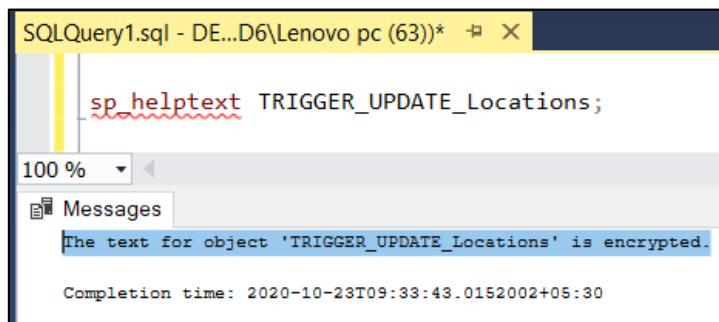


Figure 12.10: Encrypted Trigger

12.14 Dropping DML Triggers

SQL Server provides the option of dropping a DML trigger created on a table if the trigger is no longer required. The trigger can be dropped using the `DROP TRIGGER` statement. Multiple triggers can also be dropped using a single drop trigger statement.

When a table is dropped, all the triggers defined on that table are also dropped.

Figure 12.11 depicts the concept of dropped DML triggers.

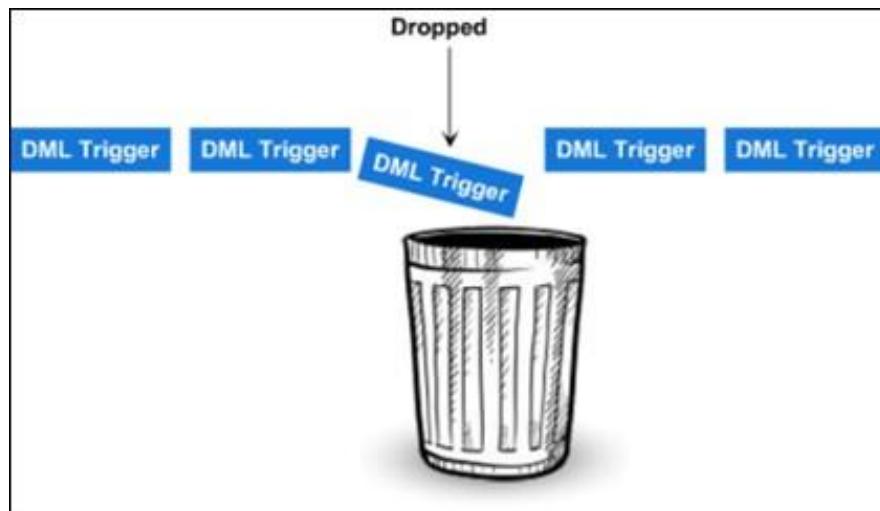


Figure 12.11: Dropping DML Triggers

Note - When the DML trigger is deleted from the table, the information about the trigger is also removed from the catalog views.

Following is the syntax for dropping DML triggers:

Syntax

```
DROP TRIGGER <DML_trigger_name> [ ,...n ]
```

where,

`DML_trigger_name`: specifies the name of the DML trigger to be dropped.

`[,...n]`: specifies that multiple DML triggers can be dropped.

Code Snippet 15 drops the `TRIGGER_UPDATE_Locations` trigger created on `Locations` table.

Code Snippet 15:

```
DROP TRIGGER TRIGGER_UPDATE_Locations
```

12.15 DDL Triggers

A Data Definition Language (DDL) triggers execute stored procedures when DDL events such as `CREATE`, `ALTER`, and `DROP` statements occur in the database or the server. DDL triggers can operate only on completion of DDL events.

DDL triggers can be used to prevent modifications in the database schema. A schema is a collection of objects such as tables, views, and so forth in a database.

DDL triggers can invoke an event or display a message based on the modifications attempted on the schema. DDL triggers are defined either at the database level or at the server level. Figure 12.12 displays different types of DDL triggers.

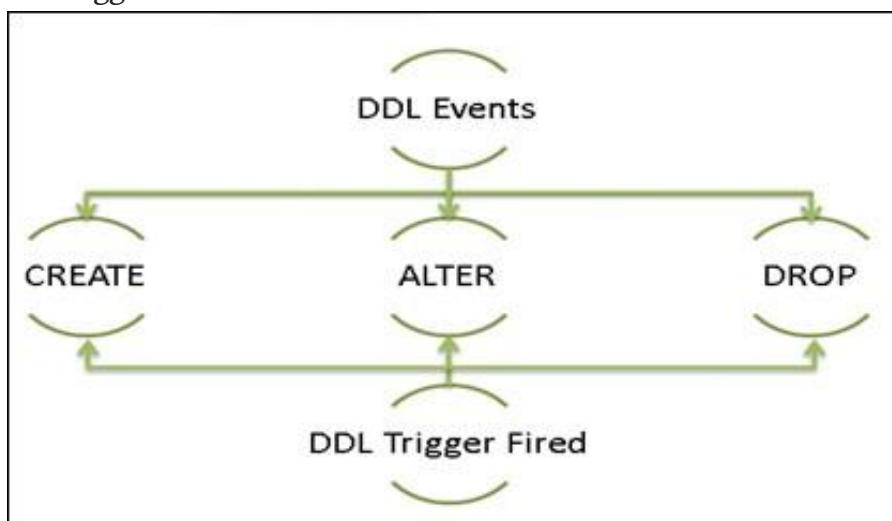


Figure 12.12: DDL Triggers

Following is the syntax for creating DDL triggers:

Syntax

```
CREATE TRIGGER <trigger_name> ON { ALL SERVER  
| DATABASE } [WITH ENCRYPTION]  
{ FOR | AFTER } { <event_type> }  
AS <sql_statement>
```

where,

ALL SERVER: specifies that the DDL trigger executes when DDL events occur in the current server.

DATABASE: specifies that the DDL trigger executes when DDL events occur in the current database.

event_type: specifies the name of the DDL event that invokes the DDL trigger.

Code Snippet 16 creates a DDL trigger for dropping and altering a table.

Code Snippet 16:

```
CREATE TRIGGER Secure ON DATABASE  
FOR DROP_TABLE, ALTER_TABLE AS  
PRINT 'You must disable Trigger "Secure" to drop or alter tables!'  
ROLLBACK;
```

In this code, the DDL trigger is created for DROP TABLE and ALTER TABLE statements.

12.15.1 Scope of DDL Triggers

DDL triggers are invoked by SQL statements executed either in the current database or on the current server. For example, a DDL trigger created for a CREATE TABLE statement executes on the CREATE TABLE event in the database. A DDL trigger created for a CREATE LOGIN statement executes on CREATE LOGIN event in the server.

The scope of the DDL trigger depends on whether the trigger executes for database events or server events. Accordingly, the DDL triggers are classified into two types, which are as follows:

- **Database-Searched DDL Triggers**

Database-scoped DDL triggers are invoked by the events that modify the database schema. These triggers are stored in the database and execute on DDL events, except those related to temporary tables.

- **Server-Searched DDL Triggers**

Server-scoped DDL triggers are invoked by DDL events at the server level. These triggers are stored in the master database.

Figure 12.13 displays the scope of DDL triggers.

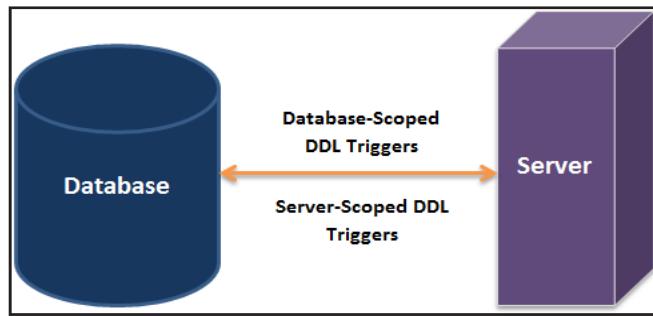


Figure 12.13: Scope of DDL Triggers

12.16 Nested Triggers

Both DDL and DML triggers are nested when a trigger implements an action that initiates another trigger. DDL and DML triggers can be nested up to 32 levels. Suppose if a trigger modifies a table on which there is another trigger, the second trigger is initiated, which then calls a third trigger, and so on.

If the nested triggers are allowed, then the triggers in the sequence start an infinite loop. This will exceed the nesting level and the trigger will terminate.

Nested triggers can be used to perform the functions such as storing the backup of the rows that are affected by the previous actions.

A Transact-SQL trigger executes the managed code through referencing a CLR routine, aggregate, or type, that references the counts as one level against the 32-level nesting limit. Methods which are invoked from within managed code are not counted against this limit.

Users can disable nested triggers, by setting the nested triggers option of `sp_configure` to 0 or OFF. The default configuration is allowed for nested triggers. If nested trigger option is off, then the recursive trigger is disabled, irrespective of the recursive triggers setting that is set by using the `ALTER DATABASE`.

Code Snippet 17 creates an AFTER DELETE trigger named `Employee_Deletion` on `HumanResources.Employee` table in **AdventureWorks2019** database.

Code Snippet 17:

```

CREATE TRIGGER Employee_Deletion ON HumanResources.Employee
AFTER DELETE
AS
BEGIN
PRINT 'Deletion will affect EmployeePayHistory table'
DELETE FROM EmployeePayHistory WHERE BusinessEntityID IN (SELECT
BusinessEntityID FROM deleted)
END;

```

When a record is deleted from the `HumanResources.Employee` table, the `Employee_Deletion` trigger is activated and a message is displayed. Also, the record of the employee is deleted from the `HumanResources.EmployeePayHistory` table.

Code Snippet 18 creates an AFTER DELETE trigger `Deletion_Confirmation` on the

HumanResources.EmployeePayHistory table.

Code Snippet 18:

```
CREATE TRIGGER Deletion_Confirmation
ON HumanResources.EmployeePayHistory AFTER DELETE
AS
BEGIN
PRINT 'Employee details successfully deleted from EmployeePayHistory table'
END;
DELETE FROM EmployeePayHistory WHERE EmpID=1
```

When a record is deleted from the HumanResources.EmployeePayHistory table, the Deletion_Confirmation trigger is activated. This trigger prints the confirmation message of the record being deleted.

Thus, the Employee_Deletion and the Deletion_Confirmation triggers are seen to be nested.

12.17 UPDATE()

UPDATE () function returns a Boolean value that specifies whether an UPDATE or INSERT action was performed on a specific view or column of a table.

UPDATE() function can be used anywhere inside the body of a Transact-SQL UPDATE or INSERT trigger to test whether the trigger should execute some actions.

Following is the syntax for UPDATE():

Syntax

```
UPDATE (column);
```

where,

column: is the name of the column to test for either an INSERT or UPDATE action.

Code Snippet 19 creates a trigger Accounting on the Production.TransactionHistory table to update the columns TransactionID or ProductID.

Code Snippet 19:

```
CREATE TRIGGER Accounting ON Production.TransactionHistory AFTER UPDATE
AS
IF (UPDATE (TransactionID) OR UPDATE (ProductID) ) BEGIN
RAISERROR (50009, 16, 10) END;
GO
```

12.18 Handling of Multiple Rows in a Session

When a user writes the code for a DML trigger, then the statement that causes the trigger to fire will be single statement. This single statement will affect multiple rows of data, instead of a single row. This is a common behavior for DELETE and UPDATE triggers as these statements often affect multiple rows. The behavior for INSERT triggers is less common as the basic INSERT statement adds only one row.

When the functionality of a DML trigger involves automatically recalculating summary values of one table and storing the result in another table, then multirow considerations are important.

Code Snippet 20 stores a running total for a single-row insert.

Code Snippet 20:

```
USE AdventureWorks2019;
GO
CREATE TRIGGER PODetails
ON Purchasing.PurchaseOrderDetail AFTER INSERT AS
UPDATE Purchasing.PurchaseOrderHeader
SET SubTotal = SubTotal + LineTotal FROM inserted
WHERE PurchaseOrderHeader.PurchaseOrderID = inserted.PurchaseOrderID;
```

In this code, the subtotal is calculated and stored for a single-row insert operation. Code Snippet 21 stores a running total for a multi-row or single-row insert.

Code Snippet 21:

```
USE AdventureWorks2019;
GO
CREATE TRIGGER PODetailsMultiple
ON Purchasing.PurchaseOrderDetail AFTER INSERT AS
UPDATE Purchasing.PurchaseOrderHeader SET SubTotal = SubTotal +
(SELECT SUM(LineTotal) FROM inserted
WHERE PurchaseOrderHeader.PurchaseOrderID
= inserted.PurchaseOrderID)
WHERE PurchaseOrderHeader.PurchaseOrderID IN (SELECT PurchaseOrderID FROM
inserted);
```

In this code, the subtotal is calculated and stored for a multi-row or single-row insert operation.

12.19 LOGON Triggers

These triggers are fired in response to a `LOGON` event in SQL Server. Logon triggers fire stored procedures in response to a `LOGON` event. This event is raised when a user session is established with an instance of SQL Server. Logon triggers fire after the authentication phase of logging in finishes, but before the user session is actually established. Therefore, all messages originating inside the trigger that would typically reach the user, such as error messages and messages from the `PRINT` statement are diverted to the SQL Server error log. Logon triggers do not fire if authentication fails.

`LOGON` triggers are created at the server level and are useful in following cases:

- To audit login activity
- To control the login activity

LoginActivity table with **LOGONEvent** and **Logintime** columns is created in **AdventureWorks2019** database for this example as follows:

```
CREATE TABLE LoginActivity (LOGONEvent XML, Logintime datetime)
```

Use `EVENTDATA()` and write your custom code to track or control the connections. Code Snippet 22 creates a simple trigger in SQL Server for `LOGON` event. This `LOGON` trigger tracks login activities on all SQL server instances available and insert event related `XML` data and `Date` of login activity into `LoginActivity` table in **AdventureWorks2019** database.

Code Snippet 22:

```
CREATE TRIGGER [track_logins] ON ALL SERVER
FOR LOGON AS
BEGIN
    INSERT INTO LoginActivity
    SELECT EVENTDATA()
    , GETDATE()
END;
```

Note: User must be cautious while creating these triggers as login may fail if the trigger execution fails or if you do not have access to objects referenced in the `LOGON` trigger. In such cases, the only member of the `sysadmin` role can connect to the server using a dedicated administrator connection. So, it is always better to enable dedicated administrator connection when using these triggers.

12.20 Performance Implication of Triggers

In reality, triggers do not carry overheads, rather they are quite responsive. However, many performance issues can occur because of the logic present inside the trigger. Suppose a trigger creates a cursor and loops through many rows, then there will be a slowdown in the process.

Similarly, consider that the trigger executes various SQL statements against other tables separate from the `Inserted` and `Deleted` tables. This will again result in the slowdown of speed of SQL statements that are within the trigger.

A good rule will be to keep the logic simple within the triggers and avoid using cursors while executing statements against another table and different tasks that cause performance slowdown.

12.21 Check Your Progress

1. Which of these statements about triggers in SQL Server 2019 are true?

a.	Triggers retrieve information from tables of the same as well as other databases.	c.	DML triggers execute on INSERT, UPDATE, and DELETE statements.
b.	DDL triggers operate only after a table or a view is modified.	d.	DDL triggers execute either while modifying the data or after the data is modified.

(A)	a, b, c	(C)	a, b ,d
(B)	b, c, d	(D)	a, c, d

2. Match the description with different types of DML triggers in SQL Server 2019.

	Description		DML Trigger
a.	Executes when users replace an existing record with a new value.	1.	INSERT
b.	Executes on completion of the modification operations.	2.	UPDATE
c.	Executes in place of the modification operations.	3.	DELETE
d.	Executes when users add a record on a table.	4.	AFTER
e.	Executes when users remove a record from a table.	5.	INSTEAD OF

(A)	a-1, b-4, c-2, d-3, e-5	(C)	a-2, b-4, c-3, d-5, e-1
(B)	a-2, b-4, c-5, d-1, e-3	(D)	a-1, b-2, c-3, d-4, e-5

3. Which of these statements about DML triggers in SQL Server 2019 are true?

a.	DML triggers can perform multiple actions for each modification statement.	c.	UPDATE triggers do not use the Deleted table to update records in a table.
b.	The Inserted and Deleted tables are created by SQL Server 2019 when a new table is created in the database.	d.	Deleted triggers do not use the Inserted table to delete records from a table.

(A)	a, b	(C)	a, d
(B)	c, d	(D)	b, d

4. Which of these statements about the working with DML triggers of SQL Server 2019 are true?

a.	Each triggering action cannot have multiple AFTER triggers.	c.	DML trigger definition can be modified by dropping and recreating the trigger.
b.	Two triggering actions on a table can have the same first and last triggers.	d.	DML trigger definition can be viewed using the sp_helptext stored procedure.

(A)	a, c	(C)	b, d
(B)	b, c	(D)	c, d

5. _____ triggers can be used to perform the functions such as storing the backup of the rows that are affected by previous actions.

(A)	Nested	(C)	DDL
(B)	DML	(D)	INSTEAD OF

12.21.1 Answers

1.	A
2.	B
3.	C
4.	D
5.	A

Summary

- A trigger is a stored procedure that is executed when an attempt is made to insert, update, or delete data in a table that is protected by the trigger.
- Logon triggers execute stored procedures when a session is established with a `LOGON` event.
- DML triggers are executed when DML events occur in tables or views.
- The `INSERT` trigger is executed when a new record is inserted in a table.
- The `UPDATE` trigger copies the original record in the `Deleted` table and the new record into the `Inserted` table when a record is updated.
- The `DELETE` trigger can be created to restrict a user from deleting a particular record in a table.
- The `AFTER` trigger is executed on completion of `INSERT`, `UPDATE`, or `DELETE` operations.

Try It Yourself

1. **Galaxy Airlines** is a newly launched airline service that operates flights to and from various cities all over Europe. The company maintains the data pertaining to day-to-day transactions regarding flight services in SQL Server 2019 databases. To enable efficient and faster performance, **Galaxy Airlines** has decided to incorporate use of triggers in their database applications. The detailed list of operations to be performed are listed as follows:

- a. Create the following tables in **GalaxyAirlines** database. Table 12.2 lists the **Flight** table.

Field Name	Data Type	Key Field	Description
Aircraft_code	varchar(10)	Primary key	Stores aircraft code
Type	varchar(6)		Describes the type of aircraft
Source	varchar(20)		Stores the name of the city from where the aircraft will depart
Destination	varchar(20)		Stores the name of the city where the aircraft will arrive
Dep_time	varchar(10)		Stores departure time
Journey_hrs	int		Stores journey hours

Table 12.2: Flight Table

Table 12.3 lists the **Flight_Details** table.

Field Name	Data Type	Key Field	Description
Class_Code	varchar(10)	Primary key	Stores the class, whether first, business, or economy
Aircraft_code	varchar(10)	Foreign key	Stores aircraft code
Fare	money		Stores the fare amount
Seats	int		Stores total number of seats on the flight

Table 12.3: Flight_Details Table

- b. Write statements to create a trigger **trgCheckSeats** that will activate whenever a new row is being inserted into the **Flight_Details** table. The maximum limit of seats that a flight can contain is **150**. The trigger should check for the value of seats being inserted. If it is more than 150, the INSERT operation is not allowed to succeed.



Try It Yourself

- c. Insert at least five records in each table.
- d. Write statements to create a trigger **UpdateValid** that will activate whenever a row is being updated in the **Flight_Details** table. The trigger should determine if the Seats column is present in the list of columns being updated. If yes, the UPDATE operation should not succeed because the Seats column is defined as a constant and cannot be changed.
- e. Write statements to create a DDL trigger **ProhibitDelete** that will activate whenever a user is trying to delete a table from the Galaxy Airlines database. The trigger must not allow a user to perform deletes and must display a message ‘You are not allowed to delete tables in this database.

Session - 13

Programming Transact-SQL

Welcome to the Session, **Programming Transact-SQL**.

This session introduces programming with Transact-SQL and describes various Transact-SQL programming elements. The session also describes program flow statements and Transact-SQL functions. The session further explains the procedure to create and alter user-defined functions and create windows using the OVER and window functions.

In this session, you will learn to:

- Describe Transact-SQL programming
- Describe program flow statements
- Describe various Transact-SQL functions
- Explain the procedure to create and alter User-Defined Functions (UDFs)
- Explain creation of windows with OVER
- Describe window functions

13.1 *Introduction*

Transact-SQL programming is a procedural language extension to SQL. Transact-SQL programming is extended by adding the subroutines and programming structures similar to high-level languages. Like high-level languages, Transact-SQL programming also has rules and syntax that control and enable programming statements to work together. Users can control the flow of programs by using conditional statements such as IF and loops such as WHILE.

13.2 *Transact-SQL Programming Elements*

Transact-SQL programming elements enable to perform various operations that cannot be done in a single statement. Users can group several Transact-SQL statements together by using one of the following ways:

- **Batches**

A batch is a collection of one or more Transact-SQL statements that are sent as one unit from an application to the server.

- **Stored Procedures**

A stored procedure is a collection of Transact-SQL statements that are precompiled and predefined on the server.

- **Triggers**

A trigger is a special type of stored procedure that is executed when the user performs an event

such as an `INSERT`, `DELETE`, or `UPDATE` operation on a table.

- **Scripts**

A script is a chain of Transact-SQL statements stored in a file that is used as input to the SSMS code editor or `sqlcmd` utility.

Following features enable users to work with Transact-SQL statements:

- **Variables**

A variable allows a user to store data that can be used as input in a Transact-SQL statement.

- **Control-of-flow**

Control-of-flow is used for including conditional constructs in Transact-SQL.

- **Error Handling**

Error handling is a mechanism that is used for handling errors and provides information to the users about the error occurred.

13.2.1 Transact-SQL Batches

A Transact-SQL batch is a group of one or more Transact-SQL statements sent to the server as one unit from an application for execution. SQL Server compiles the batch SQL statements into a single executable unit, also called as an execution plan. In the execution plan, the SQL statements are executed one by one. A Transact-SQL batch statement should be terminated with a semicolon. This condition is not mandatory, but the facility to end a statement without a semicolon is deprecated and may be removed in the new versions of SQL Server in the future. Hence, it is recommended to use semicolons to terminate batches.

A compile error such as syntax error restricts the compilation of the execution plan. So, if a compile-time error occurs, no statements in the batch are executed.

A run-time error such as a constraint violation or an arithmetic overflow has one of the following effects:

- Most of the run-time errors stop the current statement and the statements that follow in the batch.
- A specific run-time error such as a constraint violation stops only the existing statement and the remaining statements in the batch are executed.

The SQL statements that execute before the run-time error is encountered are unaffected. The only exception is when the batch is in a transaction and the error results in the transaction being rolled back.

For example, suppose there are 10 statements in a batch and the sixth statement has a syntax error, then the remaining statements in the batch will not execute. If the batch is compiled and the third statement fails to run, then, the results of the first two statements remains unaffected as it is already executed.

Following rules are applied to use batches:

1. CREATE FUNCTION, CREATE DEFAULT, CREATE RULE, CREATE TRIGGER, CREATE PROCEDURE, CREATE VIEW, and CREATE SCHEMA statements cannot be jointly used with other statements in a batch. The CREATE SQL statement starts the batch and all other statements that are inside the batch will be considered as part of the CREATE statement definition.
2. No changes are made in the table and the new columns reference the same batch.
3. If the first statement in a batch has the EXECUTE statement, then, the EXECUTE keyword is not required. It is required only when the EXECUTE statement does not exist in the first statement in the batch.

Code Snippet 1 creates a view in a batch.

Code Snippet 1:

```
USE AdventureWorks2019;
GO
CREATE VIEW dbo.vProduct
AS
SELECT ProductNumber, Name FROM Production.Product;
GO
SELECT * FROM dbo.vProduct;
GO
```

In Code Snippet 1, a view is created in a batch. The CREATE VIEW is the only statement in the batch, the GO commands are essential to separate the CREATE VIEW statement from the SELECT and USE statements. This was a simple example to demonstrate the use of a batch. In the real-world, a large number of statements may be used within a single batch. It is also possible to combine two or more batches within a transaction.

Code Snippet 2 shows an example of this.

Code Snippet 2:

```
BEGIN TRANSACTION
GO
USE AdventureWorks2019;
GO
CREATE TABLE Company (
Id_Num int IDENTITY(100, 5),
Company_Name nvarchar(100))
GO
INSERT Company (Company_Name) VALUES (N'A Bike Store')
INSERT Company (Company_Name) VALUES (N'Progressive Sports')
INSERT Company (Company_Name) VALUES (N'Modular Cycle Systems')
INSERT Company (Company_Name) VALUES (N'Advanced Bike Components')
INSERT Company (Company_Name) VALUES (N'Metropolitan Sports Supply')
INSERT Company (Company_Name) VALUES (N'Aerobic Exercise Company')
INSERT Company (Company_Name) VALUES (N'Associated Bikes')
INSERT Company (Company_Name) VALUES (N'Exemplary Cycles')
GO
SELECT Id_Num, Company_Name FROM dbo. Company
ORDER BY Company_Name ASC;
```

```
GO  
COMMIT;  
GO
```

In Code Snippet 2, several batches are combined into one transaction. The BEGIN TRANSACTION and COMMIT statements enclose the transaction statements. The CREATE TABLE, BEGIN TRANSACTION, SELECT, COMMIT, and USE statements are in single-statement batches. The INSERT statements are all included in one batch.

13.2.2 Transact-SQL Variables

Variables allow users to store data for using as input in a Transact-SQL statement. For example, users can create a query that requires various types of data values specified in the WHERE clause each time the query is executed. Here, the users can use variables in the WHERE clause and write the logic to store the variables with the appropriate data.

SQL Server provides the following statements to set and declare local variables.

- **DECLARE**

Variables are declared with the DECLARE statement in the body of a batch. These variables are assigned values by using the SELECT or SET statement. The variables are initialized with NULL values if the user has not provided a value at the time of the declaration.

Following is the basic syntax to declare a local variable:

Syntax

```
DECLARE {{ @local_variable [AS] data_type } | [ = value ] }
```

where,

@local_variable: specifies the name of the variables and begins with @ sign.

data_type: specifies the data type. A variable cannot be of image, text, or ntext data type.

=value: Assigns an inline value to a variable. The value can be an expression or a constant value. The value should match with the variable declaration type or it should be implicitly converted to that type.

Code Snippet 3 uses a local variable to retrieve contact information for the last names starting with ‘Man’.

Code Snippet 3:

```
USE AdventureWorks2019;  
GO  
DECLARE @find varchar(30) = 'Man%';  
SELECT p.LastName, p.FirstName, ph.PhoneNumber FROM Person.Person AS p  
JOIN Person.PersonPhone AS ph ON p.BusinessEntityID = ph.BusinessEntityID  
WHERE LastName LIKE @find;
```

In Code Snippet 3, a local variable named **@find** is used to store the search criteria, which will be then, used to retrieve the contact information. Here, the criteria include all last names beginning with ‘Man’. Figure 13.1 displays the output.

```

SQLQuery1.sql - DE..D6\Lenovo pc (52)*  ×
GO

USE AdventureWorks2019;
GO
DECLARE @find varchar(30) = 'Man%';
SELECT p.LastName, p.FirstName, ph.PhoneNumber FROM Person.Person AS p
JOIN Person.PersonPhone AS ph ON p.BusinessEntityID = ph.BusinessEntityID
WHERE LastName LIKE @find;

```

100 %

	LastName	FirstName	PhoneNumber
1	Manchepalli	Ajay	1 (11) 500 555-0174
2	Manek	Parul	1 (11) 500 555-0146
3	Manzanares	Tomas	1 (11) 500 555-0178

Figure 13.1: Contact Information

- **SET**

The SET statement sets the local variable created by the DECLARE statement to the specified value.

Following is the basic syntax to set a local variable:

Syntax

```

SET
{@local_variable} = {expression}
}
|
{@local_variable}
{+= | -= | *= | /= | %= | &= | ^= | |=} expression
}

```

where,

{@local_variable}: specifies the name of the variable and begins with @ sign.
 =: Assigns the value on the right side to the variable on the left side.
 {+= | -= | *= | /= | %= | &= | ^= | |= }: specifies the compound assignment operators.

They are as follows:

- + = Add and then, assign
- = Subtract and then, assign
- * = Multiply and then, assign
- / = Divide and then, assign
- % = Modulo and then, assign
- & = Bitwise AND and then, assign
- ^ = Bitwise XOR and then, assign
- | = Bitwise OR and then, assign

expression: specifies any valid expression which can even include a scalar subquery.

Code Snippet 4 demonstrates the use of SET to assign a string value to a variable.

Code Snippet 4:

```
DECLARE @myvar char(20);
SET @myvar = 'This is a test';
```

In Code Snippet 4, the @myvar variable is assigned a string value.

- **SELECT**

The SELECT statement indicates that the specified local variable that was created using DECLARE should be set to the given expression.

Following is the syntax of the SELECT statement:

Syntax

```
SELECT {@local_variable {= | += | -= | *= | /= | %= | &= | ^= | |=}
expression} [
,...n] [ ; ]/
```

where,

@local_variable: specifies the local variable to which a value will be assigned.

=: Assigns the value on the right-hand side to the variable on the left-hand side.

{= | += | -= | *= | /= | %= | &= | ^= | |= }: specifies the compound assignment operators.

expression: specifies any valid expression which can even include a scalar subquery.

Code Snippet 5 shows how to use SELECT to return a single value.

Code Snippet 5:

```
USE AdventureWorks2019;
GO
DECLARE @var1 nvarchar(30);
SELECT @var1 = 'Unnamed Company';
SELECT @var1 = Name FROM Sales.Store WHERE BusinessEntityID = 10;
SELECT @var1 AS 'Company Name';
```

In Code Snippet 5, the variable @var1 is assigned Unnamed Company as its value.

The query against the Store table will return zero rows as the value specified for the BusinessEntityID does not exist in the table. The variable will then, retain the Unnamed Company value and will be displayed with the heading Company Name. Figure 13.2 displays the output.

```

USE AdventureWorks2019;
GO
DECLARE @var1 nvarchar(30);
SELECT @var1 = 'Unnamed Company';
SELECT @var1 = Name FROM Sales.Store
WHERE BusinessEntityID = 10;
SELECT @var1 AS 'Company Name';

```

	Company Name
1	Unnamed Company

Figure 13.2: Generic Name

Though both the `SET` and `SELECT` statements look similar, they are not. Here are a few differences between the two:

- It is possible to assign only one variable at a time using `SET`. However, using `SELECT` you can make multiple assignments at once.
- `SET` can only assign a scalar value when assigning from a query. It raises an error and does not work if the query returns multiple values/rows. However, `SELECT` assigns one of the returned values to the variable and the user will not even know that multiple values were returned.

Note - To assign variables, it is recommended to use `SET @local_variable` instead of `SELECT @local_variable`.

13.3 Synonyms

Synonyms are database objects that serve the following purposes:

- They offer another name for a different database object, also called as the base object, which may exist on a remote or local server.
- They present a layer of abstraction that guards a client application from the modifications made to the location and the name of the base object.

For example, consider that the `Department` table of `AdventureWorks2019` is located on the first server named `Server1`. To reference this table from the second server, `Server2`, a client application would have to use the following four-part name:

`Server1.AdventureWorks2019.HumanResources.Department`

If the location of the table was modified, for example, to another server, the client application would have to be rectified to reflect that change. To address both these issues, users can create a synonym, `DeptEmpTable`, on `Server2` for the `Department` table on `Server1`.

Now, the client application only has to use the single name, `DeptEmpTable`, to refer to the `Department` table.

Similarly, if the location of **Department** table changes, users have to modify the synonym, **DeptEmpTable**, to point to the new location of the **Department** table. Since, there is no ALTER SYNONYM statement, you first have to drop the synonym, **DeptEmpTable**, and then, re-create the synonym with the same name, but point the synonym to the new location of **Department**.

Note - A synonym is a part of schema, and similar to other schema objects, the synonym name must be unique.

Table 13.1 lists the database objects for which the users can create synonyms.

Database Objects
Extended stored procedure
SQL table-valued function
SQL stored procedure
Table (User-defined)
Replication-filter-procedure
SQL scalar function
SQL inline-table-valued function
View

Table 13.1: Database Objects

- **Synonyms and Schemas**

Suppose users want to create a synonym and have a default schema that is not owned by them. In such a case, they can qualify the synonym name with the schema name that they actually own. Consider, for example, a user owns a schema **Resources**, but **Materials** is the user's default schema. If this user wants to create a synonym, he/she must prefix the name of the synonym with the schema **Resources**.

- **Granting Permissions on Synonyms**

Only members of the roles `db_owner` or `db_ddladmin` or synonym owners are allowed to grant permissions on a synonym. Users can deny, grant, or revoke all or any of the permissions on a synonym. Table 13.2 displays the list of permissions that are applied on a synonym.

Permissions
DELETE
INSERT
TAKE OWNERSHIP
VIEW DEFINITION
CONTROL
EXECUTE
SELECT
UPDATE

Table 13.2: Permissions

- **Working with Synonyms**

Users can work with synonyms in SQL Server 2019 using either Transact-SQL or SSMS. To create a synonym using SSMS, perform the following steps:

1. In **Object Explorer**, expand the database where you want to create a new synonym.
2. Select the **Synonyms** folder, right-click it and then, click **New Synonym** as shown in figure 13.3.

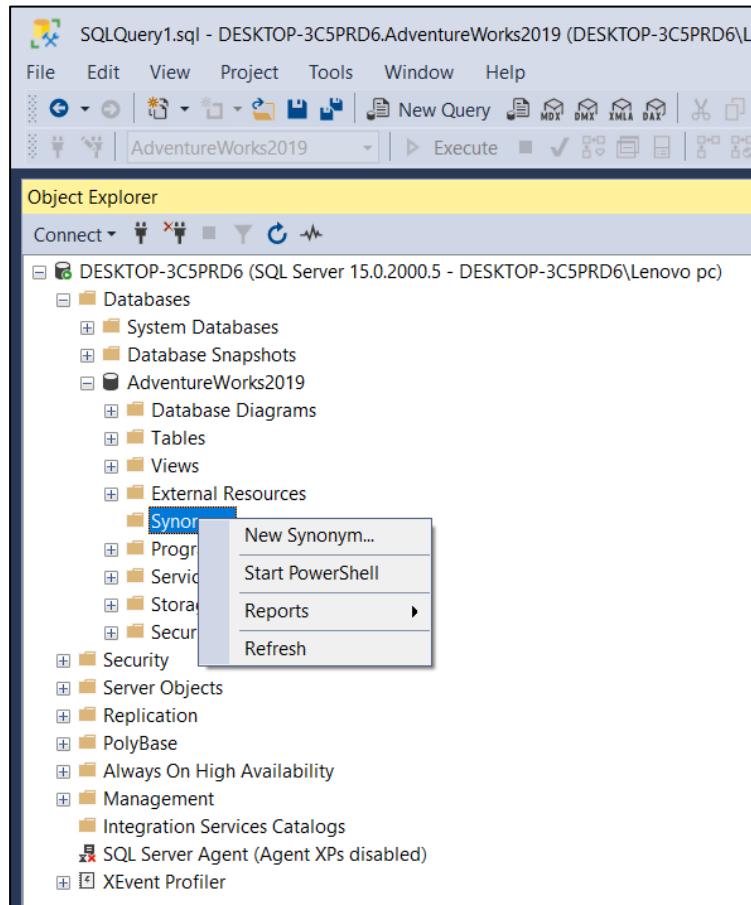


Figure 13.3: Creating a New Synonym

3. In the **New Synonym** dialog box, provide the information as shown in figure 13.4.

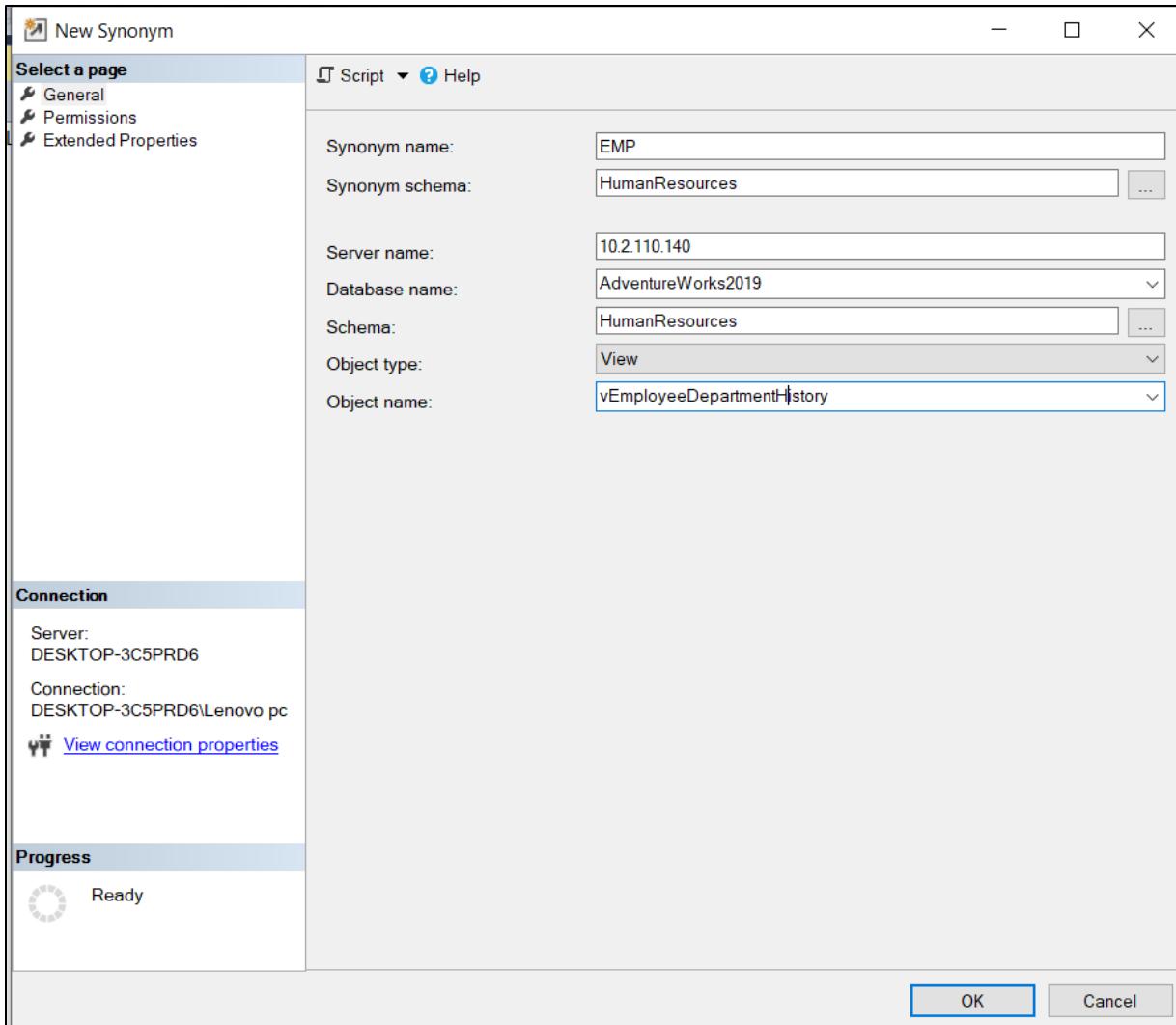


Figure 13.4: Adding Details in the New Synonym Dialog Box

where,

Synonym name: is the new name for the object. Here, **Emp** is the name.

Synonym schema: is the new name for the schema object. Here, the same schema name **HumanResources** is used for the synonym and the object type.

Server name: is the name of the server to be connected. Here, the server name is specified as **10.2.110.140**.

Database name: is the database name to connect the object. Here, **AdventureWorks2019** is the database name.

Schema: is the schema that owns the object.

Object type and **Object name:** is the object type and name respectively. Here, the object type selected is **view** and the object name that refers to the synonym is **vEmployeeDepartmentHistory**.

To create a synonym using Transact-SQL, perform the following steps:

1. Connect to the Database Engine.
2. Click **New Query** in the Standard bar.
3. Write the query to create the synonym in the query window.
4. Click Execute on the toolbar to complete creation of the synonym.

Following is the syntax to create a synonym:

Syntax

```
CREATE SYNONYM [schema_name_1.] synonym_name FOR <object>
<object>::=
{
[server_name.[database_name] . [schema_name_2].| database_name.
[schema_name_2].| schema_name_2.] object_name
}
```

where,

schema_name_1: states that the schema in which the synonym is created.
synonym_name: specifies the new synonym name.
server_name: specifies the server name where the base object is located.
database_name: specifies the database name where the base object is located.
schema_name_2: specifies the schema name of the base object.
object_name: specifies the base object name, which is referenced by the synonym.

Code Snippet 6 creates a synonym from an existing table.

Code Snippet 6:

```
USE AdventureWorks2019;
GO
CREATE SYNONYM MyAddressType
FOR AdventureWorks2019.Person.AddressType;
GO
```

In Code Snippet 6, a synonym is created from an existing table present in the **AdventureWorks2019** database.

13.4 Program Flow Statements

There are different types of program flow statements and functions supported by Transact-SQL. Some of these are as follows:

- **Transact-SQL Control-of-Flow language**

Control-of-flow language determines the execution flow of Transact-SQL statements, statement blocks, user-defined functions, and stored procedures.

By default, Transact-SQL statements are executed sequentially, in the order they occur. Control-of-flow language elements allow statements to be executed in a particular order, to be related to each other and made interdependent using constructs similar to programming languages.

Table 13.3 lists some of the Transact-SQL control-of-flow language keywords.

Control-Of-Flow Language Keywords
RETURN
THROW
TRY....CATCH
WAITFOR
WHILE
BEGIN....END
BREAK
CONTINUE
GOTO label
IF...ELSE

Table 13.3: Keywords

- **BEGIN....END**

The BEGIN...END statements with a series of Transact-SQL statements so that a group of Transact-SQL statements is executed.

Following is the basic syntax for the BEGIN and END statement:

Syntax:

```
BEGIN
{
sql_statement | statement_block
}
END
```

where,

{sql_statement| statement_block}: Is any valid Transact-SQL statement that is defined using a statement block.

Code Snippet 7 shows the use of BEGIN and END statements.

Code Snippet 7:

```
USE AdventureWorks2019;
GO
BEGIN TRANSACTION;
GO
IF @@TRANCOUNT = 0 BEGIN
SELECT FirstName, MiddleName
FROM Person.Person WHERE LastName = 'Andy';
ROLLBACK TRANSACTION;
PRINT N'Rolling back the transaction two times would cause an error.';
END;
ROLLBACK TRANSACTION;
PRINT N'Rolled back the transaction.';
GO
```

In Code Snippet 7, BEGIN and END statements describe a sequence of Transact-SQL statements that are executed together. Suppose the BEGIN and END are not included, then, the ROLLBACK TRANSACTION statements will execute and both the PRINT messages will be displayed.

- **IF...ELSE**

The `IF...ELSE` statement enforces a condition on the execution of a Transact-SQL statement. The Transact-SQL statement is followed with the `IF` keyword and the condition executes only if the condition is satisfied and returns `TRUE`. The `ELSE` keyword is an optional Transact-SQL statement that executes only when the `IF` condition is not satisfied and returns `FALSE`.

Following is the syntax for the `IF...ELSE` statement:

Syntax:

```
IF Boolean_expression  
{sql_statement | statement_block}  
[ELSE  
{sql_statement | statement_block} ]
```

where,

`Boolean_expression`: specifies the expression that returns `TRUE` or `FALSE` value.

`{sql_statement| statement_block}`: Is any Transact-SQL statement or statement grouping that is defined by using a statement block. If a statement block is not used, the `IF` or `ELSE` condition can affect the performance of only one Transact-SQL statement. In order to define the statement block, the `BEGIN` and `END` keywords are used.

Code Snippet 8 shows the use of `IF...ELSE` statements.

Code Snippet 8:

```
USE AdventureWorks2019;  
GO  
DECLARE @ListPrice money;  
SET @ListPrice = (SELECT MAX(p.ListPrice) FROM Production.Product AS p  
JOIN Production.ProductSubcategory AS s  
ON p.ProductSubcategoryID = s.ProductSubcategoryID WHERE s.[Name] = 'Mountain  
Bikes');  
PRINT @ListPrice  
IF @ListPrice <3000  
PRINT 'All the products in this category can be purchased for an amount less  
than 3000'  
ELSE  
PRINT 'The prices for some products in this category exceed 3000'
```

In Code Snippet 8, the `IF...ELSE` statement is used to form a conditional statement. First, a variable `@ListPrice` is defined and a query is created to return the maximum list price of the product category Mountain Bikes. Then, this price is compared with a value of 3000 to determine if products can be purchased for an amount less than 3,000. If yes, an appropriate message is printed using the first `PRINT` statement. If not, then the second `PRINT` statement executes.

- **WHILE**

The `WHILE` statement specifies a condition for the repetitive execution of the statement block. The statements are executed repetitively as long as the specified condition is true. The execution of statements in the `WHILE` loop can be controlled by using the `BREAK` and `CONTINUE` keywords.

Following is the syntax for WHILE statement:

Syntax

```
WHILE Boolean_expression  
{sql_statement | statement_block | BREAK | CONTINUE}
```

where,

Boolean_expression: specifies the expression that returns TRUE or FALSE values.

{sql_statement | statement_block}: Is any Transact-SQL statement that defines the statement block.

BREAK: Results in an exit from the innermost WHILE loop. Every statement that appears after the END keyword, that marks the end of the loop, is executed.

CONTINUE: Results in the WHILE loop being restarted. The statements after the CONTINUE keyword within the body of the loop are not executed.

Code Snippet 9 shows the use of WHILE statement.

Code Snippet 9:

```
DECLARE @flag int SET @flag = 10 WHILE (@flag <=95) BEGIN  
IF @flag%2 =0 PRINT @flag  
SET @flag = @flag + 1  
CONTINUE; END  
GO
```

Using Code Snippet 9, all the even numbers beginning from 10 until 95 are displayed. This is achieved using a WHILE loop along with an IF statement. Similarly, a WHILE loop can also be used with queries and other Transact-SQL statements.

13.5 Transact-SQL Functions

The Transact-SQL functions that are commonly used are as follows:

- **Deterministic and non-deterministic functions**

User-defined functions possess properties that define the capability of the SQL Server Database Engine. The Database Engine is used to index the result of a function either through computed columns that the function calls or the indexed views that reference the functions. One such property is the determinism of a function.

Deterministic functions return the same result every time they are called with a definite set of input values and specify the same state of the database. Non-deterministic functions return different results every time they are called with specified set of input values even though the database that is accessed remains the same.

For example, if a user calls the DAY() function on a particular column, it always returns the numerical day for the date parameter passed in. However, if the user calls the DATENAME() function, the output cannot be predicted since it may be different each time, depending on what part of the date is passed as input. Thus, here, DAY() is a deterministic function, while DATENAME() is a non-deterministic function. Similarly, RAND (without any seed), @@TIMETICKS, @@CONNECTIONS, and GETDATE()

are non-deterministic.

Users cannot influence the determinism of built-in functions. Every built-in function is deterministic or non-deterministic depending on how the function is implemented by SQL Server.

Table 13.4 lists some of the deterministic and non-deterministic built-in functions.

Deterministic Built-in Functions	Non-Deterministic Built-in Functions
POWER	@@TOTAL_WRITE
ROUND	CURRENT_TIMESTAMP
RADIANS	GETDATE
EXP	GETUTCDATE
FLOOR	GET_TRANSMISSION_STATUS
SQUARE	NEWID
SQRT	NEWSEQUENTIALID
LOG	@@CONNECTIONS
YEAR	@@CPU_BUSY
ABS	@@DBTS
ASIN	@@IDLE
ACOS	@@IOBUSY
SIGN	@@PACK_RECEIVED
SIN	@@PACK_SENT

Table 13.4: Deterministic and Non-deterministic Built-in Functions

There are some functions that are not always deterministic, but you can use them in indexed views if they are given in a deterministic manner. Table 13.5 lists some of these functions.

Function	Description
CONVERT	Is deterministic only if one of these conditions exists: <ul style="list-style-type: none"> • Has an sql_variant source type. • Has an sql_variant target type and source type is non-deterministic. • Has its source or target type as smalldatetime or datetime, has the other source or target type as a character string, and has a non-deterministic style specified. The style parameter must be a constant to be deterministic.
CAST	Is deterministic only if it is used with smalldatetime, sql_variant, or datetime.
ISDATE	Is deterministic unless used with the CONVERT function, the CONVERT style parameter is specified, and style is not equal to 0, 100, 9, or 109.
CHECKSUM	Is deterministic, with the exception of CHECKSUM(*).

Table 13.5: Deterministic Functions

- **Calling Extended Stored Procedures from Functions**

Functions calling extended stored procedures are non-deterministic because the extended stored procedures may result in side effects on the database.

Changes made to the global state of a database such as a change to an external resource, or updates to a table, file, or a network are called side effects. For example, sending an e-mail, or deleting a file can cause side effects. While executing an extended stored procedure from a user-defined function, the user cannot assure that it will return a consistent resultset.

Therefore, the user-defined functions that create side effects on the database are not recommended.

- **Scalar-valued Functions**

A Scalar-valued Function (SVF) always returns an int, bit, or string value. The data type returned from and the input parameters of SVF can be of any data type except text, ntext, image, cursor, and timestamp.

An inline scalar function has a single statement and no function body. A multi-statement scalar function encloses the function body in a BEGIN...END block.

- **Table-valued Functions**

Table-valued functions are user-defined functions that return a table. Similar to an inline scalar function, an inline table-valued function has a single statement and no function body.

Code Snippet 10 shows the creation of a table-valued function.

Code Snippet 10:

```
USE AdventureWorks2019;
GO
IF OBJECT_ID ('N'Sales.ufn_CustDates', N'IF') IS NOT NULL DROP FUNCTION
Sales.ufn_ufn_CustDates;
GO
CREATE FUNCTION Sales.ufn_CustDates () RETURNS TABLE
AS RETURN (
SELECT A.CustomerID, B.DueDate, B.ShipDate FROM Sales.Customer A
LEFT OUTER JOIN
Sales.SalesOrderHeader B ON
A.CustomerID = B.CustomerID AND YEAR(B.DueDate)<2020
);
```

Here, an inline table-valued function defines a left outer join between the tables Sales.Customer and Sales.SalesOrderHeader.

Tables are joined based on customer ids. In this case, all records from the left table and only matching records from the right table are returned. The resultant table is then returned from the table-valued function.

The function is invoked as shown in Code Snippet 11.

Code Snippet 11:

```
SELECT * FROM Sales.ufn_CustDates();
```

The result will be the outcome of the join represented in a tabular format.

13.6 Altering User-defined Functions

Users can modify the user-defined functions in SQL Server 2019 by using the Transact-SQL or SSMS. Changing the user-defined functions does not modify the functions' permissions, nor will it affect any stored procedures, triggers, or functions.

- **Limitations and Restrictions**

The `ALTER FUNCTION` does not allow the users to perform the following actions:

- Modify a scalar-valued function to a table-valued function.
- Modify an inline function to a multi-statement function.
- Modify a Transact-SQL to a CLR function.

- **Permissions**

The `ALTER` permission is required on the schema or the function. If the function specifies a user-defined type, then it requires the `EXECUTE` permission on the type.

- **Modifying a User-defined function using SSMS**

Users can also modify user-defined functions using SSMS.

To modify the user-defined function using SSMS, perform the following steps:

1. Click the plus (+) symbol beside the database that contains the function to be modified.
2. Click the plus (+) symbol next to the Programmability folder.
3. Click the plus (+) symbol next to the folder, which contains the function to be modified. There are four folder types as follows:
 - Table-valued Functions
 - Scalar-valued Functions
 - Aggregate Functions
 - System Functions
4. Right-click the function to be modified and then, select **Modify**. The code for the function appears in a query editor window.
5. In the query editor window, make the required changes to the `ALTER FUNCTION` statement body.
6. Click **Execute** on the toolbar to execute the `ALTER FUNCTION` statement.

- **Modifying a User-defined function using Transact-SQL**

To modify the user-defined function using Transact-SQL, perform the following steps:

- In the **Object Explorer**, connect to the Database Engine instance.
- On the **Standard bar**, click **New Query**.
- Type the `ALTER FUNCTION` code in the **Query Editor**.

- Click **Execute** on the toolbar to execute the ALTER FUNCTION statement.

Code Snippet 12 demonstrates modifying a table-valued function.

Code Snippet 12:

```
USE AdventureWorks2019;
GO
ALTER FUNCTION [dbo].[ufnGetAccountingEndDate] () RETURNS [datetime]
AS BEGIN
RETURN DATEADD(millisecond, -2, CONVERT(datetime, '20040701', 112));
END;
```

13.7 Creation of Windows with OVER

A window function is a function that applies to a collection of rows. The word 'window' is used to refer to the collection of rows that the function works on.

In Transact-SQL, the `OVER` clause is used to define a window within a query resultset. Using windows and the `OVER` clause with functions provides several advantages. For instance, they help to calculate aggregated values. They also enable row numbers in a resultset to be generated easily.

13.7.1 Windowing Components

Three core components of creating windows with the `OVER` clause are as follows:

- **Partitioning**

Partitioning is a feature that limits the window of the recent calculation to only those rows from the resultset that contains the same values in the partition columns as in the existing row. It uses the `PARTITION BY` clause.

Code Snippet 13 demonstrates use of the `PARTITION BY` and `OVER` clauses with aggregate functions. Here, using the `OVER` clause proves to be better efficient than using subqueries to calculate the aggregate values.

Code Snippet 13:

```
USE AdventureWorks2019;
GO
SELECT SalesOrderID, ProductID, OrderQty
, SUM(OrderQty) OVER(PARTITION BY SalesOrderID) AS Total
, MAX(OrderQty) OVER(PARTITION BY SalesOrderID) AS MaxOrderQty FROM
Sales.SalesOrderDetail
WHERE ProductId IN(776, 773);
GO
```

Output of Code Snippet 13 is shown in figure 13.5.

Results Messages

	SalesOrderID	ProductID	OrderQty	Total	MaxOrderQty
1	43659	776	1	3	2
2	43659	773	2	3	2
3	43661	776	4	6	4
4	43661	773	2	6	4
5	43664	773	1	1	1
6	43665	773	1	2	1
7	43665	776	1	2	1
8	43667	773	1	1	1
9	43670	776	1	3	2
10	43670	773	2	3	2
11	43672	776	2	2	2
12	43676	776	2	2	2
13	43683	773	2	4	2
14	43683	776	2	4	2
15	43693	773	1	1	1
16	43694	776	3	5	3
17	43694	773	2	5	3

Query executed successfully.

Figure 13.5: Partitioning with OVER Clause

- **Ordering**

The ordering element defines the ordering for calculation in the partition. In a standard SQL ordering element all functions are supported. Earlier, SQL Server had no support for ordering elements with aggregate functions as it only supported partitioning. In SQL Server 2019, there is a support for the ordering element with aggregate functions. The ordering element has different meaning to some extent for different function categories. With ranking functions, ordering is spontaneous.

Code Snippet 14 demonstrates an example of the ordering element.

Code Snippet 14:

```
SELECT CustomerID, StoreID,
RANK() OVER(ORDER BY StoreID DESC) AS Rnk_All, RANK() OVER(PARTITION BY
PersonID
ORDER BY CustomerID DESC) AS Rnk_Cust
FROM Sales.Customer;
```

Code Snippet 14 makes use of the `RANK()` function which returns the rank of each row in the partition of a resultset. The rank of a row is determined by adding 1 to the number of ranks that come before the specified row. For example, while using descending ordering, the `RANK()` function returns one more than the number of rows in the respective partition that has a greater ordering value than the specified one.

Figure 13.6 displays the output of Code Snippet 14.

	CustomerID	StoreID	Rnk_All	Rnk_Cust
1	701	844	813	1
2	700	1030	633	2
3	699	842	815	3
4	698	640	1009	4
5	697	1032	631	5
6	696	840	817	6
7	695	638	1011	7
8	694	1034	629	8
9	693	838	819	9
10	692	802	855	10
11	691	1036	627	11
12	690	836	821	12
13	689	1402	278	13
14	688	1038	625	14
15	687	834	823	15
16	686	1400	279	16
17	685	1040	623	17

Figure 13.6: Ordering

Code Snippet 15 displays a query with two RANK calculations and the ORDER BY clause.

Code Snippet 15:

```
SELECT TerritoryID, Name, SalesYTD, RANK() OVER(ORDER BY SalesYTD DESC) AS Rnk_One, RANK() OVER(PARTITION BY TerritoryID ORDER BY SalesYTD DESC) AS Rnk_Two
FROM Sales.SalesTerritory;
```

Code Snippet 15 makes use of the RANK() function which returns the rank of each row in the partition of a resultset. In general, the rank of a row is determined by adding 1 to the number of ranks that come before the specified row. Here in this code, the first RANK() function generates the attribute Rnk_One that depends on the default partitioning, and the second RANK function generates Rnk_Two that uses explicit partitioning by TerritoryID.

Figure 13.7 displays the partitions defined for a sample of three results of calculations in the query: one Rnk_One value and two Rnk_Two value.

	TerritoryID	Name	SalesYTD	Rnk_One	Rnk_Two
1	1	Northwest	7887186.7882	2	1
2	2	Northeast	2402176.8476	10	1
3	3	Central	3072175.118	8	1
4	4	Southwest	10510853.8739	1	1
5	5	Southeast	2538667.2515	9	1
6	6	Canada	6771829.1376	3	1
7	7	France	4772398.3078	6	1
8	8	Germany	3805202.3478	7	1
9	9	Australia	5977814.9154	4	1
10	10	United Kingdom	5012905.3656	5	1

Figure 13.7: Partitioning and Ranking

- **Framing**

Framing is a feature that enables you to specify a further division of rows within a window partition. This is done by assigning upper and lower boundaries for the window frame that presents rows to the window function. In simple terms, a frame is similar to a moving window over the data that starts and ends at specified positions. Window frames can be defined using the ROW or RANGE subclauses and providing starting and ending boundaries.

Code Snippet 16 displays a query against the **ProductInventory**, calculating the running total quantity for each product and location.

Code Snippet 16:

```
SELECT ProductID, Shelf, Quantity,
SUM(Quantity) OVER(PARTITION BY ProductID
ORDER BY LocationID
ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS RunQty
FROM Production.ProductInventory;
```

In Code Snippet 16, the window function applies the **SUM** aggregate to the attribute **Quantity**, partitions the window by **ProductID**, orders the partition rows by **LocationID**, and frames the partition rows depending on the given ordering between unbounded preceding (no low boundary point) and the current row. In other words, the result will be the sum of all prior rows in the frame, including the current row. Figure 13.8 displays the output of Code Snippet 16.

	ProductId	Shelf	Quantity	RunQty
1	1	A	408	408
2	1	B	324	732
3	1	A	353	1085
4	2	A	427	427
5	2	B	318	745
6	2	A	364	1109
7	3	A	585	585
8	3	B	443	1028
9	3	A	324	1352
10	4	A	512	512
11	4	B	422	934
12	4	A	388	1322
13	316	A	532	532
14	316	B	388	920
15	316	B	441	1361
16	317	C	283	283
17	317	A	158	441

Query executed successfully.

Figure 13.8: Framing

13.8 Window Functions

Some of the different types of window functions are as follows:

- **Ranking functions**

These functions return a rank value for each row in a partition. Based on the function that is used, many rows will return the same value as the other rows. Ranking functions are non-deterministic.

Table 13.6 lists various ranking functions.

Ranking Functions	Description
NTILE	Spreads rows in an ordered partition into a given number of groups, beginning at 1. For each row, the function returns the number of the group to which the row belongs.
ROW NUMBER	Retrieves the sequential number of a row in a partition of a resultset, starting at 1 for the first row in each partition.
DENSE RANK	Returns the rank of rows within the partition of a resultset, without any gaps in the ranking. The rank of a row is one plus the number of distinct ranks that come before the row in question.

Table 13.6: Ranking Functions

Code Snippet 17 demonstrates the use of ranking functions.

Code Snippet 17:

```
USE AdventureWorks2019;
GO
SELECT p.FirstName, p.LastName
```

```

, ROW_NUMBER() OVER (ORDER BY a.PostalCode) AS 'Row Number'
, NTILE(4) OVER (ORDER BY a.PostalCode) AS 'NTILE'
,s.SalesYTD, a.PostalCode FROM Sales.SalesPerson AS s
INNER JOIN Person.Person AS p
ON s.BusinessEntityID = p.BusinessEntityID INNER JOIN Person.Address AS a
ON a.AddressID = p.BusinessEntityID
WHERE TerritoryID IS NOT NULL AND SalesYTD <> 0;

```

The `NTILE()` function breaks a given input collection into N equal sized logical groups. To determine how many rows belong in each group, SQL Server has to determine the total number of rows in the input collection. The `OVER` clause decides the order of the rows when they have been divided into groups. It is possible to perform the grouping in one order and return the resultset in another order.

Figure 13.9 displays the output of Code Snippet 17.

	FirstName	LastName	Row Number	NTILE	SalesYTD	PostalCode
1	Michael	Blythe	1	1	3763178.1787	98027
2	Linda	Mitchell	2	1	4251368.5497	98027
3	Jillian	Carson	3	1	3189418.3662	98027
4	Garrett	Vargas	4	1	1453719.4653	98027
5	Tsvi	Reiter	5	2	2315185.611	98027
6	Pamela	Anzman-Wolfe	6	2	1352577.1325	98027
7	Shu	Ito	7	2	2458535.6169	98055
8	José	Saraiva	8	2	2604540.7172	98055
9	David	Campbell	9	3	1573012.9383	98055
10	Tete	Mensa-Annan	10	3	1576562.1966	98055
11	Lynn	Tsoflias	11	3	1421810.9242	98055
12	Rachel	Valdez	12	4	1827066.7118	98055
13	Jae	Pak	13	4	4116871.2277	98055
14	Ranjit	Varkey Chudukatil	14	4	3121616.3202	98055

Figure 13.9: Output of Code Snippet 17

- **OFFSET functions**

Different types of offset functions are as follows:

- **SWITCHOFFSET**

This function returns a `DATETIMEOFFSET` value that is modified from the stored time zone offset to a specific new time zone offset.

Following is the syntax for the `SWITCHOFFSET` function:

Syntax

`SWITCHOFFSET (DATETIMEOFFSET, time_zone)`

where,

`DATETIMEOFFSET`: is an expression that is resolved to a `datetimeoffset(n)` value.

`time_zone`: specifies the character string in the format `[+|-]Tzh:Tzm` or a signed integer (of

minutes) which represents the time zone offset and is assumed to be daylight-saving aware and adjusted.

Code Snippet 18 displays the use of `SWITCHOFFSET` function.

Code Snippet 18:

```
CREATE TABLE Test
    ColDatetimeoffset datetimeoffset
);
GO
INSERT INTO Test
VALUES ('1998-09-20 7:45:50.71345 -5:00');
GO
SELECT SWITCHOFFSET (ColDatetimeoffset, '-08:00')
FROM Test;
GO
--Returns: 1998-09-20 04:45:50.7134500 -08:00
SELECT ColDatetimeoffset
FROM Test;
```

Figure 13.10 displays the output of Code Snippet 18.

(No column name)	
1	1998-09-20 04:45:50.7134500 -08:00
ColDatetimeoffset	
1	1998-09-20 07:45:50.7134500 -05:00

Figure 13.10: Use of SWITCHOFFSET Function

- **DATETIMEOFFSETFROMPARTS**

This function returns a `datetimeoffset` value for the specified date and time with specified precision and offset.

Following is the syntax for `DATETIMEOFFSETFROMPARTS`:

Syntax

```
DATETIMEOFFSETFROMPARTS (year, month, day, hour,
minute, seconds, fractions, hour_offset,
minute_offset, precision)
```

where,

- `year`: specifies the integer expression for a year.
- `month`: specifies the integer expression for a month.
- `day`: specifies the integer expression for a day.
- `hour`: specifies the integer expression for an hour.
- `minute`: specifies the integer expression for a minute.
- `seconds`: specifies the integer expression for a day.
- `fractions`: specifies the integer expression for fractions.

hour_offset: specifies the integer expression for the hour portion of the time zone offset.
minute_offset: specifies the integer expression for the minute portion of the time zone offset.
precision: specifies the integer literal precision of the datetimeoffset value to be returned.

Code Snippet 19 displays the use of DATETIMEOFFSETFROMPARTS function.

Code Snippet 19:

```
SELECT DATETIMEOFFSETFROMPARTS (2010, 12, 31, 14, 23, 23, 0, 12, 0, 7)
AS Result;
```

Code Snippet 19 displays a datetimeoffset value for the given date and time with the specified precision and offset.

Figure 13.11 displays the output of Code Snippet 19.

Results		Messages
	Result	
1	2010-12-31 14:23:23.0000000 +12:00	

Figure 13.11: Use of DATETIMEOFFSETFROMPARTS Function

• **SYSDATETIMEOFFSET**

These functions returns datetimeoffset(7) value which contains the date and time of the computer on which the instance of SQL Server is running.

Following is the syntax for SYSDATETIMEOFFSET:

Syntax:

```
SYSDATETIMEOFFSET () ;
```

Code Snippet 20 displays different formats used by the date and time functions.

Code Snippet 20:

```
SELECT SYSDATETIME() AS SYSDATETIME
, SYSDATETIMEOFFSET() AS SYSDATETIMEOFFSET
, SYSUTCDATETIME() AS SYSUTCDATETIME
```

Figure 13.12 displays the use of SYSDATETIMEOFFSET function.

Results		Messages	
	SYSDATETIME	SYSDATETIMEOFFSET	SYSUTCDATETIME
1	2020-10-27 21:37:55.1078981	2020-10-27 21:37:55.1078981 +05:30	2020-10-27 16:07:55.1078981

Figure 13.12: Use of SYSDATETIMEOFFSET Function

- **Analytic Functions**

SQL Server 2019 supports several analytic functions. These functions compute aggregate value based on a group of rows. Analytic functions compute running totals, moving averages, or top-N results within a group.

Table 13.7 lists some of the analytic functions.

Function	Description
LEAD	Provides access to data from a subsequent row in the same resultset without using a self-join.
LAST_VALUE	Retrieves the last value in an ordered set of values.
LAG	Provides access to data from a previous row in the same resultset without using a self-join.
FIRST_VALUE	Retrieves the first value in an ordered set of values.
CUME_DIST	Computes the cumulative distribution of a value in a group of values.
PERCENTILE_CONT	Computes a percentile based on a continuous distribution of the column value in SQL.
PERCENTILE_DISC	Calculates a particular percentile for sorted values in an entire rowset or within distinct partitions of a rowset.

Table 13.7: Analytic Functions

Code Snippet 21 demonstrates the use of `LEAD()` function.

Code Snippet 21:

```
USE AdventureWorks2019;
GO
SELECT BusinessEntityID, YEAR(QuotaDate) AS QuotaYear, SalesQuota AS NewQuota,
LEAD(SalesQuota, 1,0) OVER (ORDER BY YEAR(QuotaDate)) AS FutureQuota FROM
Sales.SalesPersonQuotaHistory
WHERE BusinessEntityID = 275 and YEAR(QuotaDate) IN ('2011','2014');
```

In Code Snippet 21, the `LEAD()` function is used to return the difference in the sales quotas for a particular employee over the subsequent years.

Code Snippet 22 demonstrates the use of `FIRST_VALUE()` function.

Code Snippet 22:

```
USE AdventureWorks2019;
GO
SELECT Name, ListPrice,
FIRST_VALUE(Name) OVER (ORDER BY ListPrice ASC) AS LessExpensive FROM
Production.Product
WHERE ProductSubcategoryID = 37
```

In Code Snippet 22, the `FIRST_VALUE()` function compares products in the product category 37 and returns the product name that is less expensive as shown in figure 13.13.

	Name	ListPrice	LessExpensive
1	Patch Kit/8 Patches	2.29	Patch Kit/8 Patches
2	Road Tire Tube	3.99	Patch Kit/8 Patches
3	Touring Tire Tube	4.99	Patch Kit/8 Patches
4	Mountain Tire Tube	4.99	Patch Kit/8 Patches
5	LL Road Tire	21.49	Patch Kit/8 Patches
6	ML Road Tire	24.99	Patch Kit/8 Patches
7	LL Mountain Tire	24.99	Patch Kit/8 Patches
8	Touring Tire	28.99	Patch Kit/8 Patches
9	ML Mountain Tire	29.99	Patch Kit/8 Patches
10	HL Road Tire	32.60	Patch Kit/8 Patches
11	HL Mountain Tire	35.00	Patch Kit/8 Patches

Figure 13.13: First Value() Function

13.9 Check Your Progress

1. Which of the following is a feature that controls the use of multiple Transact-SQL statements at one time?

(A)	Scripts	(C)	Control-of-flow
(B)	Error Handling	(D)	Variables

2. Which of the following are used to set and declare local variables provided by SQL Server?

a.	DECLARE
b.	SET
c.	DELETE
d.	INSERT

(A)	a, d	(C)	a, b
(B)	b, c	(D)	c, d

3. Which of the following is not a permission that is applied on a synonym?

(A)	GRANT	(C)	DELETE
(B)	CONTROL	(D)	UPDATE

4. Which of the following code uses a local variable to retrieve contact information for the last names starting with 'Per'?

(A)	USE AdventureWorks2019; GO DECLARE @find varchar(30); SET @find = 'Per%'; SELECT p.LastName, p.FirstName, ph.PhoneNumber FROM Person.Person AS p JOIN Person.PersonPhone AS ph ON p.BusinessEntityID = ph.BusinessEntityID WHERE LastName LIKE @find;
-----	--

(B)	<pre> USE AdventureWorks2019; GO DECLARE find varchar(30) = 'Per%'; SET find = 'Per%'; SELECT p.LastName, p.FirstName, ph.PhoneNumber FROM Person.Person AS p JOIN Person.PersonPhone AS ph ON p.BusinessEntityID = ph.BusinessEntityID WHERE LastName LIKE find;</pre>
(C)	<pre> USE AdventureWorks2019; GO @find varchar(30); @find varchar(30) = 'Per%'; SET @find = 'Per%'; SELECT p.LastName, p.FirstName, ph.PhoneNumber FROM Person.Person AS p JOIN Person.PersonPhone AS ph ON p.BusinessEntityID = ph.BusinessEntityID WHERE LastName LIKE @find;</pre>
(D)	<pre> USE AdventureWorks2019; GO SET @find varchar(30); SET @find varchar(30) = 'Per%'; SET @find = 'Per'; SELECT p.LastName, p.FirstName, ph.PhoneNumber FROM Person.Person AS p JOIN Person.PersonPhone AS ph ON p.BusinessEntityID = ph.BusinessEntityID WHERE LastName LIKE @find;</pre>

5. Which of the following is not a non-deterministic function?

(A)	@@PACK_SENT	(C)	@@DBTS
(B)	@@IOBUSY	(D)	IDLE

13.9.1 Answers

1.	A
2.	C
3.	A
4.	A
5.	D



Summary

- Transact-SQL provides basic programming elements such as variables, control-of-flow elements, conditional, and loop constructs.
- A batch is a collection of one or more Transact-SQL statements that are sent as one unit from an application to the server.
- Variables allow users to store data for using as input in other Transact-SQL statements.
- Synonyms provide a way to have an alias for a database object that may exist on a remote or local server.
- Deterministic functions always return same result each time when they are called with a definite set of input values.
- Non-deterministic functions return different results every time they are called with specified set of input values even though the database that is accessed remains the same.
- A window function is a function that applies to a collection of rows.



Try It Yourself

- Zen Technologies is a leading company in textiles located in California. The management of the company wants to give a loyalty award to all employees completing tenure of five years in the organization. Using the same Employee table, create a Transact-SQL batch to return **EmployeeID**, **FirstName**, **Department**, and **HireDate** of all such employees. Assume that the management gives twenty five percent salary increment to everybody completing one year in the organization. The chairman of the organization wants to participate in a national salary survey.

Write a batch to determine the total salary paid to a department which employs more than one employee. The management wants to find out which department has hired largest number of employees in last five years.

Hint:

- Use the `DATETIMEOFFSETFROMPARTS` function.

- Maxwel distribution is product Distribution Company and having various department across various countries you have to find the country wise report to display unique department in the company. The report should display all unique departments' country wise with the name of their leaders.

For this example, you have to create `HumanResource.Departments` table in `AdventureWorks2019` database with **DepartmentID**, **DepartmentName**, **Country**, and **DepartmentLeader**.

Session - 14

Transactions

Welcome to the Session, **Transactions**.

This session explains the types of transactions and the procedure to implement the transactions. It also describes the process to control and mark a transaction and lists the differences between the implicit and explicit transactions. The session also covers the isolation levels, scope, different types of locks, and transaction management.

In this session, you will learn to:

- Define and describe transactions
- Explain the procedure to implement transactions
- Explain the steps to mark a transaction
- Distinguish between implicit and explicit transactions
- Explain isolation levels
- Explain the scope and different types of locks
- Explain transaction management

14.1 Introduction

A transaction is a single unit of work. A transaction is successful only when all data modifications that are made in a transaction are committed and are saved in the database permanently. If the transaction is rolled back or cancelled, then it means that the transaction has encountered errors and there are no changes made to the contents of the database. Hence, a transaction can be either committed or rolled back.

14.2 Need for Transactions

There are many circumstances where users are required to make many changes to the data in more than one database tables. In many cases, the data will be inconsistent that executes the individual commands.

Suppose if the first statement executes correctly but the other statements fail then the data remains in an incorrect state.

For example, a good scenario will be the funds transfer activity in a banking system. The transfer of funds will need an `INSERT` and two `UPDATE` statements. First, the user has to increase the balance of the destination account and then, decrease the balance of the source account. The user has to check that the transactions are committed and whether the same changes are made to the source account and the destination account.

➤ Defining Transactions

A logical unit of work must exhibit four properties, called the Atomicity, Consistency, Isolation, and Durability (ACID) properties, to qualify as a transaction.

- **Atomicity:** If the transaction has many operations then all should be committed. If any of the operation in the group fails then, it should be rolled back.
- **Consistency:** The sequence of operations must be consistent.
- **Isolation:** Operations that are performed must be isolated from other operations on the same server or on the same database.
- **Durability:** The operations that are performed on the database must be saved and stored in the database permanently.

➤ Implementing Transactions

SQL Server supports transactions in several modes. Some of these modes are as follows:

- **Autocommit Transactions:** Each single-line statement is automatically committed as soon as it completes. In this mode, one does not need to write any specific statements to start and end the transactions. It is the default mode for SQL Server Database Engine.
- **Explicit Transactions:** Each transaction explicitly starts with the `BEGIN TRANSACTION` statement and ends with a `ROLLBACK` or `COMMIT` transaction.
- **Implicit Transactions:** A new transaction is automatically started when the earlier transaction completes and every transaction is explicitly completed by using the `ROLLBACK` or `COMMIT` statement.
- **Batch-scoped Transactions:** These transactions are related to Multiple Active Result Sets (MARS). Any implicit or explicit transaction that starts in a MARS session is a batch-scoped transaction.
- **Distributed Transactions:** It spans two or more servers known as resource managers. The management of the transaction must be coordinated between the resource managers by a server component called transaction manager. Each instance of the SQL Server Database Engine can operate as a resource manager in distributed transactions coordinated by transaction managers, such as Microsoft Distributed Transaction Coordinator (MS DTC), or other transaction managers that support the Open Group XA specification for distributed transaction processing.

➤ Transactions Extending Batches

The transaction statements identify the block of code that should either fail or succeed and provide the facility where the database engine can undo or roll back the operations. Errors that encounter during the execution of simple batch have the possibility of partial success, which is not a desired result. This also led to inconsistencies in the tables and databases. To overcome this, users can add code to identify the batch as a transaction and place the batch between the `BEGIN TRANSACTION` and `COMMIT TRANSACTION`. Users can add error-handling code to roll back the transaction in case of errors. The error-handling code will undo the partial changes that were made before the error had occurred. This way, inconsistencies in the tables and databases can be prevented.

14.3 Controlling Transactions

Transactions can be controlled through applications by specifying the beginning and ending of a transaction. This is done by using the database API functions or Transact-SQL statements.

Transactions are managed at the connection level, by default. When a transaction is started on a connection, all Transact-SQL statements are executed on the same connection and are a part of the connection until the transaction ends.

14.3.1 Starting and Ending Transactions Using Transact-SQL

One of the ways users can start and end transactions is by using Transact-SQL statements. Users can start a transaction in SQL Server in the implicit or explicit modes. Explicit transaction mode starts a transaction

by using a BEGIN TRANSACTION statement. Users can end a transaction using the ROLLBACK or COMMIT statements.

Following are some of the types of transaction statements:

➤ BEGIN TRANSACTION

The BEGIN TRANSACTION statement marks the beginning point of an explicit or local transaction.

Following is the syntax for the BEGIN TRANSACTION statement:

Syntax:

```
BEGIN {TRAN | TRANSACTION}
[ {transaction_name | @tran_name_variable} [WITH MARK['description']] ]
[ ; ]
```

where,

`transaction_name`: specifies the name that is assigned to the transaction. It should follow the rules for identifiers and limit the identifiers that are 32 characters long.

`@tran_name_variable`: specifies the name of a user-defined variable that contains a valid transaction name.

`WITH MARK['description']`: specifies the transaction that is marked in the log. The description string defines the mark.

Code Snippet 1 shows how to create and begin a transaction.

Code Snippet 1:

```
USE AdventureWorks2019;
GO
DECLARE @TranName VARCHAR(30);
SELECT @TranName = 'FirstTransaction';
BEGIN TRANSACTION @TranName;
DELETE FROM HumanResources.JobCandidate WHERE JobCandidateID = 13;
```

In Code Snippet 1, a transaction name is declared using a variable with value `FirstTransaction`. A new transaction with this name is then created having a `DELETE` statement. As the transaction comprises a single-line statement, it is implicitly committed.

➤ COMMIT TRANSACTION

The COMMIT TRANSACTION statement marks an end of a successful implicit or explicit transaction. If the `@@TRANCOUNT` is 1, then, COMMIT TRANSACTION performs all data modifications performed on the database and becomes a permanent part of the database. Further, it releases the resources held by the transaction and decrements `@@TRANCOUNT` by 0. If `@@TRANCOUNT` is greater than 1, then the COMMIT TRANSACTION decrements the `@@TRANCOUNT` by 1 and keeps the transaction in active state.

Following is the syntax for the COMMIT TRANSACTION statement:

Syntax:

```
COMMIT {TRAN | TRANSACTION} [transaction_name | @tran_name_variable]
] [ ; ]
```

where,

`transaction_name`: specifies the name that is assigned by the previous BEGIN TRANSACTION statement. It should follow the rules for identifiers and do not allow identifiers that are 32 characters long.

`@tran_name_variable`: specifies the name of a user-defined variable that contains a valid transaction name. The variable can be declared as `char`, `varchar`, `nchar`, or `nvarchar` data type. If more than 32 characters are passed to the variable, then only 32 characters are used and the remaining characters will be truncated.

Code Snippet 2 shows how to commit a transaction in the `HumanResources.JobCandidate` table of `AdventureWorks2019` database.

Code Snippet 2:

```
BEGIN TRANSACTION;
GO
DELETE FROM HumanResources.JobCandidate WHERE JobCandidateID = 11;
GO
COMMIT TRANSACTION;
GO
```

Code Snippet 2 defines a transaction that will delete a job candidate record having `JobCandidateID` as 11.

➤ COMMIT WORK

The `COMMIT WORK` statement marks the end of a transaction. Following is the syntax for the `COMMIT WORK` statement:

Syntax

```
COMMIT [WORK] [ ; ]
```

`COMMIT TRANSACTION` and `COMMIT WORK` are identical except for the fact that `COMMIT TRANSACTION` accepts a user-defined transaction name.

➤ Marking a Transaction

Code Snippet 3 shows how to mark a transaction in the `HumanResources.JobCandidate` table of `AdventureWorks2019` database.

Code Snippet 3:

```
BEGIN TRANSACTION DeleteCandidate
WITH MARK N'Deleting a Job Candidate';
GO
DELETE FROM HumanResources.JobCandidate WHERE JobCandidateID = 11;
GO
COMMIT TRANSACTION DeleteCandidate;
```

In Code Snippet 3, a transaction named `DeleteCandidate` is created and marked in the log.

➤ ROLLBACK TRANSACTION

This transaction rolls back or cancels an implicit or explicit transaction to the starting point of the transaction, or to a savepoint in a transaction. A savepoint is a mechanism to roll back some parts of transaction. The `ROLLBACK TRANSACTION` is used to delete all data modifications made from the

beginning of the transaction or to a savepoint. It also releases the resources held by the transaction.

Following is the syntax for the ROLLBACK TRANSACTION statement:

Syntax

```
ROLLBACK {TRAN | TRANSACTION}
[transaction_name | @tran_name_variable
| savepoint_name | @savepoint_variable]
[ ; ]
```

where,

- transaction_name: specifies the name that is assigned to the BEGIN TRANSACTION statement.
- @tran_name_variable: specifies the name of a user-defined variable that contains a valid transaction name. The variable can be declared as char, varchar, nchar, or nvarchar data type.
- savepoint_name: specifies the savepoint_name from a SAVE TRANSACTION statement. Use savepoint_name only when a conditional roll back affects a part of a transaction.
- @savepoint_variable: specifies the name of savepoint variable that contain a valid savepoint name. The variable can be declared as char, varchar, nchar, or nvarchar data type.

Consider an example that demonstrates the use of ROLLBACK. Assume that a database named Sterling has been created. A table named **ValueTable** is created in this database as shown in Code Snippet 4.

Code Snippet 4:

```
USE Sterling;
GO
CREATE TABLE ValueTable ([value] char)
GO
```

Code Snippet 5 creates a transaction that inserts two records into **ValueTable**. Then, it rolls back the transaction and again inserts one record into **ValueTable**. When a SELECT statement is used to query the table, you will see that only a single record with value C is displayed. This is because the earlier INSERT operations have been rolled back or cancelled.

Code Snippet 5:

```
BEGIN TRANSACTION
INSERT INTO ValueTable VALUES('A'); INSERT INTO ValueTable VALUES('B');
GO
ROLLBACK TRANSACTION
INSERT INTO ValueTable VALUES('C');
SELECT [value] FROM ValueTable;
```

➤ ROLLBACK WORK

This statement rolls back a user-specified transaction to the beginning of the transaction. Following is the syntax for the ROLLBACK WORK statement:

Syntax

```
ROLLBACK [WORK] [ ; ]
```

The keyword WORK is optional and is rarely used.

Figure 14.1 displays the working of transactions.

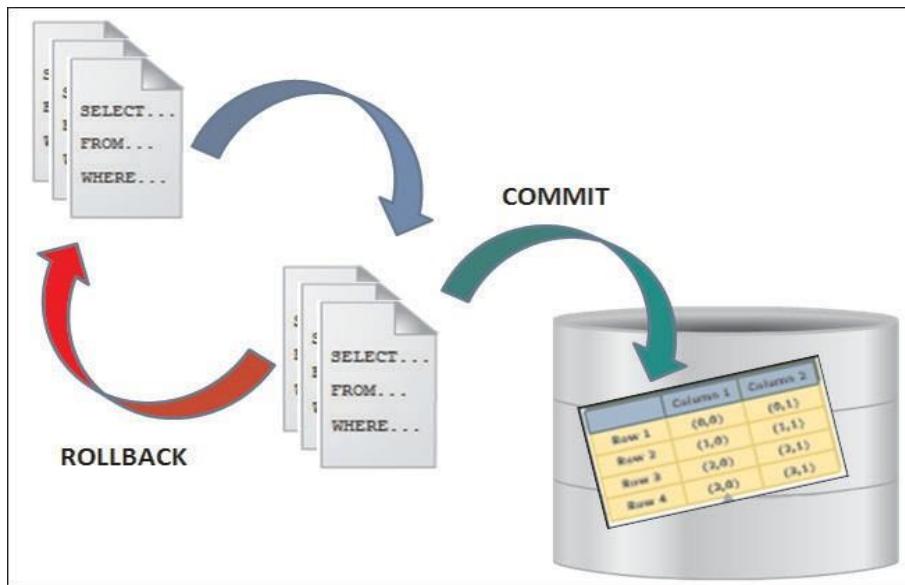


Figure 14.1: Working of Transactions

➤ **SAVE TRANSACTION**

The `SAVE TRANSACTION` statement sets a savepoint within a transaction. Following is the syntax for the `SAVE TRANSACTION` statement:

Syntax

```
SAVE {TRAN | TRANSACTION} {savepoint_name | @savepoint_variable} [ ; ]
```

where,

`savepoint_name`: specifies the savepoint_name assigned.

`@savepoint_variable`: specifies the name of a user-defined variable that contain a valid savepoint name. The variable can be declared as `char`, `varchar`, `nchar`, or `nvarchar` data type.

Code Snippet 6 shows how to use a savepoint transaction.

Code Snippet 6:

```
CREATE PROCEDURE SaveTranExample @InputCandidateID INT
AS
DECLARE @TranCounter INT;
SET @TranCounter = @@TRANCOUNT; IF @TranCounter > 0
SAVE TRANSACTION ProcedureSave;
ELSE
BEGIN TRANSACTION;
DELETE HumanResources.JobCandidate
WHERE JobCandidateID = @InputCandidateID; IF @TranCounter = 0
COMMIT TRANSACTION;
IF @TranCounter = 1
ROLLBACK TRANSACTION ProcedureSave;
GO
```

In Code Snippet 6, a savepoint transaction is created within a stored procedure. This will then be used to roll back only the changes made by the stored procedure if an active transaction has started before the stored procedure executes.

14.4 The @@TRANCOUNT

The @@TRANCOUNT system function returns a number of BEGIN TRANSACTION statements that occur in the current connection. Figure 14.2 displays an example of using @@TRANCOUNT.

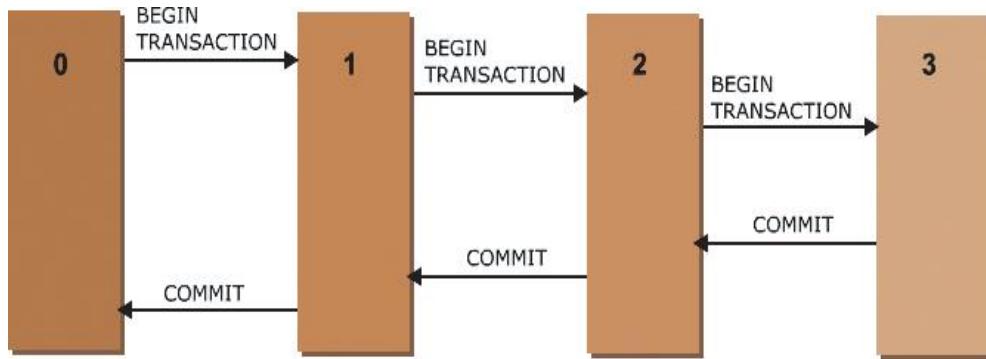


Figure 14.2: @@TRANCOUNT

Following is the syntax for the @@TRANCOUNT statement:

Syntax

```
@@TRANCOUNT
```

Code Snippet 7 shows the effect that nested BEGIN and COMMIT statements have on the @@TRANCOUNT variable.

Code Snippet 7:

```
PRINT @@TRANCOUNT BEGIN TRAN  
PRINT @@TRANCOUNT BEGIN TRAN  
PRINT @@TRANCOUNT COMMIT  
PRINT @@TRANCOUNT COMMIT  
PRINT @@TRANCOUNT
```

Code Snippet 7 displays the number of times the BEGIN TRAN and COMMIT statement execute in the current connection.

Figure 14.3 displays the output of Code Snippet 7.

```

0
1
2
1
0

Completion time: 2020-10-29T18:54:16.1988295+05:30

```

Figure 14.3: Output of Code Snippet 7

Code Snippet 8 shows the effect that nested BEGIN and ROLLBACK statements have on the @@TRANCOUNT variable.

Code Snippet 8:

```

PRINT @@TRANCOUNT BEGIN TRAN
PRINT @@TRANCOUNT BEGIN TRAN
PRINT @@TRANCOUNT
ROLLBACK
PRINT @@TRANCOUNT

```

In this case, code snippet 8 displays the number of times the BEGIN and ROLLBACK statements execute in the current connection.

Figure 14.4 displays the output of Code Snippet 8.

```

0
1
2
0
0

Completion time: 2020-10-29T19:26:12.5683198+05:30

```

Figure 14.4: Output of Code Snippet 8

14.5 Marking a Transaction

Users can use transaction marks to recover the related updates made to two or more related databases. Though this recovery loses every transaction that is committed after the mark was used as the recovery point. Marking a transaction is useful only when the user is willing to lose recently committed transactions or is testing related databases.

Marking related transactions on a routine basis in every single related database creates a sequence of common recovery points in a database. The transaction marks are incorporated in log backups and are also recorded in the transaction log. In case of any disaster, user can restore each of the databases to the same transaction mark in order to recover them to a consistent point.

➤ **Concerns for Using Marked Transactions**

Consider the following situations before inserting the named marks in the transaction:

- As the transaction mark consume log space, use them only for transactions that play an important

- role in the database recovery strategy.
- When the marked transaction is committed, then a row is inserted in the `logmarkhistory` table in `msdb`.
- If a marked transaction spans over multiple databases on different servers or on the same database server, the marks must be logged in the records of all affected databases.

14.5.1 Create Marked Transactions

For creating a marked transaction, users can use the `BEGIN TRANSACTION` statement and the `WITH MARK [description]` clause. The optional description is a textual description of the mark. A mark name for the transaction is reused and required. The transaction log records the mark description, name, user, database, datetime information, and the Log Sequence Number (LSN). The datetime information is used with the mark name for unique identification of the mark.

For creating a marked transaction in a set of databases, the following steps are required:

1. Name the transaction in the `BEGIN TRAN` statement and use the `WITH MARK` clause.
2. Execute an update against all of the databases in the set.

Code Snippet 9 updates the list price in the `Product` table of the `AdventureWorks2019` database.

Code Snippet 9:

```
USE AdventureWorks2019;
GO
BEGIN TRANSACTION ListPriceUpdate
WITH MARK 'UPDATE Product list prices';
GO
UPDATE Production.Product
SET ListPrice = ListPrice * 1.20 WHERE ProductNumber LIKE 'BK-%';
GO
COMMIT TRANSACTION ListPriceUpdate;
GO
```

14.6 Differences Between Implicit and Explicit Transactions

Table 14.1 lists the differences between implicit and explicit transactions.

Implicit	Explicit
These transactions are maintained by SQL Server for each and every DML and DDL statements.	These transactions are defined by programmers.
These DML and DDL statements execute under the implicit transactions.	DML statements are included to execute as a unit.
SQL Server will roll back the entire statement.	SELECT Statements are not included as they do not modify data.

Table 14.1: Implicit Vs. Explicit Transactions

14.7 Isolation Levels

Transactions identify the isolation levels that define the degree to which one transaction must be isolated from the data modifications or resource that are made by other transactions. Isolation levels are defined in terms of which the concurrency side effects such as dirty reads are allowed. When one transaction changes a value and a second transaction reads the same value before the original change has been committed or rolled back, it is called as a dirty read.

Transaction isolation levels control the following:

- When data is read, are there any locks taken and what types of locks are requested?
- How much amount of time the read locks are held?
- If a read operation that is referencing a row modified by some other transaction is:
 - Blocking until the exclusive lock on the row is free.
 - Retrieving the committed version of the row that exists at the time when the transaction or statement started.
 - Reading the uncommitted data modification.

While choosing a transaction isolation level, those locks that prevent data modification are not affected. A transaction acquires an exclusive lock every time on each data that it modifies. Then, it holds that lock until the transaction is completed, irrespective of the isolation level that is set for that transaction.

Transaction isolation levels mainly describe the protection levels from the special effects of changes made by other transactions for read operations. A lower isolation level increases the capability of several users to access data at the same time. However, it increases the number of concurrency effects such as dirty reads or lost updates that users might come across. On the other hand, a higher isolation level decreases the types of concurrency effects which user may encounter. This requires additional system resources and increases the chance of one transaction blocking another transaction.

Selecting a suitable isolation level is based on the data integrity requirements of the application as compared to the overheads of each isolation level. The higher isolation level, serializable, assures that a transaction will recover the same data each time it repeats the read operation. Then, it does this by performing a level of locking that is expected to influence other users in a multi-user system. The lower isolation levels retrieves data that is modified and is not committed by other transactions. All concurrency side effects occur in read uncommitted, however, there is no read versioning or locking, hence, the overhead is minimized.

Table 14.2 lists the concurrency effects that are allowed by different isolation levels.

Isolation Level	Dirty Read	NonRepeatable Read
Read committed	No	Yes
Read uncommitted	Yes	No
Snapshot	No	No
Repeatable Read	No	No
Serializable	No	No

Table 14.2: Isolation Levels

Transactions need to execute at an isolation level of at least repeatable read that prevents lost updates occurring when two transactions each retrieve the same row and then, updates the row that is dependent

on the originally retrieved rows.

14.8 Scope and Different Types of Locks

The SQL Server Database Engine locks the resources that use different lock modes, which determine the resources that are accessible to concurrent transactions.

Table 14.3 lists the resource lock modes used by the Database Engine.

Lock Mode	Description
Update	Is used on resources that are to be updated.
Shared	Is used for read operations that do not change data such as SELECT statement.
Intent	Is used to establish a hierarchy of locks.
Exclusive	Is used for INSERT, UPDATE, or DELETE data-modification operations.
BULK UPDATE	Is used while copying bulk data into the table.
Schema	Is used when the operation is dependent on the table schema.

Table 14.3: Lock Modes

Different types of locks are as follows:

➤ Update Locks

These locks avoid common forms of deadlock. In a serializable transaction, the transaction will read data, acquire a shared lock on the row or a page, and modify the data that requires lock conversion to an exclusive lock.

When two transactions acquire a shared lock on a resource and try to update data simultaneously, the same transaction attempts the lock conversion to an exclusive lock. The shared mode to exclusive lock conversion should wait as the exclusive lock for one transaction is not compatible with shared mode lock of the other transaction a lock wait occurs. Similarly, the second transaction tries to acquire an exclusive lock for update. As both the transactions are converting to exclusive locks and each waits for the other transaction to release its shared lock mode, a deadlock occurs.

Figure 14.5 depicts the concept of such a deadlock.

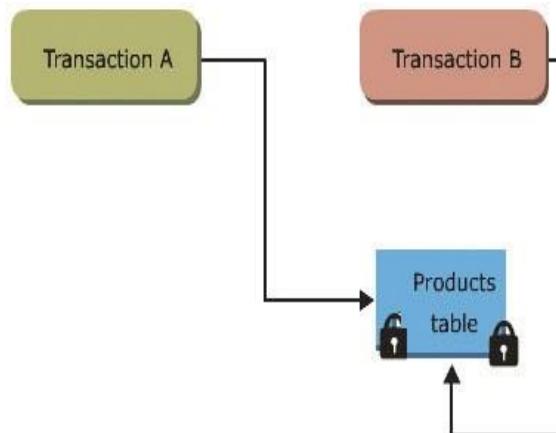


Figure 14.5: Deadlock

To avoid this deadlock, update locks are used. Only one transaction can obtain an update lock to a resource at a time. When a transaction modifies a resource, then the update lock is converted to an exclusive lock.

➤ Shared Locks

These locks allow parallel transactions to read a resource under pessimistic concurrency control. Transactions can change the data while shared locks exist on the resource. Shared locks are released on a resource once the read operation is completed, except the isolation level is set to repeatable read or higher.

➤ Exclusive Locks

These locks prevent access to resources by concurrent transactions. By using an exclusive lock, no other transaction can change data and read operations take place only through the read uncommitted isolation level or NOLOCK hint. DML statements such as INSERT, DELETE, and UPDATE combine modification and read operations. These statements first perform a read operation to get data before modifying the statements. DML statements usually request both exclusive and shared locks. For example, if the user wants to use an UPDATE statement that modifies the row in one table that is dependent on a join with other table. Therefore, the update statements request shared lock on the rows that reads from the join table and request exclusive locks on the modified rows.

➤ Intent Locks

The Database Engine uses intent locks for protecting and places an exclusive or shared lock on the resource that is at a lower level in the lock hierarchy. The name intent locks is given because they are acquired before a lock at the low level and hence, indicate intent to place locks at low level. An intent lock is useful for two purposes:

- To prevent other transactions from changing the higher-level resource in a way that will invalidate the lock at the lower-level.
- To improve the efficiency of the Database Engine for identifying the lock conflicts those are at the higher level of granularity.

For example, a shared intent locks are requested at the table level before requesting the shared locks on rows or pages within the table. Setting the intent lock at the table level protects other transaction from subsequently acquiring an exclusive lock on the table containing pages. Intent locks also contain Intent Exclusive (IX), Intent Shared (IS), and Shared with Intent Exclusive (SIX). Table 14.4 lists the lock modes in Intent locks.

Lock Mode	Description
Intent shared (IS)	Protects the requested shared lock on some resources that are lower in the hierarchy.
Intent exclusive (IX)	Protects the requested exclusive lock on some resources lower in the hierarchy. IX is a superset of IS, that protects requesting shared locks on lower level resources.
Shared with Intent Exclusive (SIX)	Protects the requested shared lock on all resources lower in the hierarchy and intent exclusive locks on some of the lower level resources. Concurrent IS locks are allowed at the top-level resource.
Intent Update (IU)	Protects the requested update locks on all resources lower in the hierarchy. IU locks are used only on page resources. IU locks are converted to IX locks if an update operation takes place.
Shared intent update (SIU)	Provides combination of S and IU locks, as a result of acquiring these locks separately and simultaneously holding both locks.

Lock Mode	Description
Update intent exclusive (UIX)	Provides combination of U and IX locks, as a result of acquiring these locks separately and simultaneously holding both locks.

Table 14.4: Lock Modes in Intent Locks

➤ Bulk Update locks

Bulk update locks are used by the database engine. These locks are used when a large amount of data is copied into a table (bulk copy operations) and either the `table lock on bulk load` option is set or the `TABLOCK` hint is specified using the `sp_tableoption`.

These locks allow multiple threads to load bulk data continuously in the same table however, preventing other processes that are not bulk loading data from accessing the table.

➤ Schema Locks

Schema modification locks are used by Database Engine while performing a table DDL operation such as dropping a table or a column. Schema locks prevent concurrent access to the table, which means a schema lock blocks all external operations until the lock releases.

Some DML operations such as truncating a table use the Schema lock to prevent access to affected tables by concurrent operations.

Schema stability locks are used by the database engine while compiling and executing the queries. These stability locks do not block any of the transaction locks including the exclusive locks. Hence, the transactions that include X locks on the table continue to execute during the query compilation. Though, the concurrent DML and DDL operations that acquire the Schema modification locks do not perform on the tables.

➤ Key-Range Locks

These types of locks protect a collection of rows that are implicitly present in a recordset which is being read by a Transact-SQL statement while using the serializable transaction isolation level. Key-range locks prevent phantom reads. By protecting the range of keys between rows, they also prevent the phantom deletions or insertions in the recordset that accesses a transaction.

14.9 Transaction Management

Each single statement that is executed is, by default, transactional in SQL Server. If a single SQL statement is issued, then, an implicit transaction is started. It means the statement will start and complete implicitly. When users use explicit `BEGIN TRAN/COMMIT TRAN` commands, they can group them together as an explicit transaction. These statements will either succeed or fail. SQL Server implements several transaction isolation levels that ensure the `ACID` properties of these transactions. In reality, it means that it uses locks to facilitate transactional access to shared database resources and also, prevent the interference between the transactions.

14.10 Transaction Log

Each SQL Server database has a transaction log, which records all transactions and the database modifications made by every transaction. The transaction log should be truncated regularly to keep it from filling up. Monitoring log size is important as there may be some factors that delay the log truncation.

Transaction log is a critical component of the database and if a system failure occurs, the transaction log will be required to bring the database to a consistent data. The transaction log should not be moved or deleted until users understand the consequences of doing it. Operations supported by the transaction log are as follows:

- Individual transactions recovery
- Incomplete transactions recovery when SQL Server starts
- Transactional replication support
- Disaster recovery solutions and high availability support
- Roll back a file, restored database, filegroup, or page forward to the point of failure

➤ Truncating a Transaction Log

Truncating a log frees the space in the log file for reusing the transaction log. Truncation of logs starts automatically after the following events:

- In a simple recovery model after the checkpoint.
- In a bulk-logged recovery model or full recovery model, if the checkpoint is occurred ever since the last backup, truncation occurs after a log backup.

There are factors that delay a log truncation. When the log records remain active for a long time, transaction log truncation is late and the transaction log fills up. Log truncations are delayed due to many reasons. Users can also discover if anything prevents the log truncation by querying the `log_reuse_wait_desc` and `log_reuse_wait` columns of the `sys.databases` catalog view.

Table 14.5 lists the values of some of these columns.

Log_reuse_wait	Log_reuse_wait desc	Description
0	NOTHING	Specifies that at present, there are more than one reusable virtual log file.
1	CHECKPOINT	Specifies that there is no checkpoint occurred since the last log truncation or the head of the log has not moved beyond a virtual log file.
2	LOG_BACKUP	Specifies a log backup that is required before the transaction log truncates.
3	ACTIVE_BACKUP_OR_RESTORE	Specifies that the data backup or a restore is in progress.
4	ACTIVE_TRANSACTION	Specifies that a transaction is active.
5	DATABASE_MIRRORING	Specifies that the database mirroring is paused or under high-performance mode, the mirror database is significantly behind the principal database.

Table 14.5: Catalog View Columns

14.11 Check Your Progress

1. Which of the following types of transaction is related to Multiple Active Result Sets?

(A))	Autocommit	(C)	Implicit
(B)	Explicit	(D))	Batch-scoped

2. _____ marks the beginning point of an explicit or local transaction.

(A))	ROLLBACK TRANSACTION	(C)	COMMIT WORK
(B)	BEGIN TRANSACTION	(D))	COMMIT TRANSACTION

3. Identify the function that returns a number of BEGIN TRANSACTION statements that occur in the current connection.

(A))	@@TRANCOUNTER	(C)	@@TRANCOUNT
(B)	@@ERRORMESSAGE	(D))	@@ERROR

4. Which of the following is not the concurrency effect allowed by different isolation levels?

(A))	Read committed	(C)	Repeatable Read
(B)	Snapshot	(D))	COMMIT

5. Match the description with corresponding Lock Modes.

	Description		Lock Modes
a.	Is used on resources that are to be updated.	1.	Schema
b.	Is used for read operations that do not change data such as SELECT statement.	2.	Exclusive
c.	Is used to establish a hierarchy of locks.	3.	Intent
d.	Is used for INSERT, UPDATE, or DELETE data-modification operations.	4.	Shared
e.	Is used when the operation is dependent on the table schema.	5.	Update

(A))	a-1, b-4, c-2, d-3, e-5	(C)	a-2, b-4, c-3, d-5, e-1
(B)	a-5, b-4, c-3, d-2, e-1	(D))	a-1, b-2, c-3, d-4, e-5

14.11.1 Answers

1.	D
2.	B
3.	C
4.	D
5.	B



Summary

- A transaction is a sequence of operations that works as a single unit.
- BEGIN TRANSACTION marks the beginning point of an explicit or local transaction.
- COMMIT TRANSACTION marks an end of a successful implicit or explicit transaction.
- ROLLBACK with an optional keyword WORK rolls back a user-specified transaction to the beginning of the transaction.
- @@TRANCOUNT is a system function that returns a number of BEGIN TRANSACTION statements that occur in the current connection.
- Isolation levels are provided by the transaction to describe the extent to which a single transaction must be isolated from changes made by other transactions.
- The SQL Server Database Engine locks the resources using different lock modes, which determine the resources that are accessible to concurrent transactions.

Try It Yourself

- Zamora Electronics Ltd. employs more than 500 workers in its units. Some of these are at junior level while some are at senior level depending upon their expertise and years of experience. Each employee is given annual leave based on the designation. The management at Zamora Electronics Ltd. is planning to computerize their human resources department and all the data pertaining to employees will now be stored in an SQL Server database. The company has made some changes in the leave policy of the employees and wants to update the same in their tables. Assume that you are the database administrator and that you are assigned the following tasks:
 - Create a transaction to update the records in the table as per the new leave policy.
 - Check if the transactions are updated in the appropriate table.
 - Check if the transactions are not updated. Then, ensure that they are rolled back with the appropriate error messages.

Table 14.6 lists the **EmployeeDetails** table.

Field Name	Data Type	Key Field	Description
Employee_Id	varchar(5)	Primary Key	Stores employee identification number
FirstName	varchar(30)		Stores first name of the employee
LastName	varchar(30)		Stores last name of the employee
Address	varchar(60)		Stores address of the employee
PhoneNumber	varchar(20)		Stores phone number of the employee, it could be landline or mobile
Department_Id	varchar(4)		Stores department id of the department to which the employee belongs
Designation	varchar(30)		Stores designation or job role of the employee
Salary	money		Stores salary of the employee
Join_date	datetime		Stores date of joining for the employee
Performance_Rating	int		Stores rating of the employee

Table 14.6: EmployeeDetails Table

Session - 15

Error Handling

Welcome to the Session, **Error Handling**.

This session introduces error-handling techniques in SQL Server and describes the use of TRY-CATCH blocks. Various system functions and statements that can help display error information are also covered in the session.

In this session, you will learn to:

- Explain error handling and its implementation
- Describe the TRY-CATCH block
- Explain the procedure to display error information
- Describe the @@ERROR and RAISERROR statements
- Explain the use of ERROR_STATE, ERROR_SEVERITY, and ERROR_PROCEDURE
- Explain the use of ERROR_NUMBER, ERROR_MESSAGE, and ERROR_LINE
- Describe the THROW statement

15.1 Introduction

Error handling in SQL Server has become easy through a number of different techniques. SQL Server has introduced options that can help you to handle errors efficiently. Often, it is not possible to capture errors that occur at the user's end. SQL Server provides the TRY...CATCHstatement that helps to handle errors effectively at the back end. There are a number of system functions that print error related information, which can help fix errors easily.

15.2 Types of Errors

As a Transact-SQL programmer, one must be aware of various types of errors that can occur while working with SQL Server statements. The first step one can perform is to identify the type of the error and then, determine how to handle or overcome it.

Some of the types of errors are as follows:

➤ **Syntax Errors**

Syntax errors are the errors that occur when code cannot be parsed by SQL Server. Such errors are detected by SQL Server before beginning the execution process of a Transact-SQL block or stored procedure.

Some scenarios where syntax errors occur are as follows:

- If a user is typing an operator or a keyword is used in a wrong way, the code editor will display the tooltip showing the error. Figure 15.1 displays an example of syntax error.

The screenshot shows a SQL query window titled "SQLQuery1.sql - DE...D6\Lenovo pc (53)*". The code entered is:

```
Use AdventureWorks2019;
GO
DECLARE @Number SET;
GO
```

A tooltip box appears at the end of the word "SET" with the text "Incorrect syntax near 'SET'".

Figure 15.1: Syntax Error

In figure 15.1, the SEToperator is wrongly used in the Transact-SQL statement, hence a syntax error will be raised.

- If a user types a keyword or an operator wrongly because the user does not remember the valid usage, the code editor will appropriately indicate it.

Figure 15.2 displays an example.

The screenshot shows a SQL query window titled "SQLQuery1.sql - DE...D6\Lenovo pc (53)*". The code entered is:

```
Us AdventureWorks2019;
```

A tooltip box appears at the start of the word "Us" with the text "Could not find stored procedure 'Us'".

Figure 15.2: Wrong Keyword Error

- If the user forgets to type something that is required, the code editor will display an error once the user executes that statement.

Syntax errors are easily identified as the code editor points them out. Therefore, these errors can be easily fixed. However, if users use a command-based application such as sqlcmd, the error is shown only after the code is executed.

➤ Run-time Errors

Run-time errors are errors that occur when the application tries to perform an action that is supported neither by SQL Server nor by the operating system. Run-time errors are sometimes difficult to fix as they are not clearly identified or are external to the database.

Some instances where run-time errors can occur are as follows:

- Performing a calculation such as division by 0
- Trying to execute code that is not defined clearly

Figure 15.3 shows the divide by zero error.

The screenshot shows a SQL query window titled "SQLQuery1.sql - DE...D6\Lenovo pc (53)*". The code is:

```
Use AdventureWorks2019;
GO
DECLARE @Number1 int;
SET @Number1=217/0;
GO
```

The "Messages" pane displays an error message:

```
Msg 8134, Level 16, State 1, Line 4
Divide by zero error encountered.
```

Completion time: 2020-10-30T07:23:07.6777505+05:30

Figure 15.3: Divide by Zero Error

Here, the code editor will not show any error before execution because there is no syntax error.

Figure 15.4 displays an example that tries to store a value in the variable that does not meet the specified range. In such a case, an arithmetic overflow error occurs.

The screenshot shows a SQL query window titled "SQLQuery1.sql - DE...D6\Lenovo pc (53)*". The code is:

```
Use AdventureWorks2019;
GO
DECLARE @Number1 tinyint;
SET @Number1=2260;
GO
```

The "Messages" pane displays an error message:

```
Msg 220, Level 16, State 2, Line 4
Arithmetic overflow error for data type tinyint, value = 2260.
```

Completion time: 2020-10-30T07:25:46.0855512+05:30

Figure 15.4: Arithmetic Overflow Error

Following are some common situations when run-time error occurs:

- Using a stored procedure, or a function, or a trigger that is unavailable
- Trying to perform an action that an object or a variable cannot handle
- Attempting to access or use computer memory that is insufficient
- Trying to perform an action on incompatible types
- Using wrongly conditional statements

15.3 Implementing Error Handling

While developing any application, one of the most important things that users need to take care of is error handling. In the same way, users also have to take care of handling exception and errors while designing the database. Various error handling mechanisms can be used. Some of them are as follows:

- When executing some DML statements such as `INSERT`, `DELETE`, and `UPDATE`, users can handle errors to ensure correct output.
- When a transaction fails and the user must roll back the transaction, an appropriate error message can be displayed.
- When working with cursors in SQL Server, users can handle errors to ensure correct results.

15.4 TRY...CATCH

TRY...CATCH statements are used to implement exception handling in Transact-SQL. One or more Transact-SQL statements can be enclosed within a TRY block. If an error occurs in the TRY block, the control is passed to the CATCH block that may contain one or more statements.

Figure 15.5 illustrates the TRY...CATCH logic.

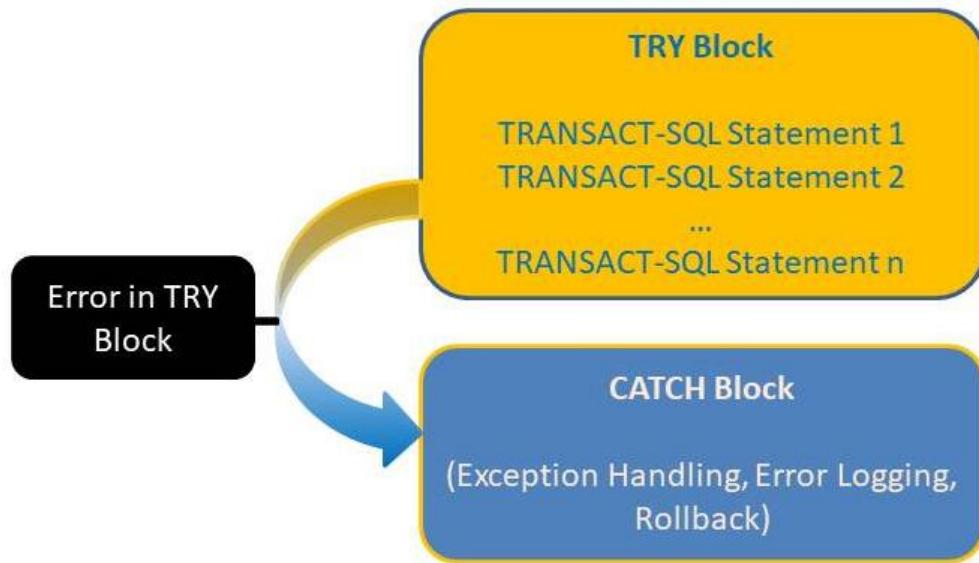


Figure 15.5: TRY...CATCH Logic

Following is the syntax for TRY...CATCH statements:

Syntax:

```
BEGIN TRY
    { sql_statement | statement_block }
END TRY
BEGIN CATCH
    [ { sql_statement | statement_block } ]
END CATCH
[ ; ]
```

where,

`sql_statement`: specifies any Transact-SQL statement.

`statement_block`: specifies the group of Transact-SQL statements in a BEGIN...END block.

A TRY...CATCH construct will catch all run-time errors that have severity higher than 10 and that do not close the database connection. A TRYblock is followed by a related CATCHblock. A TRY...CATCHblock cannot span multiple batches or multiple blocks of Transact-SQL statements.

If there are no errors in the TRY block, after the last statement in the TRYblock has executed, control is passed to the next statement following the END...CATCHstatement. If there is an error in the TRY block, control is passed to the first statement inside the CATCH block. If END...CATCH is the last statement in a trigger or a stored procedure, control is passed back to the calling block.

Code Snippet 1 demonstrates a simple example of TRY...CATCH statements.

Code Snippet 1:

```
BEGIN TRY
DECLARE @num int;
SELECT @num=217/0;
END TRY
BEGIN CATCH
PRINT 'Error occurred, unable to divide by 0'
END CATCH;
```

In this code, an attempt is made to divide a number by zero. This will cause an error, hence, the TRY...CATCH statement is used here to handle the error.

Both TRY and CATCH blocks can contain nested TRY...CATCH constructs. For example, a CATCH block can have an embedded TRY...CATCH construct to handle errors faced by the CATCH code. Errors that are encountered in a CATCH block are treated just like errors that are generated elsewhere. If the CATCH block encloses a nested TRY...CATCH construct, any error in the nested TRY block passes the control to the nested CATCH block. If there is no nested TRY...CATCH construct the error is passed back to the caller.

TRY...CATCH constructs can also catch unhandled errors from triggers or stored procedures that execute through the code in TRY block. However, as an alternative approach, triggers or stored procedures can also enclose their own TRY...CATCH constructs to handle errors generated through their code.

Note: GOTO statements can be used to jump to a label inside the same TRY...CATCH block or to leave the TRY...CATCH block. The TRY...CATCH construct should not be used in a user-defined function.

15.5 Information

It is a good practice to display error information along with the error, so that it can help to solve the error quickly and efficiently.

To achieve this, system functions need to be used in the CATCH block to find information about the error that initiated the CATCH block to execute.

System functions are as follows:

- ERROR_NUMBER(): returns the number of errors.
- ERROR_SEVERITY(): returns the severity.
- ERROR_STATE(): returns state number of the error.
- ERROR_PROCEDURE(): returns the name of the trigger or stored procedure where the error occurred.
- ERROR_LINE(): returns the line number that caused the error.
- ERROR_MESSAGE(): returns the complete text of the error. The text contains the value supplied for the parameters such as object names, length, or times.

The functions return NULL when they are called outside the scope of the CATCH block.

➤ Using TRY...CATCH with error information

Code Snippet 2 demonstrates a simple example displaying error information.

Code Snippet 2:

```
USE AdventureWorks2019;
GO
BEGIN TRY
SELECT 217/0;
END TRY
BEGIN CATCH
SELECT
ERROR_NUMBER() AS ErrorNumber, ERROR_SEVERITY() AS ErrorSeverity, ERROR_LINE()
AS ErrorLine, ERROR_MESSAGE() AS ErrorMessage;
END CATCH;
GO
```

In this code, the `SELECT` statement will cause a divide-by-zero error that is handled using the `TRY...CATCH` statement. The error causes execution to jump to the associated `CATCH` block within which the error information will be displayed.

Figure 15.6 displays the result of the error information. The first resultset is blank because the statement fails.

(No column name)				
	ErrorNumber	ErrorSeverity	ErrorLine	ErrorMessage
1	8134	16	2	Divide by zero error encountered.

Figure 15.6: Error Information

Code Snippet 3 demonstrates a stored procedure that contains error-handling functions.

Code Snippet 3:

```
USE AdventureWorks2019;
GO
IF OBJECT_ID ( 'sp_ErrorInfo', 'P' ) IS NOT NULL
DROP PROCEDURE sp_ErrorInfo;
GO
CREATE PROCEDURE sp_ErrorInfo
AS
SELECT
ERROR_NUMBER() AS ErrorNumber,
ERROR_SEVERITY() AS ErrorSeverity,
ERROR_STATE() AS ErrorState,
ERROR_PROCEDURE() AS ErrorProcedure,
ERROR_LINE() AS ErrorLine,
ERROR_MESSAGE() AS ErrorMessage;
GO
BEGIN TRY SELECT 217/0;
END TRY
BEGIN CATCH
EXECUTE sp_ErrorInfo;
```

```
END CATCH;
```

In this code, when an error occurs, the `CATCH` block of the `TRY...CATCH` construct is called and the error information is returned.

➤ Using `TRY...CATCH` with Transaction

Code Snippet 4 demonstrates a `TRY...CATCH` block that works inside a transaction.

Code Snippet 4:

```
USE AdventureWorks2019;
GO
BEGIN TRANSACTION;
BEGIN TRY
DELETE FROM Production.Product WHERE ProductID = 980;
END TRY
BEGIN CATCH
SELECT
ERROR_SEVERITY() AS ErrorSeverity
,ERROR_NUMBER() AS ErrorNumber
,ERROR_PROCEDURE() AS ErrorProcedure
,ERROR_STATE() AS ErrorState
,ERROR_MESSAGE() AS ErrorMessage
,ERROR_LINE() AS ErrorLine; IF @@TRANCOUNT > 0
ROLLBACK TRANSACTION;
END CATCH;

IF @@TRANCOUNT > 0 COMMIT TRANSACTION;
GO
```

In this code, the `TRY...CATCH` block works within the transaction.

The statement inside the `TRY` block generates a constraint violation error as shown in figure 15.7 which is as follows:

The `DELETE` statement is conflicted with the `REFERENCE` constraint "`FK_BillOfMaterials_Product_ProductAssemblyID`". The conflict occurred in database "`AdventureWorks2019`", table "`Production.BillOfMaterials`", column '`ProductAssemblyID`'.

The screenshot shows a SQL Query window titled "SQLQuery1.sql - DE...D6\Lenovo pc (51)*". The query is as follows:

```

BEGIN TRANSACTION;
BEGIN TRY
    DELETE FROM Production.Product WHERE ProductID = 980;
END TRY
BEGIN CATCH
    SELECT
        ERROR_SEVERITY() AS ErrorSeverity
        ,ERROR_NUMBER() AS ErrorNumber
        ,ERROR_PROCEDURE() AS ErrorProcedure
        ,ERROR_STATE() AS ErrorState
        ,ERROR_MESSAGE() AS ErrorMessage
        ,ERROR_LINE() AS ErrorLine; IF @@TRANCOUNT > 0
    ROLLBACK TRANSACTION;
END CATCH;

IF @@TRANCOUNT > 0 COMMIT TRANSACTION;
GO

```

The results grid shows one row of data:

	ErrorSeverity	ErrorNumber	ErrorProcedure	ErrorState	ErrorMessage	ErrorLine
1	16	547	NULL	0	The DELETE statement conflicted with the REFERENC...	3

Figure 15.7: Constraint Violation Error

➤ Uncommittable Transactions

If an error is generated in a TRY block, it causes the state of the current transaction to be invalid and the transaction is considered as an uncommitted transaction. An uncommittable transaction performs only ROLLBACK TRANSACTION or read operations. The transaction does not execute any Transact-SQL statement that performs a COMMIT TRANSACTION or a write operation.

The XACT_STATE function returns -1 if the transaction has been classified as an uncommittable transaction. When a batch is completed, the Database Engine rolls back any uncommittable transactions. If no error messages are sent when the transaction enters the uncommittable state on completing the batch, then the error messages are sent to the client. This specifies that an uncommittable transaction is detected and rolled back.

15.6 @@ERROR

The @@ERROR function returns the error number for the last Transact-SQL statement executed.

Following is the syntax for the @@ERROR function:

Syntax:

@@ERROR

The @@ERROR system function returns a value of the integer type. This function returns 0, if the previous Transact-SQL statement encountered no errors. It also returns an error number only if the previous statements encounter an error. If the error was one of the errors in the sys.messages catalog view, then @@ERROR contains the value from the sys.messages.message_id column for that error. Users

can view the text associated with an `@@ERROR` error number in the `sys.messagescatalog` view.

Code Snippet 5 demonstrates how to use `@@ERROR` to check for a constraint violation.

Code Snippet 5:

```
USE AdventureWorks2019;
GO
BEGIN TRY
UPDATE HumanResources.EmployeePayHistory SET PayFrequency = 4
WHERE BusinessEntityID = 1;
END TRY
BEGIN CATCH
IF @@ERROR = 547
PRINT N'Check constraint violation has occurred.';
END CATCH
```

In this code, `@@ERROR` is used to check for a check constraint violation (which has error number 547) in an `UPDATE` statement.

It displays the following error message:

```
(0 rows affected)
Check constraint violation has occurred.
```

15.7 RAISERROR

The `RAISERROR` statement starts error processing for a session and displays an error message. `RAISERROR` can reference a user-defined message stored in the `sys.messages` catalog view or build dynamic error messages at run-time. The message is returned as a server error message to the calling application or to the associated `CATCH` block of a `TRY...CATCH` construct.

Following is the syntax for `RAISERROR` statement.

Syntax:

```
RAISERROR ( { msg_id | msg_str | @local_variable }
{ ,severity,state }
[ ,argument [ ,...n ] ] )
[ WITH option [ ,...n ] ]
```

where,

`msg_id`: specifies the user-defined error message number that is stored in the `sys.messages` catalog view using the `sp_addmessage`.

`msg_str`: specifies the user-defined messages with formatting. `msg_str` is a string of characters with optional embedded conversion specifications. A conversion specification has the following format:

```
% [[flag] [width] [. precision] [{h | l}]] type
```

Parameters that can be used in `msg_str` are as follows:

{`h` | `l`} `type`: Specifies the use of character types `d`, `i`, `o`, `s`, `x`, `X`, or `u`, and creates `shortint(h)` or `longint(l)` values.

Following are some of the type specifications:

`d` or `i`: Specifies the signed integer

`o`: Specifies the unsigned octal

`x` or `X`: Specifies the unsigned hexadecimal

`flag`: Specifies the code that determines the spacing and justification of the substituted value. This can include symbols such as - (minus) and + (plus) to specify left-justification or to indicate the value is a signed type respectively.

`precision`: Specifies the maximum number of characters taken from the argument value for string values. For example, if a string has five characters and the precision is 2, only the first two characters of the string value are used.

`width`: Specifies an integer that defines the minimum width for the field in which the argument value is placed.

`@local_variable`: Specifies a variable of any valid character data type that contains string formatted in the same way as `msg_str`.

`severity`: Severity levels from 0 through 18 are specified by any user. Severity levels from 19 through 25 are specified by members of the `sysadmin` fixed server role or users with ALTER TRACE permissions. Severity levels from 19 through 25 uses the `WITH LOG` option is required.

`option`: Specifies the custom option for the error.

Table 15.1 lists the values for the custom options.

Value	Description
<code>LOG</code>	Records the error in the error log and the application log for the instance of the Microsoft SQL Server Database Engine.
<code>NOWAIT</code>	Sends message directly to the client.
<code>SETERROR</code>	Sets the <code>ERROR_NUMBER</code> and <code>@@ERROR</code> values to <code>msg_id</code> or 5000 irrespective of the severity level.

Table 15.1: Type Specification Values

When `RAISERROR` executes with a severity of 11 or higher in a `TRY` block, it will transfer the control to the associated `CATCH` block.

Following errors are returned back to the caller if `RAISERROR` executes:

- Out of scope of any `TRY` block
- Having severity of 10 or lower in `TRY` block
- Having severity of 20 or higher that terminates the database connection

A `CATCH` block can use the `RAISERROR` statement to rethrow the error that has invoked the `CATCH` block. For this, it will require to know the original error information which can be obtained through the

`ERROR_NUMBER` and the `ERROR_MESSAGE` system functions.

By default, the `@@ERROR` is set to 0 for messages that have a severity from 1 through 10.

Code Snippet 6 demonstrates how to build a `RAISERROR` statement to display a customized error statement.

Code Snippet 6:

```
RAISERROR (N'This is an error message %s %d.', 10, 1, N'serial number', 23);
GO
```

In this code, the `RAISERROR` statements takes the first argument of N'serial number' changes the first conversion specification of `%s`, and the second argument of 23 changes the second conversion of `%d`. The code snippet displays the 'This is error message serial number 23'. Code Snippet 7 demonstrates how to use `RAISERROR` statement to return the same string.

Code Snippet 7:

```
RAISERROR (N'%.*s', 10, 1, 7, 3, N'Helloworld');
GO

RAISERROR (N'%7.3s', 10, 1, N'Helloworld');
GO
```

In this code, the `RAISERROR` statements return the same string, `Hel`. The first statement specifies the width and the precision values and the second statement specifies the conversion specification.

Users can also return error information from a `CATCH` block.

Code Snippet 8 demonstrates how to use `RAISERROR` statement inside the `TRY` block.

Code Snippet 8:

```
BEGIN TRY
RAISERROR ('Raises Error in the TRYblock.', 16, 1);
END TRY
BEGIN CATCH
DECLARE @ErrorMessage NVARCHAR(4000); DECLARE @ErrorSeverity INT;
DECLARE @ErrorState INT; SELECT
@ErrorMessage = ERROR_MESSAGE(), @ErrorSeverity=ERROR_SEVERITY(),
@ErrorState=ERROR_STATE();
RAISERROR (@ErrorMessage, @ErrorSeverity, @ErrorState);
END CATCH;
```

In this code, the `RAISERROR` statement used inside the `TRY` block has severity 16, which causes the execution to jump to the associated `CATCH` block.

`RAISERROR` is then used inside the `CATCH` block to return the error information about the original error.

15.8 ERROR_STATE

The `ERROR_STATE` system function returns the state number of the error that causes the `CATCH` block of a `TRY...CATCH` construct to execute. Following is the syntax for `ERROR_STATE` system function:

Syntax:

```
ERROR_STATE ( )
```

When called in a `CATCH` block, it returns the state number of the error message that caused the `CATCH` block to be run. This returns a `NULL` when it is called outside the scope of a `CATCH` block.

There are specific error messages that are raised at various points in the code for the SQL Server Database Engine. For example, an error 1105 is raised for several different conditions. Each specific condition that raises error assigns the unique state code.

`ERROR_STATE` is called from anywhere within the scope of a `CATCH` block. `ERROR_STATE` returns the error state regardless of how many times it is executed or whether it is executed within the scope of the `CATCH` block. This is in comparison with the functions such as `@@ERROR` that only returns the error number in the statement directly after the one that caused error or in the first statement of a `CATCH` block.

Users can use the `ERROR_STATE` in a `CATCH` block. Code Snippet 9 demonstrates how to use `ERROR_STATE` statement inside the `TRY` block.

Code Snippet 9:

```
BEGIN TRY  
SELECT 217/0;  
END TRY  
BEGIN CATCH  
SELECT ERROR_STATE() AS ErrorState;  
END CATCH;  
GO
```

In this code, the `SELECT` statement generates a divide-by-zero error. The `CATCH` statement will then return the state of the error. The `ERROR_STATE` is displayed as 1.

15.9 ERROR_SEVERITY

The `ERROR_SEVERITY` function returns the severity of the error that causes the `CATCH` block of a `TRY...CATCH` construct to be executed. Following is the syntax for `ERROR_SEVERITY`:

Syntax:

```
ERROR_SEVERITY ( )
```

It returns a `NULL` value if called outside the scope of the `CATCH` block. `ERROR_SEVERITY` can be called anywhere within the scope of a `CATCH` block. In nested `CATCH` blocks, `ERROR_SEVERITY` will return the error severity that is specific to the scope of the `CATCH` block where it is referenced. Users can use the `ERROR_SEVERITY` function in a `CATCH` block.

Code Snippet 10 shows how to display the severity of the error.

Code Snippet 10:

```
BEGIN TRY
SELECT 217/0;
END TRY
BEGIN CATCH
SELECT ERROR_SEVERITY() AS ErrorSeverity;
END CATCH;
GO
```

In this code, an attempt to divide by zero generates the error and causes the `CATCH` block to display the severity error as 16.

15.10 ERROR PROCEDURE

The `ERROR_PROCEDURE` function returns the trigger or a stored procedure name where the error has occurred that has caused the `CATCH` block of a `TRY...CATCH` construct to be executed.

Following is the syntax of the `ERROR_PROCEDURE`:

Syntax:

```
ERROR_PROCEDURE( )
```

It returns the `nvarchar` data type. When the function is called in a `CATCH` block, it will return the name of the stored procedure where the error occurred. The function returns a `NULL` value if the error has not occurred within a trigger or a stored procedure. `ERROR_PROCEDURE` can be called from anywhere in the scope of a `CATCH` block. The function also returns `NULL` if this function is called outside the scope of a `CATCH` block.

In nested `CATCH` blocks, the `ERROR_PROCEDURE` returns the trigger or stored procedure name specific to the scope of the `CATCH` block where it is referenced.

Code Snippet 11 shows the use of the `ERROR_PROCEDURE` function.

Code Snippet 11:

```
USE AdventureWorks2019;
GO
IF OBJECT_ID ( 'usp_Example', 'P' ) IS NOT NULL
DROP PROCEDURE usp_Example;
GO
CREATE PROCEDURE usp_Example AS
SELECT 217/0;
GO
BEGIN TRY
EXECUTE usp_Example;
END TRY
BEGIN CATCH
SELECT ERROR_PROCEDURE() AS ErrorProcedure;
END CATCH;
GO
```

In this code, the stored procedure `usp_Example` generates a divide-by-zero error. The `ERROR_PROCEDURE`

function accordingly returns the name of this stored procedure where the error has occurred.

Code Snippet 12 demonstrates the use of `ERROR_PROCEDURE` function along with other functions.

Code Snippet 12:

```
USE AdventureWorks2019;
GO
IF OBJECT_ID ('usp_Example', 'P') IS NOT NULL
DROP PROCEDURE usp_Example;
GO
CREATE PROCEDURE usp_Example AS
SELECT 217/0;
GO
BEGIN TRY
EXECUTE usp_Example;
END TRY
BEGIN CATCH SELECT
ERROR_NUMBER() AS ErrorNumber, ERROR_SEVERITY() AS ErrorSeverity, ERROR_STATE()
AS ErrorState, ERROR_PROCEDURE() AS ErrorProcedure, ERROR_MESSAGE() AS
ErrorMessage, ERROR_LINE() AS ErrorLine;
END CATCH;
GO
```

This code makes use of several error handling system functions that can help to detect and rectify an error easily.

15.11 `ERROR_NUMBER`

The `ERROR_NUMBER` system function when called in a `CATCH` block returns the error number of the error that causes the `CATCH` block of a `TRY...CATCH` construct to be executed. Following is the syntax of `ERROR_NUMBER`:

Syntax:

```
ERROR_NUMBER()
```

The function can be called from anywhere inside the scope of a `CATCH` block. The function will return `NULL` when it is called out of the scope of a `CATCH` block.

`ERROR_NUMBER` returns the error number irrespective of how many times it executes or whether it executes within the scope of a `CATCH` block. This is different than the `@@ERROR` which only returns the error number in the statement immediately after the one that causes error, or the first statement of the `CATCH` block.

Code Snippet 13 demonstrates the use of `ERROR_NUMBER` in a `CATCH` block.

Code Snippet 13:

```
BEGIN TRY
SELECT 217/0;
END TRY
BEGIN CATCH
SELECT ERROR_NUMBER() AS ErrorNumber;
END CATCH;
GO
```

As a result of this code, the error number is displayed when the attempted division by zero occurs.

15.12 ERROR_MESSAGE

The `ERROR_MESSAGE` function returns the text message of the error that causes the `CATCH` block of a `TRY...CATCH` construct to execute.

Following is the syntax of `ERROR_MESSAGE`:

Syntax:

```
ERROR_MESSAGE( )
```

When the `ERROR_MESSAGE` function is called in the `CATCH` block, it returns the full text of the error message that causes the `CATCH` block to execute. The text includes the values that are supplied for any parameter that can be substituted such as object names, times, or lengths. It also returns `NULL` if it is called outside the scope of a `CATCH` block.

Code Snippet 14 demonstrates the use of `ERROR_MESSAGE` in a `CATCH` block.

Code Snippet 14:

```
BEGIN TRY
SELECT 217/0;
END TRY
BEGIN CATCH
SELECT ERROR_MESSAGE() AS ErrorMessage;
END CATCH;
GO
```

In this code, similar to other examples, the `SELECT` statement generates a divide-by-zero error. The `CATCH` block displays the error message.

15.13 ERROR_LINE

The `ERROR_LINE` function returns the line number at which the error occurred in the `TRY...CATCH` block.

Following is the syntax of `ERROR_LINE`:

Syntax:

```
ERROR_LINE( )
```

When this function is called in the `CATCH` block, it returns the line number where the error has occurred. If the error has occurred within a trigger or a stored procedure, it returns the line number in that trigger or stored procedure. Similar to other functions, this function returns a `NULL` if it is called outside the scope of a `CATCH` block.

Code Snippet 15 demonstrates the use of `ERROR_LINE` in a `CATCH` block.

Code Snippet 15:

```
BEGIN TRY
SELECT 217/0;
END TRY
BEGIN CATCH
SELECT ERROR_LINE() AS ErrorLine;
END CATCH;
GO
```

As a result of this code, the line number at which the error has occurred will be displayed.

15.14 Errors Unaffected by the TRY...CATCH Construct

The TRY...CATCH construct does not trap the following conditions:

- Informational messages or warnings having a severity of 10 or lower
- An error that has a severity of 20 or higher that stops the SQL Server Database Engine task processing for the session. If errors occur that have severity of 20 or higher and the database connection is not interrupted, the TRY...CATCH will handle the error
- Attentions such as broken client connection or client-interrupted requests
- When the session ends because of the KILL statements used by the system administrator

Following types of errors are not handled by a CATCH block that occur at the same execution level as that of the TRY...CATCHconstruct:

- Compile errors such as syntax errors that restrict a batch from running
- Errors that arise in the statement-level recompilation such as object name resolution errors occurring after compiling due to deferred name resolution.

Code Snippet 16 demonstrates how an object name resolution error is generated by the SELECT statement.

Code Snippet 16:

```
USE AdventureWorks2019;
GO
BEGIN TRY
SELECT * FROM Nonexistent;
END TRY
BEGIN CATCH
SELECT
ERROR_NUMBER() AS ErrorNumber,
ERROR_MESSAGE() AS ErrorMessage;
END CATCH
```

This code will cause the object name resolution error in the SELECT statement. It will not be caught by the TRY...CATCHconstruct.

Running a similar SELECTstatement inside a stored procedure causes the error to occur at a level lower than the TRY block. The error is handled by the TRY...CATCH construct. Code Snippet 17 demonstrates how the error message is displayed in such a case.

Code Snippet 17:

```
IF OBJECT_ID (N'sp_Example', N'P') IS NOT NULL
DROP PROCEDURE sp_Example;
GO
CREATE PROCEDURE sp_Example AS
SELECT * FROM Nonexistent;
GO
BEGIN TRY
EXECUTE sp_Example;
END TRY
BEGIN CATCH SELECT
ERROR_NUMBER() AS ErrorNumber,
ERROR_MESSAGE() AS ErrorMessage;
END CATCH;
```

15.15 THROW

The `THROW` statement raises an exception and transfers control of the execution to a `CATCH` block of a `TRY...CATCH` construct.

Following is the syntax of the `THROW` statement:

Syntax:

```
THROW [ { error_number | @local_variable },
{ message | @local_variable },
{ state | @local_variable }
] [ ; ]
```

where,

`error_number`: specifies a constant or variable that represents the `error_number` as `int`.

`message`: specifies a variable or string that defines the exception message as `nvarchar(2048)`.

`state`: specifies a variable or a constant between 0 and 255 that specifies the state to associate with `message` as `tinyint`.

Code Snippet 18 demonstrates the use of `THROW` statement to raise an exception again.

Code Snippet 18:

```
USE tempdb;
GO
CREATE TABLE dbo.TestRethrow
(ID INT PRIMARY KEY
);
BEGIN TRY
INSERT dbo.TestRethrow(ID) VALUES(1);
INSERT dbo.TestRethrow(ID) VALUES(1);
END TRY
BEGIN CATCH
PRINT 'In catch block.';
THROW;
END CATCH;
```

In this code, the `THROW` statement is used to raise once again the exception that had last occurred.

The outcome of the code will be as follows:

```
(1 row(s) affected)
```

```
(0 row(s) affected)
```

In catch block.

```
Msg 2627, Level 14, State 1, Line 6
```

```
Violation of PRIMARY KEY constraint 'PK_TestReth_3214EC27AAB15FEE'. Cannot insert
duplicate key in object 'dbo.TestRethrow'. The duplicate key value is (1).
```

15.16 Check Your Progress

1. _____ occur only if the user writes code that cannot be parsed by SQL Server, such as a wrong keyword or an incomplete statement.

(A)	Syntax Errors	(C)	Logical Errors
(B)	Run-time Errors	(D)	Error Log

2. Which of the following constructs can catch unhandled errors from triggers or stored procedures?

(A)	IF-ELSE	(C)	RAISERROR
(B)	TRY...CATCH	(D)	@@ERROR

3. Which of the following functions returns the error number for the last Transact-SQL statement executed?

(A)	ERROR_LINE	(C)	@@ERROR
(B)	RAISERROR	(D)	@@ERROR_NUMBER

4. Which of these functions returns the severity of the error that causes the CATCH block of a TRY...CATCH construct to be executed?

(A)	ERROR_LINE	(C)	ERROR_PROCEDURE
(B)	ERROR_NUMBER	(D)	ERROR_SEVERITY

5. The _____ statement raises an exception and transmits the execution to a CATCH block of a TRY...CATCH construct in SQL Server 2019.

(A)	BEGIN	(C)	THROW
(B)	END	(D)	ERROR

15.16.1 Answers

1.	A
2.	B
3.	C
4.	D
5.	C



Summary

- Syntax errors are the errors that occur when code cannot be parsed by SQL Server.
- Run-time errors occur when the application tries to perform an action that is supported neither by Microsoft SQL Server nor by the operating system.
- TRY...CATCH statements are used to handle exceptions in Transact-SQL.
- TRY...CATCH constructs can also catch unhandled errors from triggers or stored procedures that execute through the code in a TRY block.
- GOTO statements can be used to jump to a label inside the same TRY...CATCH block or to leave a TRY...CATCH block.
- Various system functions are available in Transact-SQL to print error information about the error that occurred.
- The RAISERROR statement is used to start the error processing for a session and displays an error message.



Try It Yourself

1. Acme Technologies Private Limited is a leading software company located at New York. The company has achieved many awards as the best dealer in the development of software technologies. The company has received many new projects on mobile and Web development. At present, they are working on a database project for Payroll Management System in SQL Server 2019. They have created the database on Payroll management system for the employees of Client Company. At the time of table creation and data insertion, they receive different types of errors. Consider that you are the database administrator for Acme Technologies and your technical head has assigned you the task of error rectification.

To achieve the solution for task assigned to you, you need to perform the following steps:

- a. Write error-handling statements using the TRY...CATCH construct for both normal statements as well as stored procedures.
- b. Display the error information using the following:
 - ERROR_NUMBER
 - ERROR_MESSAGE
 - ERROR_LINE

Session - 16

Enhancements in SQL Server 2019

Welcome to the Session, **Enhancements in SQL Server 2019**.

This session covers some of the enhancements made in SQL Server 2019. These include Verbose Truncation Warnings, Vulnerability Assessment, Big Data Clusters, and how to use JSON data effectively with SQL Server 2019.

In this session, you will learn to:

- Describe the importance of new error messages
- Describe how to perform vulnerability assessment
- Explain Big Data clusters
- Explain how to use JSON data with SQL Server 2019

16.1 Verbose Truncation Warnings

'Verbose Truncation Warnings' is one of the greatest features launched in SQL Server 2019. People from development industry are been waiting for this feature from quite a long time. It saves a lot of time while importing, inserting, and updating huge amount of data.

Sometimes we receive the bad data in terms of character limits more than the defined limit in a column. For example, if we want to insert bulk data into a database table using an insert statement, we get a bad data on character length 5 into Color Name column while our existing column (Color_Name) allows only three characters, so how SQL Server will behave? It will raise an SQL truncate error. Insert statement will fail in this case. We normally call it as silent truncation and occur when we try to insert string data (varchar, nvarchar, char, and nchar) into more than the size of the column.

Let us first create a sample database and a table to demonstrate the issue in earlier version of SQL Server. SQL Server 2017 has been used for this example.

Step 1: Create a **Sample DB 2017** database and set the compatibility to 140 as shown in Code Snippet 1.
Code Snippet 1:

```
USE [master]
GO
--Create Sample Database
CREATE DATABASE [Sample DB 2017]
GO
ALTER DATABASE [Sample DB 2017]
SET COMPATIBILITY_LEVEL = 140 -- SQL Server 2017
GO
```

Step 2: Create a sample table and insert few records as shown in Code Snippet 2.

Code Snippet 2:

```
USE [Sample DB 2017]
GO
CREATE TABLE [dbo].[tbl_Color](
    [Color ID] [int] IDENTITY(1,1) NOT NULL,
    [Color Name] [varchar](3) NULL
) ON [PRIMARY]
GO

INSERT INTO [dbo].[tbl_Color]
    ([Color Name])
VALUES
    ('Red'),
    ('Blue'),
    ('Green')
GO
```

Output of Code Snippet 2 is displayed in figure 16.1.

```
SQLQuery2.sql - DE...D6\Lenovo pc (53)* ↗ ×
...
USE [Sample DB 2017]
GO
CREATE TABLE [dbo].[tbl_Color](
    [Color ID] [int] IDENTITY(1,1) NOT NULL,
    [Color Name] [varchar](3) NULL
) ON [PRIMARY]
GO

INSERT INTO [dbo].[tbl_Color]
    ([Color Name])
VALUES
    ('Red'),
    ('Blue'),
    ('Green')
GO

100 % ▶ Messages
Msg 8152, Level 16, State 30, Line 24
String or binary data would be truncated.
The statement has been terminated.

Completion time: 2020-11-17T15:18:34.8164099+05:30
```

Figure 16.1: General Error Message

In figure 16.1, as we can see, we get the SQL truncate error message ‘String or binary data would be truncated.’ In other words, data size is exceeding the table’s column size. However, it does not specify

which column and which data is exceeding the limit in the table.

Consider if insertion or updation of more than one million rows in a table which has more than 50 columns. It is complicated to perform and understand right? Now, you don't need to worry because this issue has been fixed in SQL Server 2019.

You can also set the database compatibility by selecting the **Database Properties** and then, select the **Options** and set the **Compatibility level**.

Now, let us look at the SQL Server 2019 behavior of this SQL truncate issue.

Step 1: Create a database in SQL Server 2019 by default the compatibility is 150 as shown in Code Snippet 3.

Code Snippet 3:

```
USE [master]
GO
--Create Sample Database
CREATE DATABASE [Sample DB 2019]
GO
```

Step 2: Create a sample table and insert few records as shown in Code Snippet 4.

Code Snippet 4:

```
USE [Sample DB 2019]
GO
CREATE TABLE [dbo].[tbl_Color] (
    [Color ID] [int] IDENTITY(1,1) NOT NULL,
    [Color Name] [varchar](3) NULL
) ON [PRIMARY]
GO

INSERT INTO [dbo].[tbl_Color]
    ([Color Name])
VALUES
    ('Red'),
    ('Blue'),
    ('Green')
GO
```

The output of Code Snippet 4 is displayed in Figure 16.2.

The screenshot shows a SQL Server Management Studio (SSMS) window titled "SQLQuery2.sql - DE...D6\Lenovo pc (53)*". The query window contains the following T-SQL code:

```
USE [Sample DB 2019]
GO
CREATE TABLE [dbo].[tbl_Color](
    [Color ID] [int] IDENTITY(1,1) NOT NULL,
    [Color Name] [varchar](3) NULL
) ON [PRIMARY]
GO

INSERT INTO [dbo].[tbl_Color]
([Color Name])
VALUES
    ('Red'),
    ('Blue'),
    ('Green')
GO
```

The results pane shows a single error message:

Msg 2628, Level 16, State 1, Line 46
String or binary data would be truncated in table 'Sample DB 2019.dbo.tbl_Color', column 'Color Name'. Truncated value: 'Blu'.
The statement has been terminated.

Completion time: 2020-11-17T15:36:04.0473260+05:30

Figure 16.2: Detailed Error Message

Output displayed in figure 16.2 shows error message which is very precise. It shows the database name, table name, column name, and specify data exceeding the limit. In this case 'Blue' in Column 'Color Name' is exceeding the column size which is varchar (3) .

16.2 Vulnerability Assessment

SQL Vulnerability Assessment is an easy-to-configure service that can discover, track, and help you reverse or reduce potential database vulnerabilities. You can use it to proactively improve your database security.

Vulnerability Assessment is part of the Azure Defender for SQL offering, which is a unified package for advanced SQL security capabilities. Vulnerability Assessment can be accessed and managed via the central Azure Defender for SQL portal.

Note: Vulnerability Assessment is supported for Azure SQL Database, Azure SQL Managed Instance, and Azure Synapse Analytics (formerly SQL Data Warehouse).

SQL Vulnerability Assessment is a service that provides visibility into your security state. Vulnerability Assessment includes actionable steps to resolve security issues and enhance your database security. It can help you to:

- Meet compliance requirements that require database scan reports
- Meet data privacy standards
- Monitor a dynamic database environment where changes are difficult to track

Vulnerability Assessment is a scanning service built into Azure SQL Database. The service employs a

knowledge base of rules that flag security vulnerabilities. It highlights deviations from best practices, such as misconfigurations, excessive permissions, and unprotected sensitive data.

The rules are based on Microsoft's best practices and focus on the security issues that present the biggest risks to your database and its valuable data. They cover database-level issues and server-level security issues, such as server firewall settings and server-level permissions. These rules also represent many of the requirements from various regulatory bodies to meet their compliance standards.

Results of the scan include actionable steps to resolve each issue and provide customized remediation scripts where applicable. You can customize an assessment report for your environment by setting an acceptable baseline for:

- Permission configurations
- Feature configurations
- Database settings

Following steps are used to implement vulnerability assessment:

Step 1: Run a scan.

- Go to **Azure SQL Database**, **SQL Managed Instance Database**, or **Azure Synapse resource** in the **Azure portal**.
- Under the **Security heading**, select **Security center**.
- Then, click **Select Storage** on the **Vulnerability Assessment** pane to open the **Vulnerability Assessment settings** pane for either the entire server or managed instance.
- Configure a storage account where your scan results for all databases on the server or managed instance will be stored. After storage is configured, select **Scan** to scan your database for vulnerabilities as shown in figure 16.3.

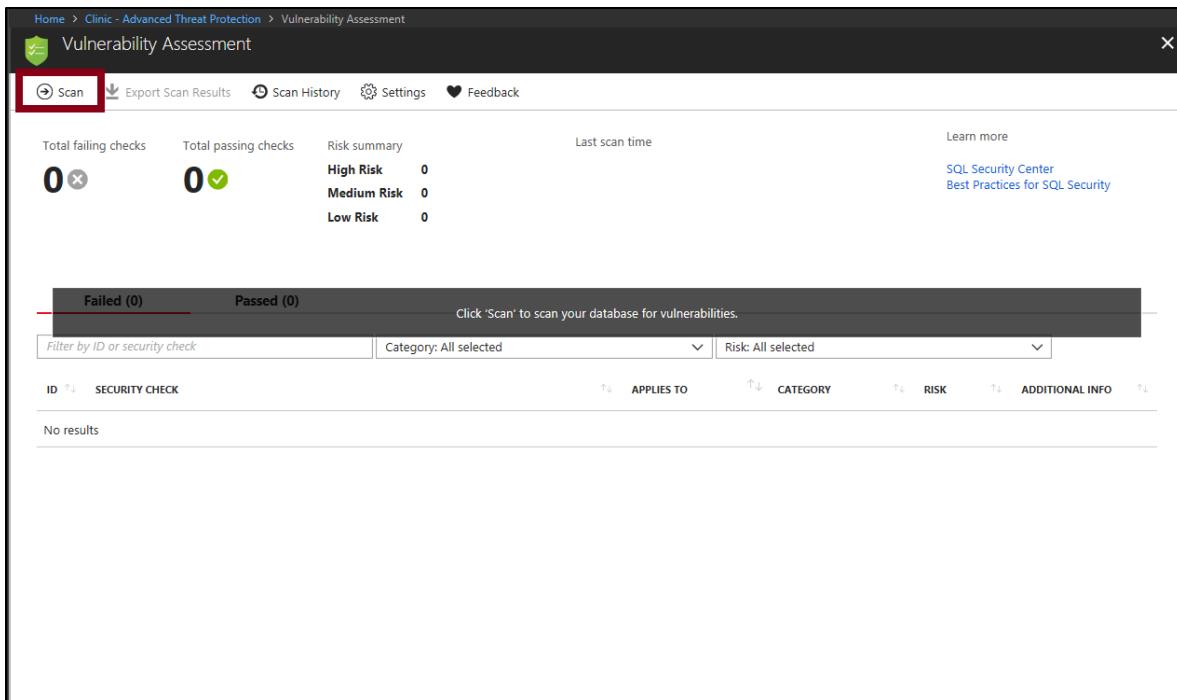


Figure 16.3: Run a Scan

Note: The scan is lightweight and safe. It takes a few seconds to run and is entirely read-only. It does not make any changes to your database.

Step 2: View the report

When vulnerability scan is finished, the scan report is automatically displayed in the Azure portal as shown in figure 16.4. The report presents an overview of your security state. It lists how many issues were found and their respective severities. Results include warnings on deviations from best practices and a snapshot of your security-related settings, such as database principals and roles and their associated permissions. Scan report also provides a map of sensitive data discovered in your database.

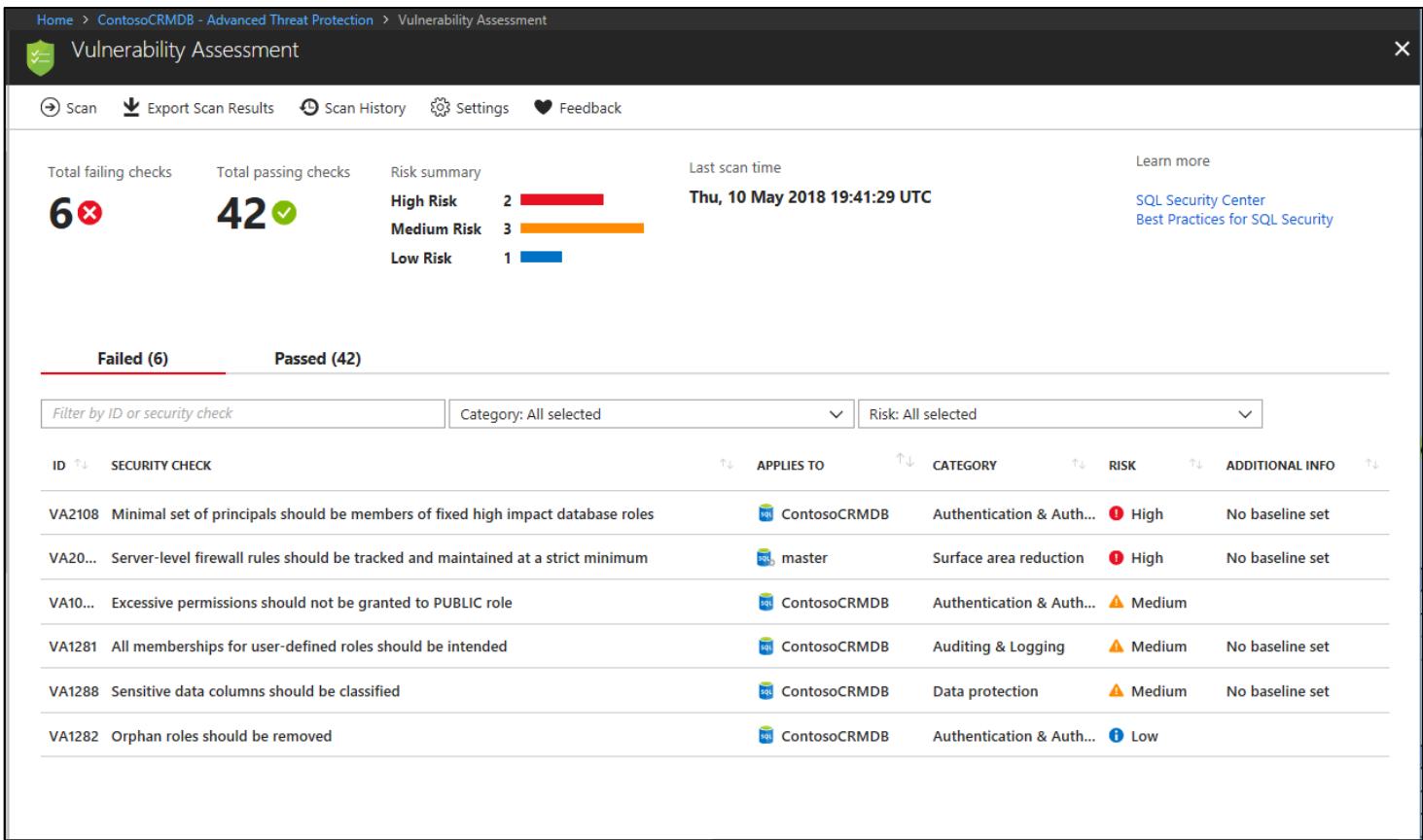


Figure 16.4: Vulnerability Report

Step 3: Analyze the results and resolve issues

Review your results and determine the findings in the report that are true security issues in your environment as shown in figure 16.5. Drill down to each failed result to understand the impact of the finding and why each security check failed. Use the actionable remediation information provided by the report to resolve the issue.

VA2108 - Minimal set of principals should be members of fixed high impact database roles

Approve As Baseline Clear Baseline

NAME	VA2108 - Minimal set of principals should be members of fixed high impact database roles			
RISK	High			
STATUS	✖ FAIL			
DESCRIPTION	<p>SQL Server provides roles to help manage the permissions. Roles are security principals that group other principals. Database-level roles are database-wide in their permission scope. This rule checks that a minimal set of principals are members of the fixed database roles.</p> <p>Fixed database roles may have administrative permissions on the system. Following the principle of least privilege, it is important to minimize membership in fixed database roles and keep a baseline of these memberships. See https://docs.microsoft.com/en-us/sql/relational-databases/security/authentication-access/database-level-roles for additional information on database roles.</p>			
RULE QUERY	<pre>SELECT user_name(sr.member_principal_id) as [Principal], user_name(sr.role_principal_id) as [Role], type_desc as [Principal Type]</pre> Run in Query Editor			
MICROSOFT RECOMMENDATION	Empty Set			
RESULTS	IN BASELINE	PRINCIPAL	ROLE	PRINCIPAL TYPE
	✖	michael8	db_ddladmin	SQL_USER
	✖	test1	db_ddladmin	DATABASE_ROLE

REMEDIATION Remove members who should not have access to the database role

REMEDIATION SCRIPT

```
ALTER ROLE [db_ddladmin] DROP MEMBER [michael8]
ALTER ROLE [db_ddladmin] DROP MEMBER [test1]
```

[Run in Query Editor](#)

Figure 16.5: Security Issues

Step 4: Set your baseline

As you review your assessment results, you can mark specific results as being an acceptable baseline in your environment as shown in figure 16.6. The baseline is essentially a customization of how the results are reported. Results that match the baseline are considered as passing in subsequent scans. After you have established your baseline security state, Vulnerability Assessment only reports on deviations from the baseline. In this way, you can focus your attention on the relevant issues.

VA2108 - Minimal set of principals should be members of fixed high impact database roles

Approve As Baseline Clear Baseline

NAME	VA2108 - Minimal set of principals should be members of fixed high impact database roles			
RISK	High			
STATUS	✖ FAIL			
DESCRIPTION	SQL Server provides roles to help manage the permissions. Roles are security principals that group other principals. Database-level roles are database-wide in their permission scope. This rule checks that a minimal set of principals are members of the fixed database roles.			
IMPACT	Fixed database roles may have administrative permissions on the system. Following the principle of least privilege, it is important to minimize membership in fixed database roles and keep a baseline of these memberships. See https://docs.microsoft.com/en-us/sql/relational-databases/security/authentication-access/database-level-roles for additional information on database roles.			
RULE QUERY	<pre>SELECT user_name(sr.member_principal_id) as [Principal], user_name(sr.role_principal_id) as [Role], type_desc as [Principal Type]</pre> Run in Query Editor			
MICROSOFT RECOMMENDATION	Empty Set			
RESULTS	IN BASELINE	PRINCIPAL	ROLE	PRINCIPAL TYPE
	✖	michael8	db_ddladmin	SQL_USER
	✖	test1	db_ddladmin	DATABASE_ROLE

Figure 16.6: Approve as Baseline

Step 5: Run a new scan to see your customized tracking report

After you finish setting up your **Rule Baselines**, run a new scan to view the customized report as shown in figure 16.7. Vulnerability Assessment now reports only the security issues that deviate from your approved baseline state.

Home > ContosoCRMDB - Advanced Threat Protection > Vulnerability Assessment

Vulnerability Assessment

Scan Export Scan Results Scan History Settings Feedback

Total failing checks	Total passing checks	Risk summary	Last scan time	Learn more
3 ✖	45 ✓	High Risk: 0 Medium Risk: 2 Low Risk: 1 	Thu, 17 May 2018 09:50:22 UTC	SQL Security Center Best Practices for SQL Security

Failed (3) Passed (45)

Filter by ID or security check: Category: All selected Status: All selected

ID	SECURITY CHECK	APPLIES TO	CATEGORY	STATUS
VA1281	All memberships for user-defined roles should be intended	ContosoCRMDB	Auditing & Logging	✓ PASS (per custom baseline)
VA2108	Minimal set of principals should be members of fixed high impact database roles	ContosoCRMDB	Authentication & Auth...	✓ PASS (per custom baseline)
VA20...	Server-level firewall rules should be tracked and maintained at a strict minimum	master	Surface area reduction	✓ PASS (per custom baseline)
VA1020	Server principal GUEST should not be a member of any role	ContosoCRMDB	Authentication & Auth...	✓ PASS
VA20...	Database-level firewall rules should not grant excessive access	ContosoCRMDB	Surface area reduction	✓ PASS
VA20...	Database-level firewall rules should be tracked and maintained at a strict minimum	ContosoCRMDB	Surface area reduction	✓ PASS
VA10...	Excessive permissions should not be granted to PUBLIC role on objects or columns	ContosoCRMDB	Authentication & Auth...	✓ PASS
VA10...	Principal GUEST should not be granted permissions in the database	ContosoCRMDB	Authentication & Auth...	✓ PASS

Figure 16.7: Customize Scan Report

Vulnerability Assessment can now be used to monitor that your database maintains a high level of security at all times and that your organizational policies are met. If compliance reports are required, Vulnerability Assessment reports can be helpful to facilitate the compliance process.

16.3 Big Data Clusters

In SQL Server 2019 Big Data Clusters allow to deploy scalable clusters of SQL Server, Spark, and Hadoop Distributed File System (HDFS) containers running on Kubernetes. These components are running side by side and enable read, write, and process big data from Transact-SQL or Spark, allowing you to easily combine and analyze your high-value relational data with high-volume big data.

Note: Kubernetes (also known as k8s or "kube") is an open source container orchestration platform that automates many of the manual processes involved in deploying, managing, and scaling containerized applications.

Following are some popular uses of Big Data Clusters:

- Deploy scalable clusters of SQL Server, Spark, and HDFS containers running on Kubernetes
- Read, write, and process big data from Transact-SQL or Spark
- Easily combine and analyze high-value relational data with high-volume big data
- Query external data sources
- Store big data in HDFS managed by SQL Server
- Query data from multiple external data sources through the cluster
- Use the data for AI, machine learning, and other analysis tasks
- Deploy and run applications in Big Data Clusters
- Virtualize data with PolyBase
- Query data from external SQL Server, Oracle, Teradata, MongoDB, and ODBC data sources with external tables
- Provide high availability for the SQL Server master instance and all databases by using Always On availability group technology

SQL Server Big Data Clusters provide flexibility in interacting with big data. Query external data sources and store big data in HDFS managed by SQL Server. Then this data can be used for AI, machine learning, and other analysis tasks.

Data Virtualization

SQL Server Big Data Clusters can query external data sources without moving or copying the data as shown in figure 16.8.

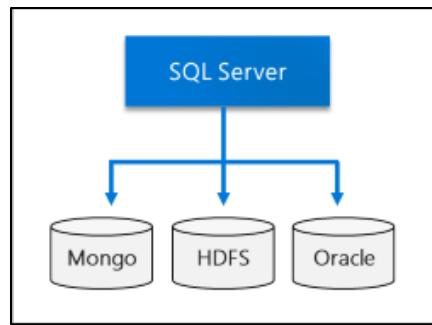


Figure 16.8: Data Virtualization

Data Lake

Data Lake is a storage repository that holds a huge amount of raw data in its native format. It is a scalable HDFS storage pool. This can be used to store big data, potentially ingested from multiple external data sources. Once the big data is stored in HDFS big data cluster, and then it can be analyzed. One can query the data and combine it with relational data available in SQL Server as shown in figure 16.9.

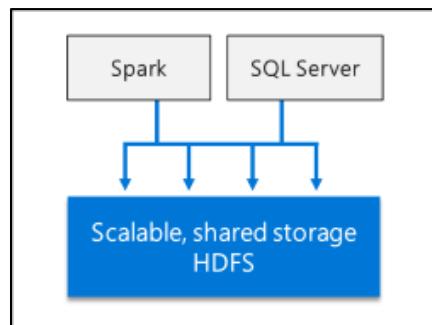


Figure 16.9: Data Lake

Scale-out data mart

SQL Server Big Data Clusters provide scale-out compute and storage to improve the performance of analyzing any data. Data from a variety of sources can be ingested and distributed across data pool nodes as a cache for further analysis as shown in figure 16.10.

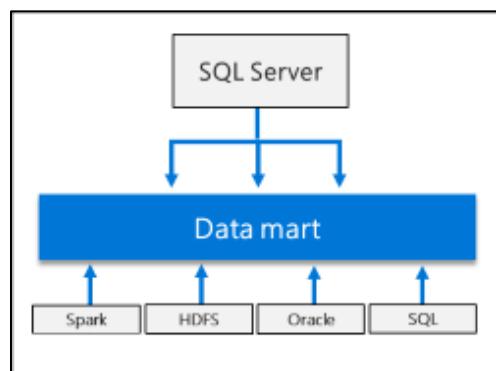


Figure 16.10: Data Mart

Integrated AI and Machine Learning

SQL Server Big Data Clusters enable AI and machine learning tasks on the data stored in HDFS storage pools and the data pools as shown in figure 16.11. Spark as well as built-in AI tools in SQL Server, using R, Python, Scala, or Java are used to perform AI and Machine learning related task.

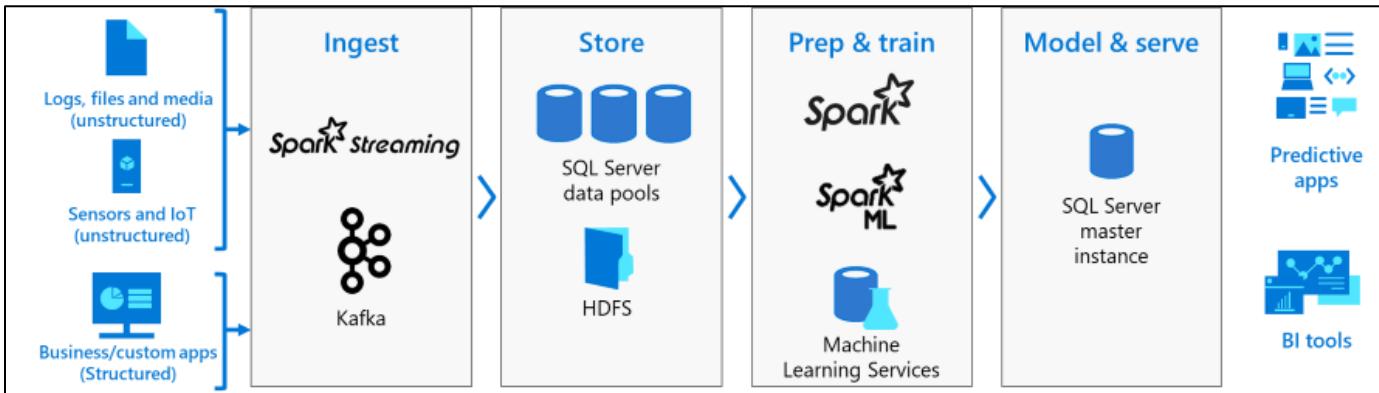


Figure 16.11: Integrated AI and ML

A SQL Server big data cluster is a cluster of Linux containers orchestrated by Kubernetes.

Kubernetes Terms

Kubernetes is an open source container orchestrator, which can scale container deployments according to the need. Table 16.1 defines some important Kubernetes terminology:

Term	Description
Cluster	A Kubernetes cluster is a set of machines, known as nodes. One node controls the cluster and is designated the master node; the remaining nodes are worker nodes. The Kubernetes master is responsible for distributing work between the workers and for monitoring the health of the cluster.
Node	A node runs containerized applications. It can be either a physical machine or a virtual machine. A Kubernetes cluster can contain a mixture of physical machine and virtual machine nodes.
Pod	A pod is the atomic deployment unit of Kubernetes. A pod is a logical group of one or more containers-and associated resources-required to run an application. Each pod runs on a node; a node can run one or more pods. The Kubernetes master automatically assigns pods to nodes in the cluster.

Table 16.1: Kubernetes Terms

In SQL Server Big Data Clusters, Kubernetes is responsible for the state of the SQL Server Big Data Clusters; Kubernetes builds and configures the cluster nodes, assigns pods to nodes, and monitors the health of the cluster.

Figure 16.12 shows the components of a SQL Server big data cluster.

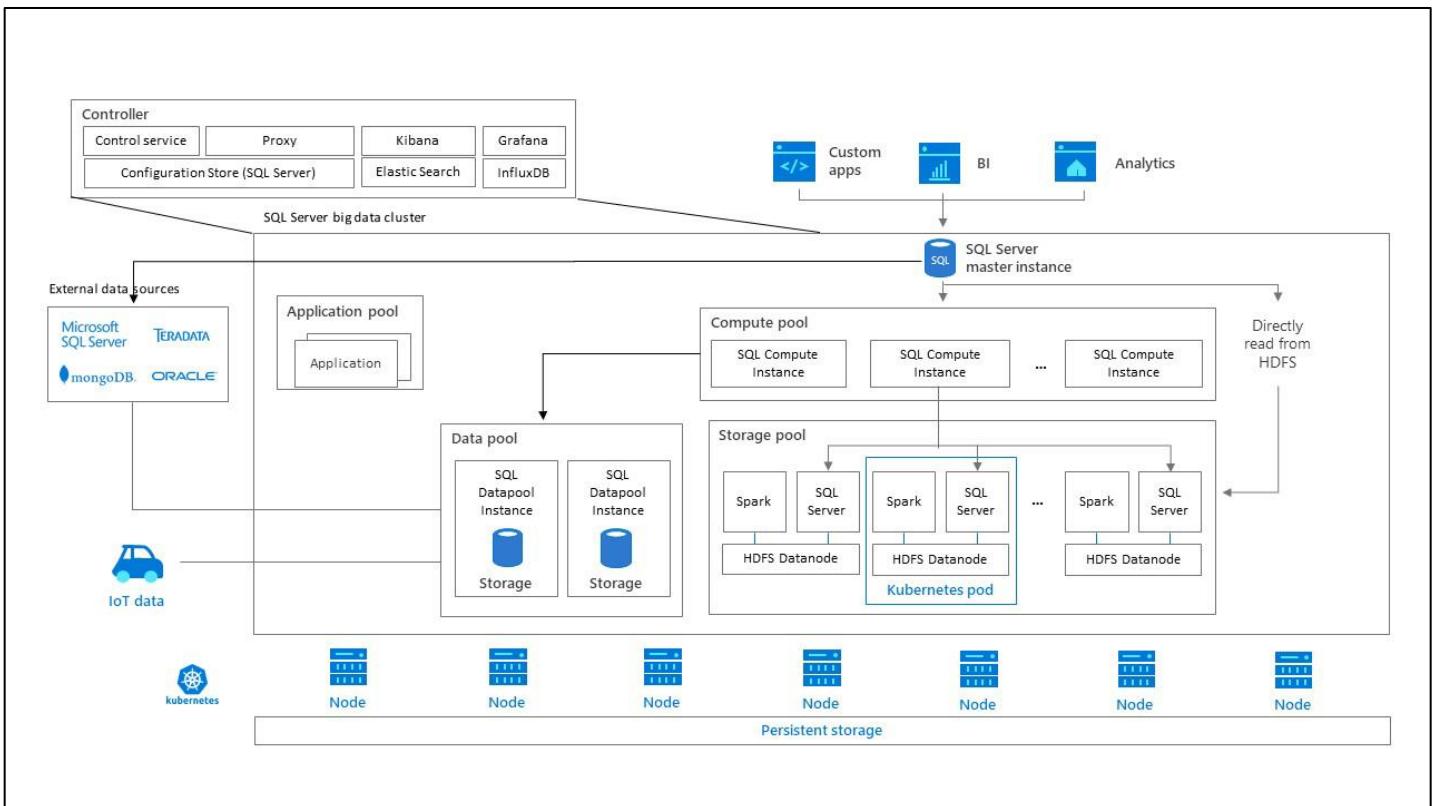


Figure 16.12: SQL Server Big Data Cluster

Controller

Controller provides management and security for the cluster. It contains the control service, the configuration store, and other cluster-level services such as Kibana, Grafana, and Elastic Search.

Compute Pool

Compute pool provides computational resources to the cluster. It contains nodes running SQL Server on Linux pods. The pods in the compute pool are divided into SQL Compute instances for specific processing tasks.

Data Pool

Data pool is used for data persistence and caching. The data pool consists of one or more pods running SQL Server on Linux. It is used to ingest data from SQL queries or Spark jobs. SQL Server big data cluster data marts are persisted in the data pool.

Storage Pool

Storage pool consists of storage pool pods comprised of SQL Server on Linux, Spark, and HDFS. All the storage nodes in a SQL Server big data cluster are members of an HDFS cluster.

16.4 JSON data in SQL Server

JSON is a textual data format that is used for exchanging data in modern Web and Mobile applications. JSON is also used to store unstructured data in log files or NoSQL databases such as Microsoft Azure Cosmos DB. Many REST Web services return results that are formatted as JSON text or accept data that is formatted as JSON.

For example, most Azure services, such as Azure Search, Azure Storage, and Azure Cosmos DB, have REST endpoints that return or consume JSON. JSON is also the main format for exchanging data between

Web pages and Web Servers by using AJAX calls.

JSON functions in SQL Server enable you to combine NoSQL and relational concepts in the same database. Now, it is possible to combine classic relational columns with columns that contain documents formatted as JSON text in the same table, parse and import JSON documents in relational structures, or format relational data to JSON text.

Code Snippet 5 shows sample JSON data.

Code Snippet 5:

```
[  
  {  
    "name": "John",  
    "skills": ["SQL", "C#", "Azure"]  
  },  
  {  
    "name": "Jane",  
    "surname": "Doe"  
  }  
]
```

By using SQL Server built-in functions and operators following things can be performed with JSON text as shown in figure 16.13.

- Parse JSON text and read or modify values
- Transform arrays of JSON objects into table format
- Run any Transact-SQL query on the converted JSON objects
- Format the results of Transact-SQL queries in JSON format

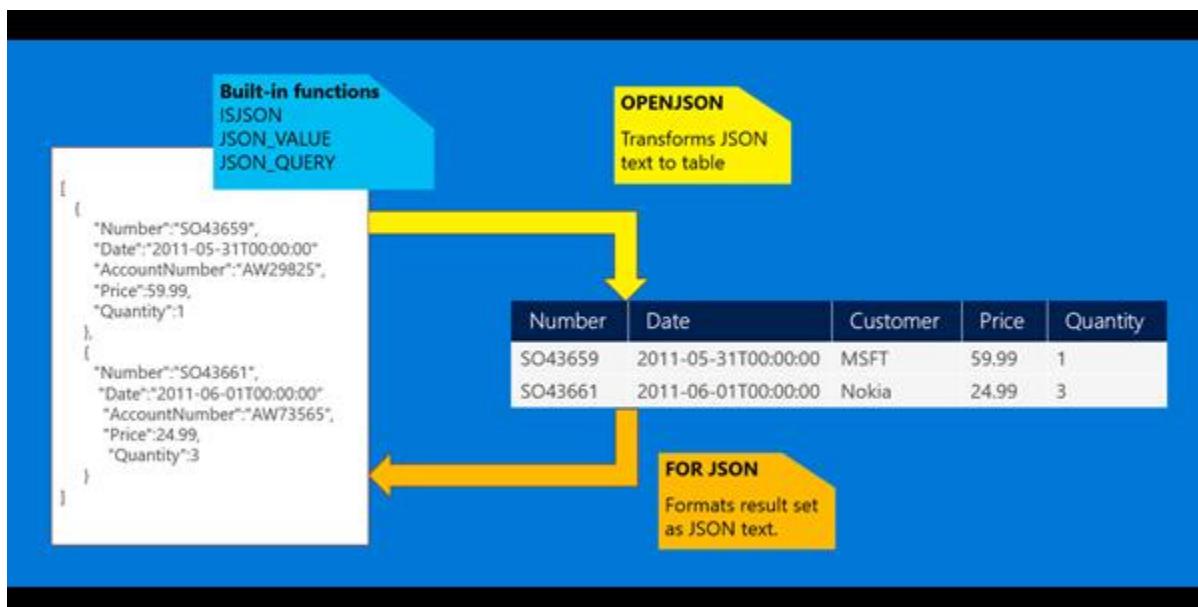


Figure 16.13: Built-in JSON function

If JSON text is stored in database tables, values in JSON text can be read or modified by using following built-in functions as shown in Code Snippet 6:

- ISJSON (Transact-SQL) tests whether a string contains valid JSON
- JSON_VALUE (Transact-SQL) extracts a scalar value from a JSON string
- JSON_QUERY (Transact-SQL) extracts an object or an array from a JSON string
- JSON_MODIFY (Transact-SQL) changes a value in a JSON string

Code Snippet 6:

```
SELECT Name, Surname,
    JSON_VALUE(jsonCol, '$.info.address.PostCode') AS PostCode,
    JSON_VALUE(jsonCol, '$.info.address."Address Line 1"') + ' '
    + JSON_VALUE(jsonCol, '$.info.address."Address Line 2"') AS Address,
    JSON_QUERY(jsonCol, '$.info.skills') AS Skills
FROM People
WHERE ISJSON(jsonCol) > 0
    AND JSON_VALUE(jsonCol, '$.info.address.Town') = 'Belgrade'
    AND Status = 'Active'
ORDER BY JSON_VALUE(jsonCol, '$.info.address.PostCode')
```

In Code Snippet 6 both relational and JSON data is used. Values from JSON text are used in any part of a Transact-SQL query including WHERE, ORDER BY, or GROUP BY clauses, window aggregates, and so on. JSON functions use syntax similar to JavaScript for referencing values inside JSON text.

Modify JSON Values

To modify parts of JSON text, JSON_MODIFY (Transact-SQL) function is used to update the value of a property in a JSON string and return the updated JSON string. Code Snippet 7 updates value of a property in a variable that contains JSON.

Code Snippet 7:

```
DECLARE @json NVARCHAR(MAX);
SET @json = '{"info": {"address": [{"town": "Belgrade"}, {"town": "Paris"}, {"town": "Madrid"}]} }';
SET @json = JSON_MODIFY(@json, '$.info.address[1].town', 'London');
SELECT modifiedJson = @json;
```

Output of Code Snippet 7 is displayed in figure 16.14.

```
{"info":{"address":[{"town":"Belgrade"}, {"town": "London"}, {"town": "Madrid"}]}
```

Figure 16.14: Modified JSON

OPENJSON to Convert JSON to rowset

Custom query language is not required to query JSON in SQL Server. To query JSON data, standard T-SQL can be used. If you wish to retrieve or create a report on JSON data, you can easily able to do that by

calling the `OPENJSON` rowset function, which will convert JSON data to rows and columns.

Code Snippet 8 calls `OPENJSON` and transforms array of objects that is stored in the `@json` variable to a rowset that can be queried with a standard SQL `SELECT` statement:

Code Snippet 8:

```
DECLARE @json NVARCHAR(MAX);
SET @json = N'[
    {"id": 2, "info": {"name": "John", "surname": "Smith"}, "age": 25},
    {"id": 5, "info": {"name": "Jane", "surname": "Smith"}, "dob": "2005-11-04T12:00:00"}
]';
SELECT *
FROM OPENJSON(@json)
    WITH (
        id INT 'strict $.id',
        firstName NVARCHAR(50) '$.info.name',
        lastName NVARCHAR(50) '$.info.surname',
        age INT,
        dateOfBirth DATETIME2 '$.dob'
    );
```

The output of Code Snippet 8 is displayed in figure 16.15.

Results				
	id	firstName	lastName	age
1	2	John	Smith	25
2	5	Jane	Smith	NULL

Figure 16.15: JSON Variable to ROWSET

`OPENJSON` transforms the array of JSON objects into a table in which each object is represented as one row and key/value pairs are returned as cells. The output observes the following rules:

- `OPENJSON` converts JSON values to the types that are specified in the `WITH` clause
- `OPENJSON` can handle both flat key/value pairs and nested, hierarchically organized objects
- It is not necessary to return all the fields that are contained in the JSON text
- If JSON values do not exist, `OPENJSON` returns `NULL` values
- Path can be specified optionally after type specification to reference a nested property or to reference a property by a different name
- Optional strict prefix in the path specifies that values for the specified properties must exist in the JSON text

JSON documents may have sub-elements and hierarchical data that cannot be directly mapped into the standard relational columns. In this case, you can flatten JSON hierarchy by joining parent entity with sub-arrays.

In Code Snippet 9, second object in array has sub-array representing person skills and each sub-object can be parsed using additional `OPENJSON` function call.

Code Snippet 9:

```
DECLARE @json NVARCHAR(MAX);
SET @json = N'[
    {"id": 2, "info": {"name": "John", "surname": "Smith"}, "age": 25},
    {"id": 5, "info": {"name": "Jane", "surname": "Smith", "skills": ["SQL", "C#", "Azure"]}, "dob": "2005-11-04T12:00:00"}
]';
SELECT *
FROM OPENJSON(@json)
WITH (
    id INT 'strict $.id',
    firstName NVARCHAR(50) '$.info.name',
    lastName NVARCHAR(50) '$.info.surname',
    age INT,
    dateOfBirth DATETIME2 '$.dob',
    skills NVARCHAR(MAX) '$.info.skills' AS JSON
)
OUTER APPLY OPENJSON(skills)
WITH (skill NVARCHAR(8) '$');
```

In Code Snippet 9, skills array is returned in first OPENJSON function as original JSON text fragment and passed to another OPENJSON function using APPLY operator. Second OPENJSON function will parse JSON array and return string values as single column rowset that will be joined with the result of first OPENJSON.

Output of Code Snippet 9 is displayed in figure 16.16.

	id	firstName	lastName	age	dateOfBirth	skills	skill
1	2	John	Smith	25	NULL	NULL	NULL
2	5	Jane	Smith	NULL	2005-11-04 12:00:00.0000000	["SQL", "C#", "Azure"]	SQL
3	5	Jane	Smith	NULL	2005-11-04 12:00:00.0000000	["SQL", "C#", "Azure"]	C#
4	5	Jane	Smith	NULL	2005-11-04 12:00:00.0000000	["SQL", "C#", "Azure"]	Azure

Figure 16.16: Result of OPENJSON function calls

Export SQL Server Data to JSON

You can format SQL Server data or results of SQL queries as JSON by adding the FOR JSON clause to a SELECT statement. The FOR JSON delegates the formatting of JSON output from your client applications to SQL Server.

Code Snippet 10 uses PATH mode with the FOR JSON clause.

Code Snippet 10:

```
USE AdventureWorks2019;
SELECT BusinessEntityId, FirstName AS "info.name", LastName AS "info.surname",
ModifiedDate AS dob
FROM Person.Person
FOR JSON PATH;
```

The `FOR JSON` clause formats SQL results as JSON text that can be provided to any app that understands JSON. The `PATH` option uses dot-separated aliases in `SELECT` clause to nest objects in query results.

Output of Code Snippet is displayed in figure 16.17.



The screenshot shows a SQL Server Management Studio window with two tabs: 'JSON_F52E2B61-18...0805F49916B1.xml*' and 'SQLQuery1.sql - DE...D6\Lenovo pc (52)*'. The XML tab displays the following JSON data:

```
[{"BusinessEntityId":1,"info":{"name":"Ken","surname":"Sánchez"},"dob":"2009-01-07T00:00:00"}, {"BusinessEntityId":2,"info":{"name":"Terri","surname":"Duffy"},"dob":"2008-01-24T00:00:00"}, {"BusinessEntityId":3,"info":{"name":"Roberto","surname":"Tamburello"},"dob":"2007-11-04T00:00:00"}, {"BusinessEntityId":4,"info":{"name":"Rob","surname":"Walters"},"dob":"2007-11-28T00:00:00"}, {"BusinessEntityId":5,"info":{"name":"Gail","surname":"Erickson"},"dob":"2007-12-30T00:00:00"}, {"BusinessEntityId":6,"info":{"name":"Jossef","surname":"Goldberg"},"dob":"2013-12-16T00:00:00"}]
```

Figure 16.17: Exported JSON Data

16.5 Check Your Progress

1. Which of the following details are shown by troubleshooting feature Verbose Truncation Warnings introduced in SQL Server 2019?

(A)	Table Name	(C)	Column Name
(B)	Data exceeding limit	(D)	All of these

2. Which of the following parameters are used to customize assessment report?

(A)	Permission configurations	(C)	Feature configurations
(B)	Database settings	(D)	All of these

3. Which of the following is not key term used in Kubernetes?

(A)	Pod	(C)	Cluster
(B)	Node	(D)	Knot

4. Which of the following big data cluster includes a scalable HDFS storage pool?

(A)	Data Lake	(C)	Data Mart
(B)	Data Virtualization	(D)	None of these

5. _____ extracts a scalar value from a JSON string.

(A)	ISJSON	(C)	JSON_VALUE
(B)	JSON_QUERY	(D)	JSON_MODIFY

16.5.1 Answers

1.	D
2.	D
3.	D
4.	A
5.	B



Summary

- Verbose Truncation Warnings makes troubleshooting easy by displaying additional and exact details in error messages.
- SQL Vulnerability Assessment is a service that provides visibility into your security state.
- Vulnerability report lists how many issues were found and their respective severities.
- SQL Server Big Data Clusters provide flexibility in interacting with Big Data.
- Kubernetes is an open source container orchestrator.
- Kubernetes is responsible for the state of the SQL Server Big Data Clusters.
- JSON is used to store unstructured data in log files or NoSQL databases such as Microsoft Azure Cosmos DB.



Try It Yourself

1. In this exercise you have to go through the Workshop - SQL Server Big Data Clusters - Architecture at following link and perform all the steps on your local machine:
<https://github.com/microsoft/sqlworkshops-bdc>
2. Smith Sons and Corp. have following JSON data related to their account, they want you to convert the same in tabular format with OPENJSON and WITH clause.

```
[  
  {  
    "Order": {  
      "Number": "SO43659",  
      "Date": "2011-05-31T00:00:00"  
    },  
    "AccountNumber": "AW29825",  
    "Item": {  
      "Price": 2024.9940,  
      "Quantity": 1  
    }  
  },  
  {  
    "Order": {  
      "Number": "SO43661",  
      "Date": "2011-06-01T00:00:00"  
    },  
    "AccountNumber": "AW73565",  
    "Item": {  
      "Price": 2024.9940,  
      "Quantity": 3  
    }  
  }  
]
```

Session - 17

PolyBase, Query Store, and Stretch Database

Welcome to the Session, **PolyBase, Query Store, and Stretch Database**.

This session introduces SQL Server features such as PolyBase, Query Store, and Stretch Databases. PolyBase is used to process Transact-SQL queries that read data from external data sources. The session also explains Query Store for analyzing query performance using built-in reports. Finally, the session Stretch Database is used to migrate all or part of the associated history table to cost-effective storage in Azure.

In this session, you will learn to:

- Describe PolyBase
- Explain features and advantages of PolyBase
- Define and describe Query Store
- Explain how to dynamically stretch warm and cold transactional data from SQL Server to Azure
- Describe how to tune workload performance with Query Store

17.1 Understanding PolyBase

PolyBase enables your SQL Server instance to process Transact-SQL queries that read data from external data sources. SQL Server 2016 and higher versions can access external data in Hadoop and Azure Blob Storage. Starting in SQL Server 2019, you can now use PolyBase to access external data in SQL Server, Oracle, Teradata, and MongoDB.

Queries that access external data can also use to target relational tables in your SQL Server instance. This allows to combine data from external sources with high-value relational data in your database. In SQL Server, an external table or external data source provides the connection to Hadoop.

PolyBase pushes some computations to the Hadoop node to optimize the overall query. However, PolyBase external access is not limited to Hadoop. Other unstructured non-relational tables are also supported, such as delimited text files.

PolyBase enables the following scenarios in SQL server:

- **Query data stored in Hadoop from SQL Server or Parallel Data Warehouse (PDW):** Users are storing data in cost-effective distributed and scalable systems, such as Hadoop. PolyBase makes it easy to query the data by using T-SQL.

- **Query data stored in Azure Blob Storage:** Azure Blob storage is a convenient place to store data for use by Azure services. PolyBase makes it easy to access the data by using T-SQL.
- **Import data from Hadoop, Azure Blob Storage, or Azure Data Lake Store:** Leverage the speed of Microsoft SQL's columnstore technology and analysis capabilities by importing data from Hadoop, Azure Blob Storage, or Azure Data Lake Store into relational tables. There is no need for a separate Extract, Transform, and Load (ETL) or import tool.
- **Export data to Hadoop, Azure Blob Storage, or Azure Data Lake Store:** Archive data to Hadoop, Azure Blob Storage, or Azure Data Lake Store to achieve cost-effective storage and keep it online for easy access.
- **Integrate with BI tools:** Use PolyBase with Microsoft's business intelligence and analysis stack or use any third party tools that are compatible with SQL Server.

17.1.1 PolyBase Architecture

The architecture of PolyBase has some similarities to the Hadoop architecture. This is because the purpose of PolyBase is interaction with Hadoop data.

Following are the main components of the PolyBase architecture shown in figure 17.1:

Head node	Compute node	Scale-out Reads
<p>It contains SQL Server instance to which PolyBase queries are submitted. Each PolyBase group can have only one head node. A head node is a logical group of SQL Database Engine, PolyBase Engine, and PolyBase Data Movement Service on the SQL Server instance.</p>	<p>It contains SQL Server instance that assists with scale-out query processing on external data. A compute node is a logical group of SQL Server and the PolyBase data movement service on SQL Server instance. A PolyBase group can have multiple compute nodes. Both head node and compute nodes must run on the same version of SQL Server.</p>	<p>When querying external SQL Server, Oracle or Teradata instances, partitioned tables will benefit from scale-out reads. Each node in a PolyBase scale-out group can spin up to eight readers to read external data and each reader is assigned one partition to read in the external table.</p>

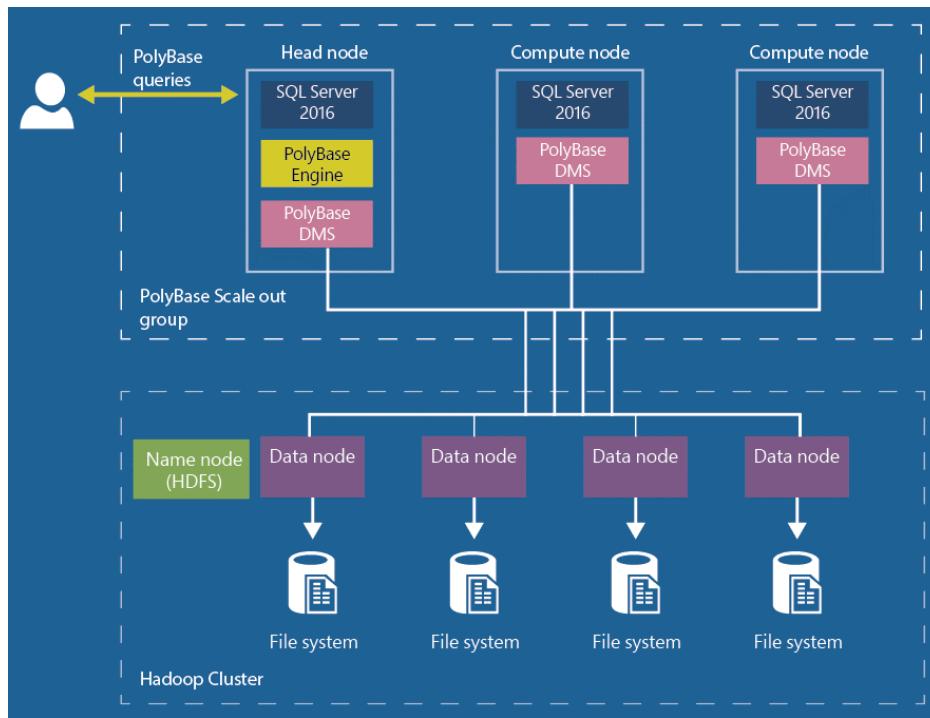


Figure 17.1: PolyBase Architecture

17.1.2 Installing PolyBase in SQL Server 2019

Following are the pre-requisite to install PolyBase in SQL Server 2019:

- 64-bit SQL Server Evaluation edition.
- Microsoft .NET Framework 4.5.
- Minimum memory: 4 GB.
- Minimum hard-disk space: 2 GB.
- Recommended: Minimum of 16 GB RAM.
- TCP/IP must be enabled for PolyBase to function correctly. TCP/IP is enabled by default on all editions of SQL Server except for the Developer and Express SQL Server editions. For PolyBase to function correctly on the Developer and Express editions, you must enable TCP/IP connectivity.

Note: PolyBase can be installed on only one SQL Server instance per machine.

Single node or PolyBase scale-out group:

Before you install PolyBase on your SQL Server instances, decide whether you want a single node installation or a PolyBase scale-out group.

For a PolyBase scale-out group, make sure that:

- All the machines are on the same domain.
- You use the same service account and password during PolyBase installation.
- Your SQL Server instances can communicate with one another over the network.
- The SQL Server instances are all the same version of SQL Server.

After you install PolyBase either standalone or in a scale-out group, you cannot change. To change this setting, you have to uninstall and reinstall the feature.

Following are the steps to install and set up PolyBase:

1. Run the SQL Server setup.exe.
2. Select Installation and then, select New standalone SQL Server installation or add features.
3. On the Feature Selection page as shown figure 17.2, select PolyBase Query Service for External Data.

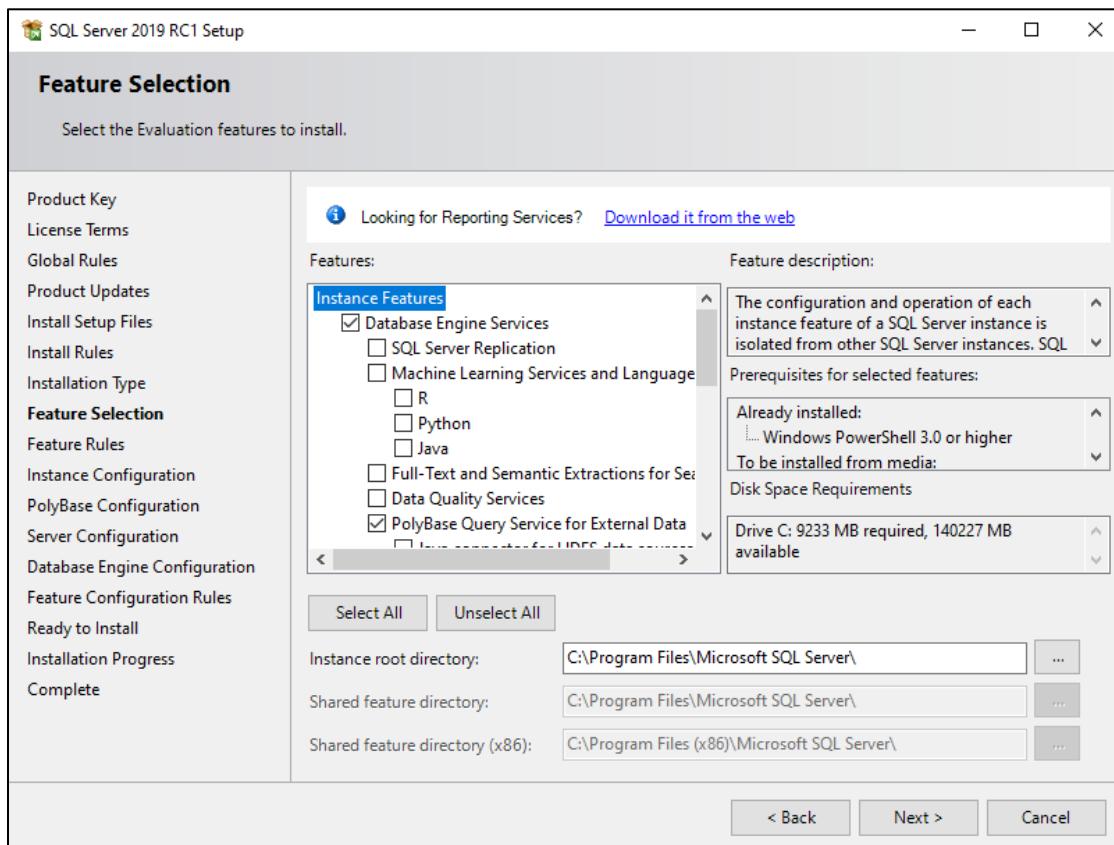


Figure 17.2: Feature Selection

Note: SQL Server 2019 PolyBase now includes an additional option Java connector for HDFS data sources. See SQL Server preview features for more information about this feature.

4. On the Server Configuration page, configure the **SQL Server PolyBase Engine Service** and **SQL Server PolyBase Data Movement Service** to run under the same domain account.
5. On the PolyBase Configuration page, select one of the two options.
 - Use the SQL Server instance as a standalone PolyBase-enabled instance. Choose this option to use the SQL Server instance as a standalone head node.
 - Use the SQL Server instance as part of a PolyBase scale-out group. This option opens the firewall to allow incoming connections. Connections are allowed for the SQL Server Database Engine, SQL

Server PolyBase Engine, SQL Server PolyBase Data Movement service, and the SQL browser. The firewall also allows incoming connections from other nodes in a PolyBase scale-out group.

This option also enables Microsoft Distributed Transaction Coordinator (MSDTC) firewall connections and modifies MSDTC registry settings.

6. On the PolyBase Configuration page, specify a port range with at least six ports. SQL Server setup allocates the first six available ports from the range.

Enable PolyBase

After installation, PolyBase must be enabled to access its features as shown in Code Snippet 1.

Code Snippet 1:

```
exec sp_configure @configname = 'polybase enabled', @configvalue = 1;
RECONFIGURE;
```

Once SQL Server installation is complete with the PolyBase feature, PolyBase must be configured to interact with external data sources.

Note: PolyBase using SQL Server 2019 can interact with Hadoop, Oracle, MongoDB, Teradata, and so on.

Configure Hadoop Connectivity

1. Run `sp_configure` with 'hadoop connectivity' and set an appropriate value for your provider. To find the value for your provider, see official document for PolyBase Connectivity Configuration. By Default, the Hadoop connectivity is set to 7.

Code Snippet 2:

```
sp_configure @configname = 'hadoop connectivity', @configvalue = 7;
GO
RECONFIGURE
GO
```

2. Restart SQL Server using `services.msc`. Restarting SQL Server restarts these services as shown in figure 17.3.
 - SQL Server PolyBase Data Movement Service
 - SQL Server PolyBase Engine

SMS Agent Host	Provides ch...	Running	Automatic (D...	Local Syste...
SNMP Trap	Receives tra...		Manual	Local Service
Software Protection	Enables the ...		Automatic (D...	Network S...
Spot Verifier	Verifies pot...		Manual (Trig...	Local Syste...
SQL Server (MSSQLSERVER)	Provides sto...	Running	Automatic	NT Service
SQL Server Agent (MSSQLSERVER)	Executes jo...		Manual	NT Service
SQL Server Browser	Provides SQ...		Disabled	Local Service
SQL Server PolyBase Data Movement Service. (MSSQLSERVER)	Manages co...	Running	Automatic	Network S...
SQL Server PolyBase Engine (MSSQLSERVER)	Creates, co...	Running	Automatic	Network S...

Figure 17.3: SQL Server Services

Configure an external table

To query the data in your Hadoop data source, you must define an external table to use in Transact-SQL queries. Following steps describe how to configure the external table:

1. Create a master key on the database, if one does not already exist as shown in Code Snippet 3. This is required to encrypt the credential secret.

Code Snippet 3:

```
CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'password';
```

PASSWORD = 'password': It is a password that is used to encrypt the master key in the database. Password must meet the Windows password policy requirements of the computer that is hosting instance of SQL Server.

2. Create a database scoped credential for Kerberos-secured Hadoop clusters as shown in Code Snippet 4.

Code Snippet 4:

```
CREATE DATABASE SCOPED CREDENTIAL HadoopUser1
WITH IDENTITY = '<hadoop_user_name>', Secret = '<hadoop_password>';
```

3. Create an external data source with CREATE EXTERNAL DATA SOURCE as shown in Code Snippet 5.

Code Snippet 5:

```
CREATE EXTERNAL DATA SOURCE MyHadoopCluster WITH (
    TYPE = HADOOP,
    LOCATION ='hdfs://10.xxx.xx.xxxx:xxxx',
    RESOURCE_MANAGER_LOCATION = '10.xxx.xx.xxxx:xxxx',
    CREDENTIAL = HadoopUser1
);
```

4. Create an external file format with CREATE EXTERNAL FILE FORMAT as shown in Code Snippet 6.

Code Snippet 6:

```
CREATE EXTERNAL FILE FORMAT TextFileFormat WITH (
    FORMAT_TYPE = DELIMITEDTEXT,
    FORMAT_OPTIONS (FIELD_TERMINATOR = '|'),
    USE_TYPE_DEFAULT = TRUE)
```

5. Create an external table pointing to data stored in Hadoop with `CREATE EXTERNAL TABLE` as shown in Code Snippet 7. In Code Snippet 7, the external data used is car sensor data.

Code Snippet 7:

```
CREATE EXTERNAL TABLE [dbo].[CarSensor_Data] (
    [SensorKey] int NOT NULL,
    [CustomerKey] int NOT NULL,
    [GeographyKey] int NULL,
    [Speed] float NOT NULL,
    [YearMeasured] int NOT NULL
)
WITH (LOCATION='/Demo/',
      DATA_SOURCE = MyHadoopCluster,
      FILE_FORMAT = TextFileFormat
);
```

6. Create statistics on an external table as shown in Code Snippet 8.

Code Snippet 8:

```
CREATE STATISTICS StatsForSensors on CarSensor_Data(CustomerKey, Speed)
```

17.2 Query Store

SQL Server Query Store feature provides an insight on query plan choice and performance. It simplifies performance troubleshooting by helping quickly to find performance differences caused by query plan changes.

Query Store automatically captures a history of queries, plans, and runtime statistics, and retains these for review. It separates data as per time frames so database usage patterns can be identified and query plan changes happened on the server are noted. Query store can be configured using `ALTER DATABASE SET` option.

17.2.1 Query Store Architecture

Each query compilation or execution by SQL Server sends a message to the Query store. Figure 17.4 illustrates the connection to the Query Store.

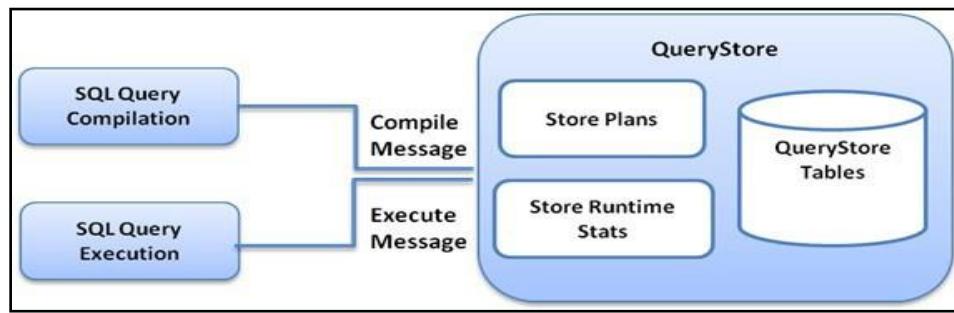


Figure 17.4: SQL Server Messages to Query Store

The information about the compilation and execution is first stored in cache and then, stored to the disk. The duration of keeping the information in the cache and the time when it is stored on the disk is determined by the `INTERVAL_LENGTH_MINUTES` parameter and the `DATA_FLUSH_INTERVAL_SECONDS` parameter, respectively. These parameters should be specified when Query Store is enabled and configured. Sometimes, if there is an overload in the cache, the information is written to the disk and then, the cache is cleared.

Note that information on both the cache and the disk can be accessed by running the `sys.query_store_runtime_stats` catalog.

17.2.2 Enabling Query Store

Query Store is not enabled by default for new SQL Server and Azure Synapse Analytics (SQL DW) databases and is enabled by default for new Azure SQL Database databases.

Using SQL Server Management Studio

1. In Object Explorer, right-click a database and then, click Properties.
2. In the Database Properties dialog box, select the Query Store page.
3. In the Operation Mode (Requested) box, select Read Write.

Note: Requires at least version 16 of Microsoft SQL Server Management Studio.

Using Transact-SQL

Use the `ALTER DATABASE` statement to enable the query store for a given database shown as follows:

```
SET QUERY_STORE = ON (OPERATION_MODE = READ_WRITE);
```

17.2.3 Configuring Query Store

Once the Query Store feature is enabled, other parameters for **Monitoring** and **Query Store Retention** can be configured. Figure 17.5 shows the parameters that can be configured.

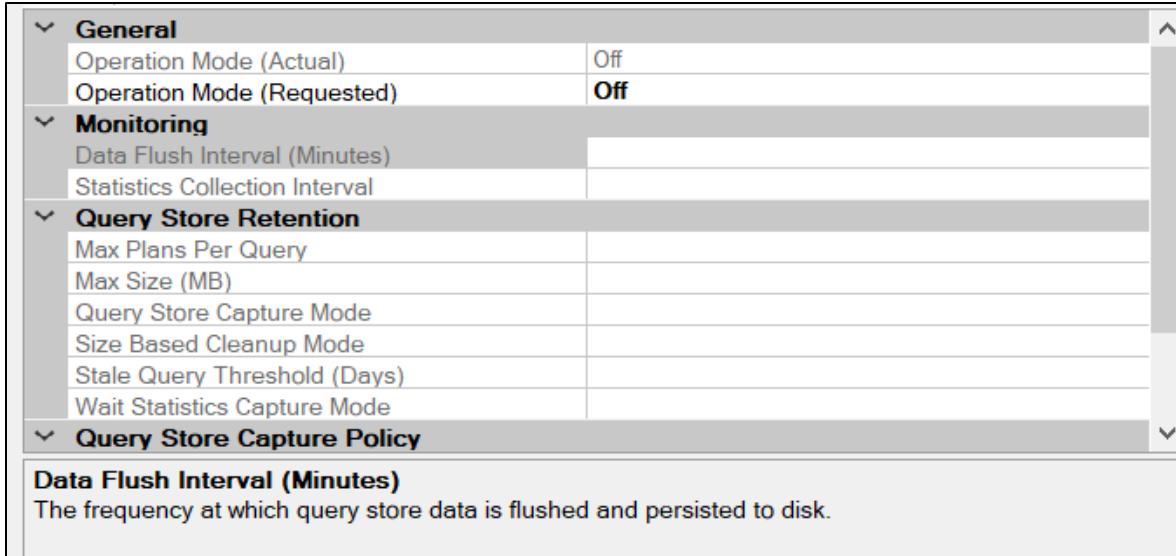


Figure 17.5: Configure Query Store Using SSMS

Alternatively, Transact-SQL statements can be used. Figure 17.6 shows the Transact-SQL statements to configure the Query Store parameters.

```
SQLQuery1.sql - LENOVO-PC\.....(52)*
ALTER DATABASE [DEMO_1]
SET QUERY_STORE (OPERATION_MODE = READ_ONLY,
CLEANUP_POLICY = (STALE_QUERY_THRESHOLD_DAYS = 367),
DATA_FLUSH_INTERVAL_SECONDS = 900,
INTERVAL_LENGTH_MINUTES = 60,
MAX_STORAGE_SIZE_MB = 100,
QUERY_CAPTURE_MODE = AUTO,
SIZE_BASED_CLEANUP_MODE = AUTO)
GO
```

Figure 17.6: Configure Query Store Using Transact-SQL

In the code shown in figure 17.6,

- CLEANUP_POLICY = STALE_QUERY_THRESHOLD_DAYS = 367 specifies that the number of days to retain data in the query store is 367.
- DATA_FLUSH_INTERVAL_SECONDS specifies that data in the query store is flushed to disk at an interval of 900 seconds.
- MAX_STORAGE_SIZE_MB specifies that the maximum size of the query store is 100 MB. Once this size is reached, the query store stops collecting new data and the state changes from read-write to read-only. Starting with SQL Server 2019 (15.x), the default value is 1 GB. Keep track of current space usage and increase the Max Size (MB) value to prevent Query Store from transitioning to read-only mode.
- INTERVAL_LENGTH_MINUTES specifies the time interval at which the execution statistics data is saved to the query store.
- SIZE_BASED_CLEANUP_MODE specifies whether the cleanup process is automatically activated when total amount of data is almost reaching the maximum size.

- `QUERY_CAPTURE_MODE` specifies whether all queries are captured automatically or only some queries are captured based on execution count and resource consumption or if only current queries are captured.
- Additionally, `MAX_PLANS_PER_QUERY` can also be given in a query. It specifies the maximum number of plans maintained for each query.

17.2.4 Query Store System Objects

New system procedures and catalog views related to Query Store can be found in SQL Server. Code Snippet 9 is used to retrieve these procedures and views.

Code Snippet 9:

```
SELECT name, type_desc FROM sys.all_objects
WHERE name LIKE '%query_Store%' or name='query_context_settings'
```

Figure 17.7 displays the list.

	name	type_desc
1	sp_query_store_force_plan	EXTENDED_STORED_PROCEDURE
2	sp_query_store_reset_exec_stats	EXTENDED_STORED_PROCEDURE
3	sp_query_store_remove_plan	EXTENDED_STORED_PROCEDURE
4	sp_query_store_flush_db	EXTENDED_STORED_PROCEDURE
5	database_query_store_options	VIEW
6	sp_query_store_remove_query	EXTENDED_STORED_PROCEDURE
7	sp_query_store_consistency_check	EXTENDED_STORED_PROCEDURE
8	sp_query_store_unforce_plan	EXTENDED_STORED_PROCEDURE
9	query_store_wait_stats	VIEW
10	query_store_runtime_stats_interval	VIEW
11	query_store_runtime_stats	VIEW
12	query_store_query_text	VIEW
13	query_store_query	VIEW
14	query_store_plan	VIEW
15	query_context_settings	VIEW

Query executed successfully.

Figure 17.7: System-Stored Procedures and Catalog Views

17.2.5 Performance Tuning with Query Store

Performance of queries may deteriorate over time as a database grows. To analyze what is causing the performance issue is a challenging and time-consuming task that is made easy with the Query Store feature. The Query Store feature maintains all the information about queries over time and aids in quick resolution of issues. As mentioned before, it also enables to ‘roll-back’ to query plans from history. Following are few examples of application scenarios, where performance-tuning solutions can be provided reactively or proactively by using the Query Store feature:

If any recent plan changes have caused poor performance, Query Store can be used to quickly identify the issue and if required revert to a previous plan.

If any queries are consuming the most of system resources, such as CPU, memory or IO, they can be identified and accordingly changed to ensure optimal usage.

When changes to applications or platforms are planned, Query Store can be used to compare performance before and after change implementation. These changes may include installing new versions of applications, installing new hardware, compatibility level upgrades to the database, and adding or modifying indexes.

When upgrades are planned for Query Optimizer, performance of queries can be recorded before upgradation and fixed during upgradation.

When there is a need to optimize resources, queries with a lower frequency of execution can be identified and allocation of resources to them can be proactively handled.

17.3 Stretch Database

Stretch Database is a new feature built into SQL Server that facilitates storage of a part of a database in the cloud. When there is a need to retain data for a longer time, Stretch Database feature enables secure migration of tables and data to the cloud. This is an economical solution for long-term data storage.

Moreover, applications can still query the data in the same way as before. Enabling Stretch Database feature makes it possible to do the following:

- Move archived data as well as current data to the cloud securely by using the encryption features
- Access and query the data on the cloud at any time, without any changes to existing applications or queries
- Reduce storage requirements of on-premise data by using the vast storage capacity of the cloud
- Reduce processing burden on the on-premise data by running the processes on the cloud in a way that is transparent to the applications

17.3.1 Stretch Database Architecture

A secure linked server definition is created in the on-premise SQL Server when the Stretch Database feature is enabled on a database. This establishes a connection to the remote or cloud database. When Stretch Database is enabled on a table, the resources on cloud are prepared and if migration is enabled, eligible data is migrated to the cloud database. Figure 17.8 illustrates the Stretch Database Architecture.

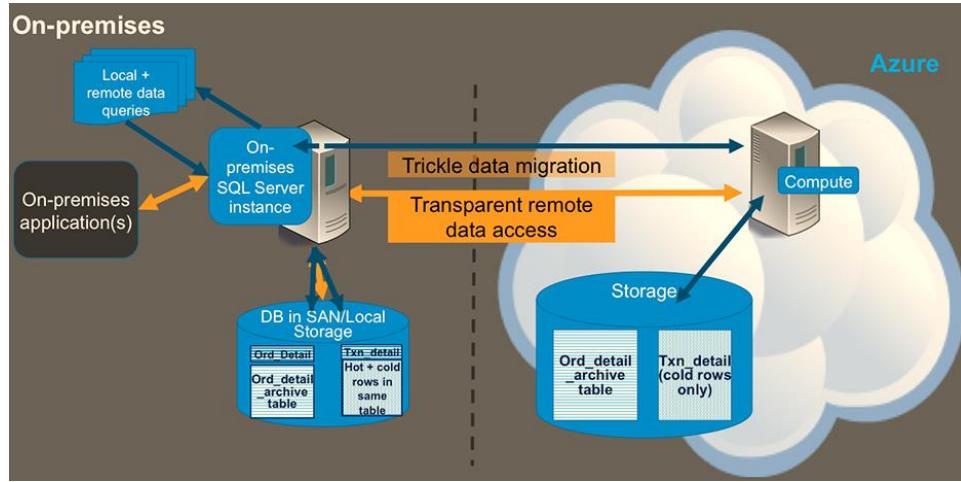


Figure 17.8: Stretch Database Architecture

17.3.2 Setting up Stretch Database

To configure an existing database for Stretch Database, select Tasks | Stretch | Enable for a database in SQL Server Management Studio to open the Enable Database for Stretch wizard.

Transact-SQL can also be used to enable Stretch Database for a database.

If you select Tasks | Stretch | Enable for an individual table and you have not yet enabled the database for Stretch Database, the wizard configures the database for Stretch Database and lets you select tables as part of the process.

Note:

Stretch Database migrates data to Azure. Therefore, you require an Azure account and a subscription for billing.

Enabling Stretch Database on a database or a table requires `db_owner` permissions. Enabling Stretch Database on a database also requires `CONTROL DATABASE` permissions.

Prerequisite to Enable Stretch Database

To enable Stretch Database on a database or a table, first enable it on the local server. This operation requires `sysadmin` or `serveradmin` permissions.

- If you have the required administrative permissions, **Enable Database for Stretch** wizard configures the server for Stretch.
- If you do not have required permissions, an administrator has to enable it manually by running `sp_configure` first then, run the wizard, or an administrator has to run the wizard.

1. Enable Stretch Database on the SQL Server instance as shown in Code Snippet 10.

Code Snippet 10:

```
EXEC sp_configure 'remote data archive', '1';
GO
RECONFIGURE;
GO
```

Code Snippet 10 runs **sp_configure** and enable **remote data archive** option. In this example **remote data archive** option is enabled by setting its value to 1.

2. To configure an existing table for Stretch Database, run the **ALTER TABLE** command. Code Snippet 11 shows syntax that migrates the entire table and begins data migration immediately.

Code Snippet 11:

```
USE <Stretch-enabled database name>;
GO
ALTER TABLE <table name>
    SET (REMOTE_DATA_ARCHIVE = ON (
        FILTER_PREDICATE = dbo.fn_stretchpredicate(),
        MIGRATION_STATE = PAUSED) ) ;
GO
```

3. Select the database and create the table that must be stretched. Code Snippet 12 shows syntax to create a new table with Stretch Database.

Code Snippet 12:

```
USE <Stretch-enabled database name>;
GO
CREATE TABLE <table name>
    ( ... )
    WITH (REMOTE_DATA_ARCHIVE = ON (MIGRATION_STATE = OUTBOUND) ) ;
GO
```

Consider that you have created a sample database and table. This table will be stretched to the Azure cloud. Code Snippet 13 creates a table **StretchSampleTable** on the **StretchDemo** database.

Code Snippet 13:

```
USE [StretchDemo];
GO
CREATE TABLE dbo.StretchSampleTable
(
    FNM VARCHAR(50), LNM VARCHAR(50)
);
GO
```

Insert sample records into the table as shown in Code Snippet 14, where sample data is inserted into the **StretchSampleTable**.

Code Snippet 14:

```
USE [StretchDemo];
GO
INSERT INTO dbo.StretchSampleTable (FNM, LNM)
VALUES('Pamella', 'Whee'), ('Kimon', 'Turing'), ('Joannie', 'Whitt'),
      ('Amelie', 'Janese'), ('Toni', 'Gregory'), ('Tedd', 'Walt');
GO
```

4. On the Object Explorer, right-click the StretchDemo database, select Tasks, then select Stretch and then, select Enable option as shown in figure 17.9.

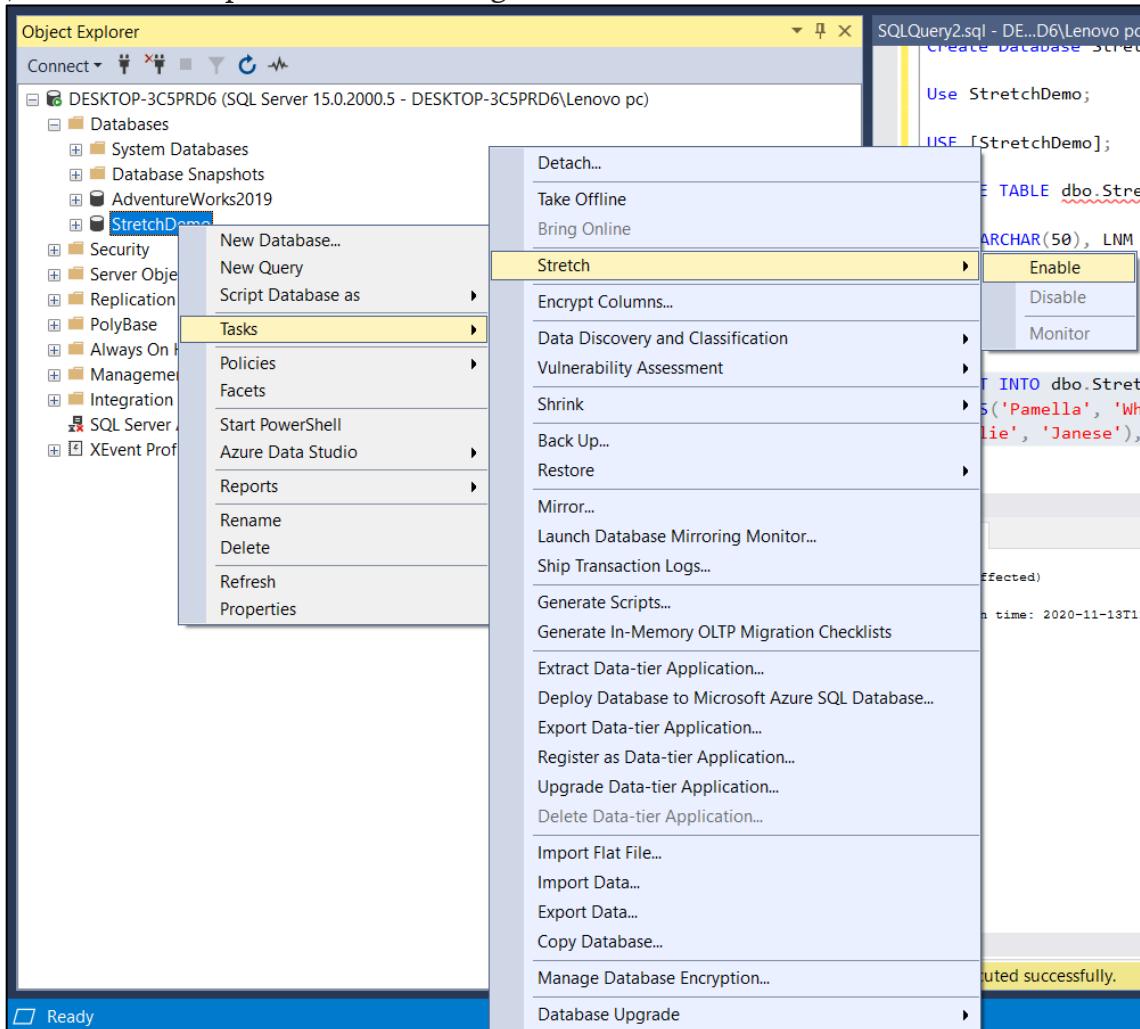


Figure 17.9: Enabling Database for Stretch

The **Enable Database for Stretch** wizard opens as shown in figure 17.10.

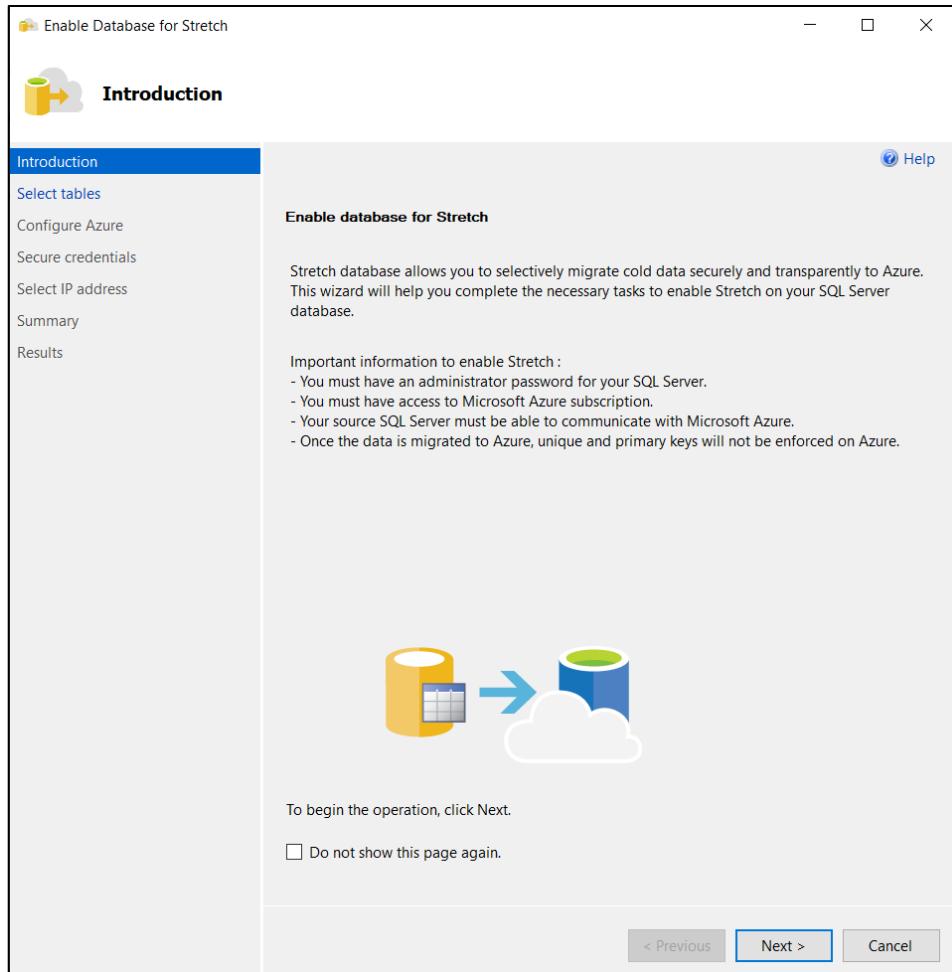


Figure 17.10: Enable Database for Stretch

Select table for stretch as shown in figure 17.11.

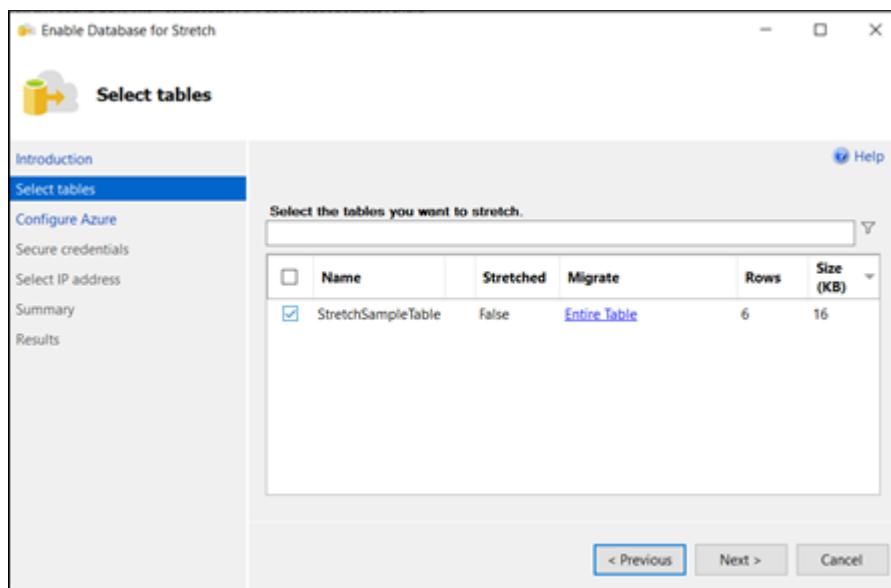


Figure 17.11: Select Tables

5. On the wizard, **Sign In** to the Microsoft Azure account and then, specify the Azure location, as well as the server access credentials. It is recommended to save access credentials.
6. Click **Finish** to verify the choices made and to complete enabling the database for Stretch. A secure linked server definition between the local database and the remote database on Azure is now established.

17.3.3 Stretch Database Limitations

Limitations for Stretch Enable table are:

Constraints

- Uniqueness is not enforced for `UNIQUE` constraints and `PRIMARY KEY` constraints in the Azure table that contains the migrated data.

DML operations

- You are not able to `UPDATE` or `DELETE` rows that have been migrated or rows that are eligible for migration, in a Stretch-enabled table or in a view that includes Stretch-enabled tables.
- You are not able to `INSERT` rows into a Stretch-enabled table on a linked server.

Indexes

- You are not able to create an index for a view that includes Stretch-enabled tables.
- Filters on SQL Server indexes are not propagated to the remote table.

Following items currently prevent you from enabling Stretch for a table:

Table properties

- Tables that have more than 1,023 columns or more than 998 indexes
- FileTables or tables that contain FILESTREAM data
- Tables that are replicated, or that are actively using Change Tracking or Change Data Capture
- Memory-optimized tables

Data types

- `text`, `ntext`, and `image`
- `timestamp`
- `sql_variant`
- `XML`
- CLR data types including `geometry`, `geography`, `hierarchyid`, and CLR user-defined types

Column types

- `COLUMN_SET`
- Computed columns

Constraints

- Default constraints and check constraints

- Foreign key constraints that reference the table. In a parent-child relationship (for example, Order and Order_Detail), you can enable Stretch for the child table (Order_Detail) but not for the parent table (Order).

Indexes

- Full text indexes
- XML indexes
- Spatial indexes
- Indexed views that reference the table

17.4 Check Your Progress

1. Which of the following are added during installation of PolyBase through SQL Server 2019?

(A)	SQL Server PolyBase Engine	(C)	SQL Server Data Movement Service
(B)	DWQueue	(D)	All of these

2. Which of the following actions are necessary for data migration from a database table to an Azure database through SQL Server?

(A)	Stretch Database must be enabled on the Server instance	(C)	Stretch Database must be enabled on the table
(B)	Stretch Database must be enabled on the database	(D)	Stretch Database must be enabled on the Server instance, database, and on the table

3. Which of the following are not supported by Stretch Database?

(A)	Migration of current data to cloud	(C)	Filestream column type
(B)	Querying data on cloud	(D)	Viewing migration status

4. Which of the following is not a function of Query Store?

(A)	Seamless integration with external data store such as Hadoop	(C)	Identify queries with degraded performance
(B)	Store history of query plans in the system	(D)	Force older execution plans when required

5. Which of the following statements are incorrect?

(A)	PolyBase can be installed on only one SQL Server instance per machine	(C)	CREATE EXTERNAL DATA SOURCE statement is necessary to define a connection to Hadoop file from SQL Server
(B)	PolyBase supports Cloudera and Hortonworks flavors of Hadoop	(D)	Delimited Text is not a format of data supported by PolyBase

17.4.1 Answers

1.	D
2.	D
3.	C
4.	A
5.	D



Summary

- The PolyBase feature provides seamless integration with external data sources, such as Hadoop or Azure Blob Storage.
- PolyBase eliminates the need to specialized skills on Hadoop internals by enabling query-runs on external data sources with simple Transact-SQL commands.
- Query Store is a built-in tool to improve performance by maintaining historical information of every query and execution plan.
- Query Store tracks the performance of queries and triggers alerts on poorly performing plans.
- Stretch Database in SQL Server 2019 enables stretching some part of a database to the Azure cloud, thereby, lowering long-term storage costs as well as maintenance efforts.
- Stretch Database is used to migrate archive and data in current transaction to the cloud securely.
- Stretch database is used to access cloud data at any time similar to the local data available on machines.



Try It Yourself

1. Starlight Digital Media is a popular digital media company in UK, they want to migrate all their data stored in relational database to Azure cloud. Assume that they have hired you for their team.

The tasks you are required to perform are:

- i. Create demo database as **StarlightDemo** and insert data in it from **Chinook database** at following link: <https://github.com/cwoodruff/ChinookDatabase>
- ii. Next, create sample Azure Cloud account for Starlight Digital Media.
- iii. Using Stretch database functionality migrate **StarlightDemo** database data to Azure Cloud.