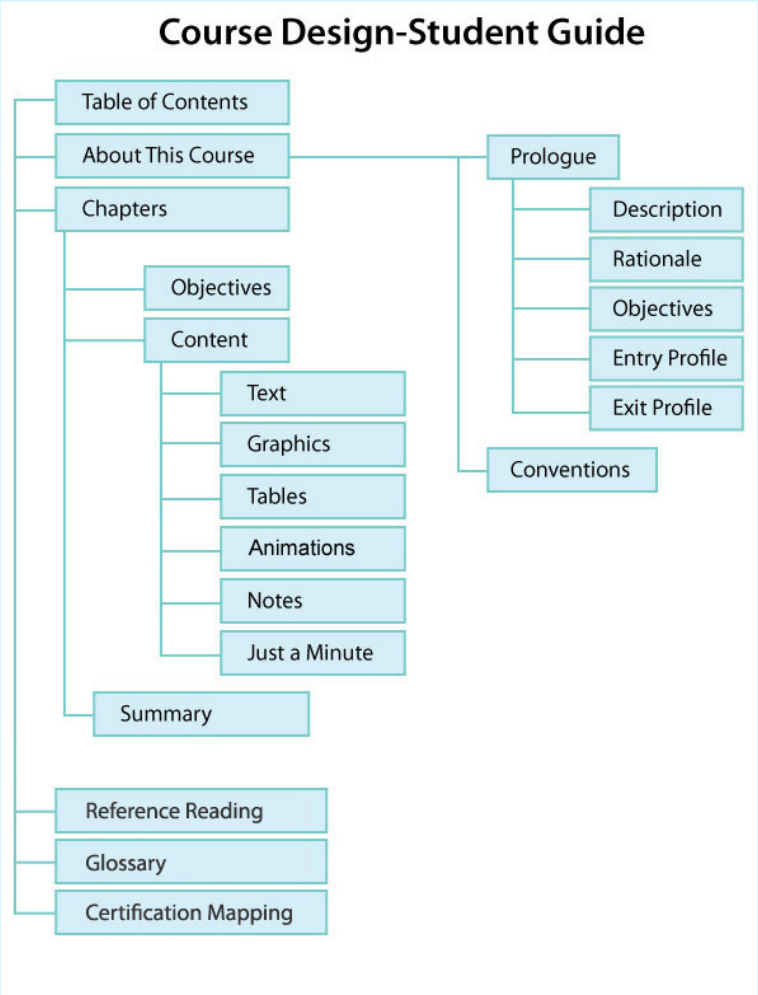
Student Guide

Querying Data Using SQL Server - I

NIIT


Course Design-Student Guide
Table of Contents
About This Course
Chapters
Objectives
Content
Text
Graphics
Tables
Animations
Notes
Just a Minute
Summary
Prologue
Description
Rationale
Objectives
Entry Profile
Exit Profile
Conventions
Reference Reading
Glossary
Certification Mapping

Trademark Acknowledgements
All products are registered trademarks of their respective organizations.
All software is used for educational purposes only.

QDUS - I_SG /12-M11-V02
Copyright ©NIIT. All rights reserved.

No part of this publication may be reproduced, stored in retrieval system or transmitted in any form, or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission of the publisher.

About This Course

Prologue

Description

The Querying Data Using SQL Server - I course is designed to provide students the necessary skills to query and manage databases using SQL Server. The course starts with an introduction to client-server architecture and an overview of SQL Server. Next, it familiarizes the students with Structured Query Language (SQL) and builds on the complexity of query handling in SQL Server with every progressing session. It also introduces the students to some advanced concepts in SQL Server such as implementing managed code and implementing services for message-based communication.

Rationale

Databases are an integral part of an organization. Aspiring database developers should be able to efficiently implement and maintain databases. This knowledge will enable them to build robust database solutions.

Objectives

After completing this course, the student should be able to:

- ❑ Identify SQL Server tools
- ❑ Query data from a single table
- ❑ Query data from multiple tables
- ❑ Manage databases and tables
- ❑ Manipulate data in tables
- ❑ Implement indexes, views, and full-text search
- ❑ Implement stored procedures and functions
- ❑ Implement triggers and transactions
- ❑ Implement managed code
- ❑ Implement services for message-based communication

Entry Profile

The students who want to take this course should have:

- ❑ Knowledge of RDBMS concepts.
- ❑ Basic knowledge of computers.

Exit Profile

After completing this course, the student should be able to:

- ❑ Query and manage databases using SQL Server.

Conventions

Convention	***Indicates...***
	*Note*
	*Animation*
	*Just a Minute*

Chapter 1

Overview of SQL Server

In today's competitive environment, an organization needs a secure, reliable, and productive data platform for its business applications. SQL Server provides a platform to build and manage data applications. In addition, it combines data analysis, reporting, integration, and notification services that enable organizations to build and deploy efficient Business Intelligence (BI) solutions.

This chapter discusses the importance of a *database* server. In addition, it provides an overview of SQL Server, its components, and features. Further, this chapter introduces the *Structured Query Language (SQL)* that is used to manipulate the *data* in a database server. Lastly, it discusses the tools provided by SQL Server to improve the productivity of the database developer and manage the server.

Objectives

In this chapter, you will learn to:

- Appreciate SQL Server as a database server
- Identify the SQL Server tools

Introduction to SQL Server

Every organization needs to maintain information related to employees, customers, business partners, or business transactions. Organizations build business applications with a user-friendly interface to store and manipulate this information and to generate reports. For this, they need a platform to store and maintain this information in an efficient way. Various Database Management Systems (DBMS) and Relational Database Management Systems (RDBMS), such as SQL Server, Oracle, and Sybase, provide the platforms for storing and maintaining this information.

SQL Server is a database engine introduced by Microsoft. It provides an environment used to create and manage databases. It allows secure and efficient storage and management of data. In addition, it provides other components and services that support the business intelligence platform to generate reports and help in analyzing historical data and predicting future trends.

As a database developer, it is important for you to identify the role of a database server in an organization. You can design a database effectively if you know all the components and services of SQL Server. In addition, you need to understand the basics of SQL, a language that is used to *query* and manage data.

Role of a Database Server

Organizations have always been storing and managing business data. Earlier, organizations used to store data on paper. With an increase in the usage of computers, organizations started maintaining the same information in computers. Data was stored in an organized way, and it was also easy to retrieve data faster than before. As data retrieval became easy and fast, organizations started using business or Web applications to support the business operations.

Consider a scenario. A Web application might be used by an organization to manage the data of the employees. This application retrieves and manipulates the data stored in a data source, such as MS SQL Server and Oracle. To retrieve and manipulate the data stored in a data store, you need to use a data access technology in the application. A data access technology interacts with a data source and enables you to retrieve and manipulate data through the application. The application can retrieve data from the data source using any of the following data access technologies:

- Language-Integrated Query (LINQ)
- ADO.NET Entity Framework
- WCF Data Services
- Sync Framework

All these technologies use ADO.NET at the core level. ADO.NET further implements SQL to access data from the data store.

Business or Web applications accept data as input, process the data based on business requirements, and provide data or information as output. For example, an application maintains the details of the number of cars sold for a particular brand, such as Ferrari. Each car has a unique identification number that is already stored in an application. Whenever a sale happens, the application checks whether the unique identification number provided for the car is correct or not. If the unique identification number is correct then the sale details for the same is updated in the application. The data is saved and an output message confirming that the data has been saved is displayed to the user. This process of checking whether the unique identification number exists in the system or not is called a business rule.

Consider another scenario. The Human Resource department of an organization uses an application to manage the employee data. The users need to add the details of new employees. For this, the application provides an interface to enter the employee details. These details are validated for accuracy based on business rules. For example, a business rule is defined to check that the date of joining of the new employee is less than or equal to the current date. If the data meets the requirements, it is

saved in the data store.

Based on the preceding scenario, a business application can have the following elements:

- The User Interface (UI) or the presentation element through which data is entered.
- The application logic or the business rule element, which helps in validating the entered data.
- The data storage or the data management element, which manages the storage and retrieval of data.

These elements form the base of the models or architectures used in application development. All these elements can exist on the same computer as a single process or on different computers as different processes. Depending on the placement of these elements, the application architecture can be categorized as:

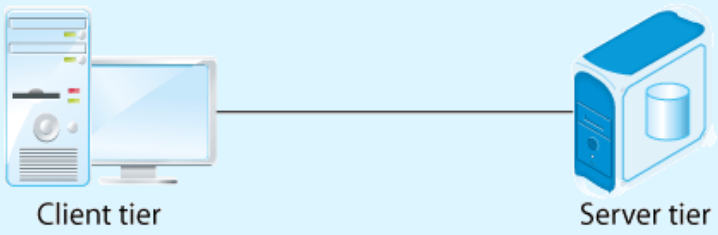
- Single-tier architecture
- Two-tier architecture
- Three-tier architecture
- N-tier architecture

Single-Tier Architecture

In a single-tier architecture, all elements of a business application are combined as a single process. If multiple users need to work on this application then it needs to be installed on the computer of every user. This type of architecture has one disadvantage. In case some errors are identified in the application then after rectifying the same, the application has to be installed again on the system of every user. This is a time-consuming process.

Two-Tier Architecture

To solve the problems of single-tier application, two-tier architecture was introduced. In two-tier architecture, the application is divided into two parts. One part handles the data, while the other provides the user interface. Therefore, this architecture is called two-tier architecture. These two parts can be located on a single computer or on separate computers over a network. The part that handles the UI is called the *client tier*. The part that implements the application logic and validates the input data based on the business rules is called the *server tier*, as shown in the following figure.



A Two-Tier Architecture

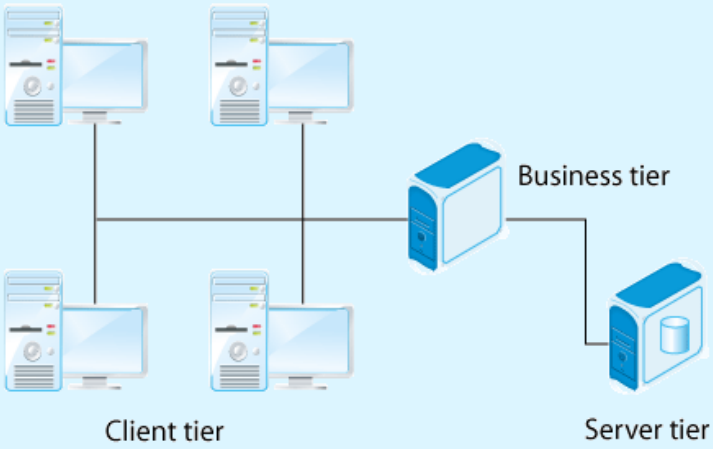
In the preceding architecture, the maintenance, upgrade, and general administration of data is easier, as it exists only on the server and not on all the clients.

A two-tier architecture is also called the client-server architecture. A client sends the request for a service and a server provides that service. Most RDBMSs, such as Microsoft Access, SQL Server, and Oracle, support client-server architecture. RDBMS provides centralized functionality while supporting many users.

Three-Tier Architecture

When implementing complex business solutions in a two-tier architecture, the tier on which the business logic is implemented becomes over loaded. As a result, it takes more time to execute. Therefore, to provide further flexibility, the two-tier architecture can be split into three tiers. In three-tier architecture, the first tier is the client tier. The second or the middle tier is called the *business tier*. The third tier is called the server tier. The server tier contains a database server that manages the data.

In this architecture, an additional tier called a *business tier* has been added between the client tier and the server tier, as shown in the following figure.



A Three-Tier Client/Server Architecture

The business tier consists of all the business rules. It consists of the application logic that implements business rules and validates the data. The advantage of a three-tier application is that it allows you to change the business rules without affecting the other two tiers.

For example, in a banking application for loans, the user tier is the frontend used by the customer to specify the loan details. The server tier can consist of an RDBMS in which the data is stored. The business tier lies between the other two tiers and consists of business rules, such as the loan limit and the interest rate charged to a customer. If there is a change in the rate of interest, only the middle tier component needs to be modified.

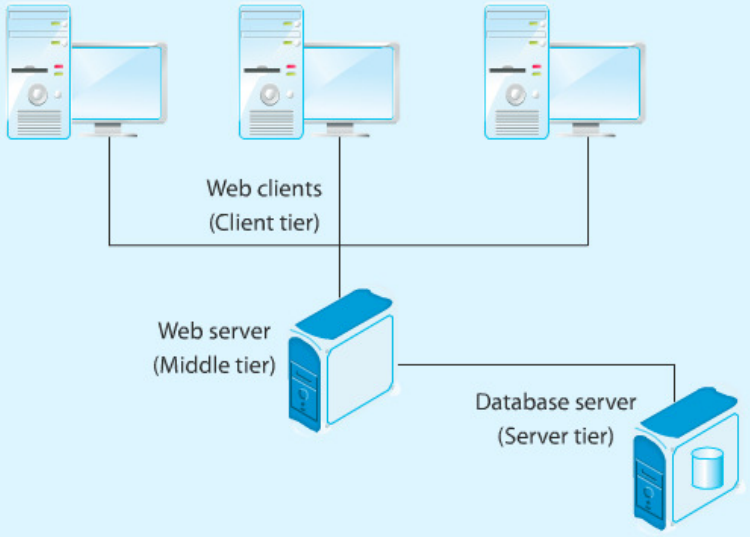
N-Tier Architecture

As the business complexities increased, the business tier became larger and unmanageable. This led to the evolution of n-tier architecture, where the business services model was divided into smaller manageable units. N-tier

architecture is also called a multi-tier architecture.
In this architecture, one component near the client tier is responsible to do the client side validation and send the data to the presentation tier. Therefore, it is possible to keep the UI-centric processing component on a computer near the client. The UI-centric processing component is responsible for processing the data retrieved from and sent to the presentation tier. In addition, you may have another component near the database server to manipulate and validate the data. You can keep the data-centric processing components on another computer near the database server, so that you gain significant performance benefits. Data-centric processing components are responsible for accessing the data tier to retrieve, modify, and delete data to and from the database server.
The n-tier architecture consists of the following layers:

- Client tier
- UI-centric processing components
- Data-centric processing objects
- Database server

The banking application, when further expanded, can represent an example of n tier architecture. The client tier would consist of the user interface, which would include the user interface controls, such as forms, menus, and toolbars. The server tier would consist of data-handling including saving data to the database server.
The business logic would include the rules and guidelines for different types of accounts, interest rates, fixed deposits, ATMs, and loan rules. All of these would form the middle tier. However, there would be some rules that need to be implemented both on the user interface and on the database. You can place these rules either on the UI-centric processing components or data-centric processing components, based on the functionality.
Applications that follow multi-tier architecture can be used across various locations. For example, in Web applications, an application is stored on the Web server. The clients access the application from any location through a browser. The clients make requests to the Web server and receive responses.
The following figure shows the architecture of the Web applications.



The Architecture of the Web Applications

Depending on the type of business rules, the applications can be implemented on any of the tiers, such as Web clients, Web server, or the database server.
To provide support to applications where users can send requests simultaneously, the database server needs to be fast, reliable, and secure. SQL Server is one such complete database platform that provides a fast, reliable, and secure RDBMS. It also helps in data analysis with integrated BI tools. The BI tools are used to prepare reports that are analyzed further to make efficient business decisions.

SQL Server Elements

SQL Server contains a number of elements. Each element in SQL Server provides various tools and services to improve the database efficiency. The various elements in SQL Server, such as SQL Server Database Engine, Analysis services, and SQL Server Data tools, working together constitute its architecture. However, you need to understand the most important and basic elements of SQL Server that are frequently used while querying data.
SQL Server has the following elements:

Services

Services are the applications that are started automatically during the startup of the computer system. Services provide support to create and manage databases in different environments. Multiple services are set up on the computer system when the SQL Server is installed, which includes some of the following services:

- **SQL Server Database Engine**: It provides support to store, query, process, and secure data on a database server. It allows you to create and manage database objects such as tables, views, stored procedures, and triggers.
- **SQL Server Agent**: It is used to automate administrative tasks by creating and managing jobs, alerts, and operators.

- **SQL Server Data Tools**: It allows developers to create, debug, and maintain databases and database objects using an integrated development environment offered by Microsoft Visual Studio.

Instances

An instance is the basic unit of program installation and program execution. It is a copy of the .EXE file that represents the programs and resources supported on SQL Server. You can install multiple instances of SQL Server on a single computer. Each instance is segregated from other instances based on disk files usage and resources allocation. SQL Server provides the following types of instances:

- **Default Instance**: The default instance of SQL Server is identified by the name of the computer on which the instance is running. SQL Server allows you to install only one default instance of SQL Server on a system.
- **Named Instances**: The named instance of SQL Server is identified by both, the computer name and the instance name. SQL Server allows you to create more than one named instance of SQL Server on a system.

Tools

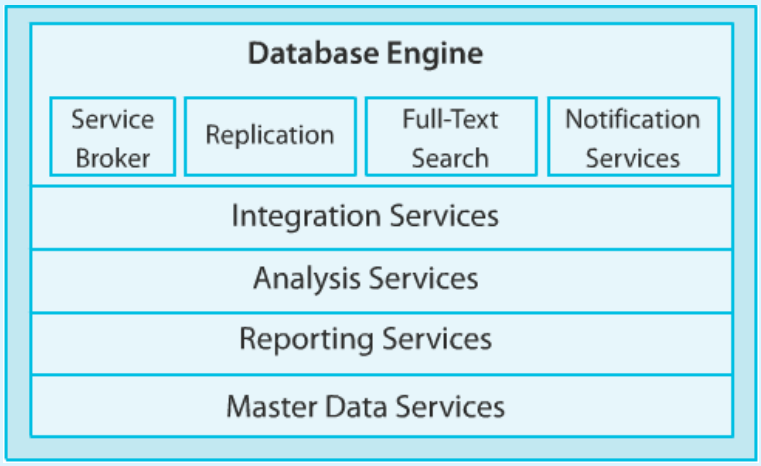
SQL Server provides various tools that enable you to query and manage the database engine. These tools are:

- **SQL Server Management Studio**: It is a powerful tool that provides a simple and integrated environment to develop and manage the SQL Server database objects.
- **SQLCMD Tool**: It allows you to execute T-SQL statements at the command prompt mode.
- **SQL Server Configuration Manager Tool**: It manages utility services, such as client network utility, server network utility, and service manager utility services.
- **SQL Server Installation Center**: It provides you the ability to add, delete, or modify the features of SQL Server, such as adding a new feature to an existing server or upgrading SQL Server.

SQL Server Components

SQL Server contains a number of components. Each component provides specific services and support to the clients connected to the server.

The following figure displays the components of SQL Server.



The Components of SQL Server

As shown in the preceding figure, SQL Server consists of the following core components:

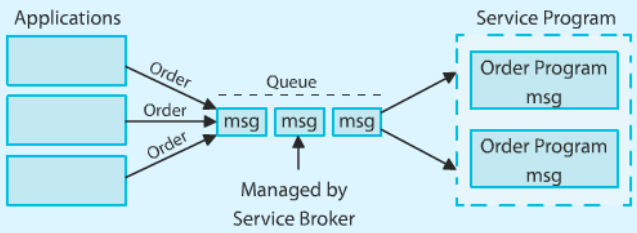
- Database engine
- Integration services
- Analysis services
- Reporting services
- Master Data services

Database Engine

A database engine provides support to store, query, process, and secure data on a database server. It allows you to create and manage database objects, such as tables, views, stored procedure, and triggers. Apart from providing support for data management, a database engine also provides the following background services:

- **Service Broker**: Provides support for asynchronous communication between clients and the database server, thereby enabling reliable query processing. The client application sends a request to the database server and continues to work. These requests are queued up at the server in case the server is not available to process the request immediately. A Service Broker ensures that the request is processed whenever the server is available.

  The following figure shows the implementation of Service Broker in the order processing system.



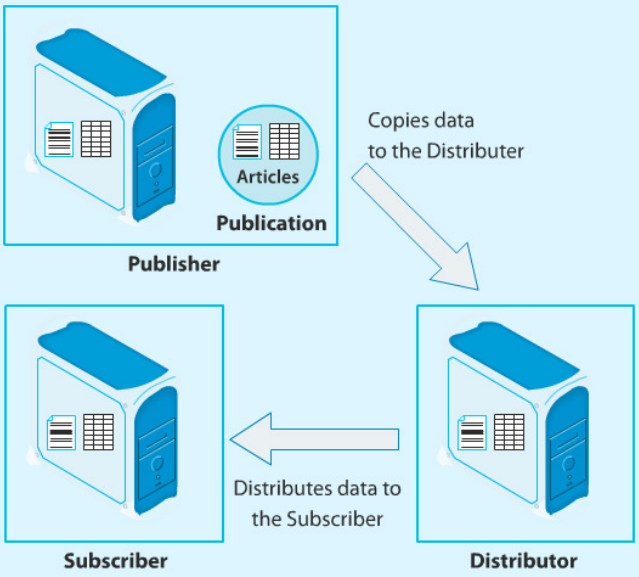
The Implementation of Service Broker in the Order Processing System

  The preceding figure describes the example of the order processing system. The client applications are sending orders to the database server to enter the order details. All these orders are placed in a *queue*, which is managed by the Service Broker.

- **Replication**: Allows you to copy and distribute

data and database objects from one database server to another. These servers can be located at remote locations to provide fast access to users at widely distributed locations. After replicating data, SQL Server allows you to synchronize different databases to maintain data consistency. For example, the database servers for your organization might be located at different locations around the world, but all the servers store common data. To ensure that the data in all the servers is synchronized, you can implement data replication. Replication follows the publisher/subscriber model. In this model, the changes are sent out by one database server ("publisher") and are received by others ("subscribers").

The following figure depicts the replication process.



The Replication Process

In the preceding figure, articles are the database objects to be replicated. These articles are stored in a database called publication and are located on the publisher server. The distributor server takes the publications from the publisher server and distributes them to the subscribers.

- **Full-text search**: Allows you to implement fast and intelligent search in large databases. It allows you to search records containing certain words and phrases. You can search for different forms of a specific word, such as 'produce', 'produces', or 'production'. In addition, you can search for synonyms of a given word, such as 'garment', 'cloth', or 'fabric'.
- **Notification services**: Allow you to generate and send notification messages to the users or administrators about any event. For example, the database administrator should be notified when a table is created or deleted. The notification messages can be sent to a variety of devices, such as computers or mobile devices. Notification service is a platform for developing and deploying highly scalable notification applications. It allows developers to build notification applications that send timely, personalized information updates, helping to enhance customer relationships. For example, a brokerage firm sends stock and fund prices based on the customer's preferences.

Integration Services

Data in different sources might be stored in different formats and structures. Integration services allow you to gather and integrate this varied data in a consistent format in a common database called the *data warehouse*. A data warehouse consists of integrated databases, which can be a DBMS, text files, or flat files. A data warehouse is similar to a physical warehouse that stores raw material or products for further distribution. The organization does not store useless materials or products in its warehouse because it costs money and affects the ability to get products in and out of the warehouse. Similarly, a data warehouse should not contain useless data. The data should be meaningful so that it could be processed quickly. A data warehouse is a large central repository of data that helps in decision-making.

Consider a telecommunications company, where the CEO notices that the frequency of cancellation of services by its customers in the past one year has increased considerably. The company is unable to analyze the service preferences of the customers because data is scattered across disparate data sources. These data sources contain data spanning two decades. In such a case, a data warehouse can be implemented to integrate two decades of historical data from disparate data sources. The integrated data will provide a holistic view of the customers to the CEO.

SQL Server Integration Services (SSIS) Import and Export Wizard provides a series of *dialogs* to help you complete the process of selecting the data source, the destination, and the objects that will be transferred to create a data warehouse.

Analysis Services

Data warehouses are designed to facilitate reporting and analysis. Enterprises are increasingly using data stored in data warehouses for analytical purposes to assist them in making quick business decisions. The applications used for such analysis are termed as BI applications. Data analysis assists in determining past trends and formulating future business decisions. This type of analysis requires a large volume of data to reach a consistent level of sampling.

In the telecommunications company scenario, with the help of the analysis tools querying on the data warehouse, the CEO can identify the customers who are canceling their services. The company can then use this information

to provide attractive offers to the identified customers and build loyalty. This kind of information analysis proves to be beneficial to the enterprise in the long run. The enterprise can retain its customers by offering loyalty programs and schemes on the basis of analysis on the historical data.

Consider another example of a soft drink manufacturer that uses data of the past few years to forecast the quantity of bottles to be manufactured in the current month. These forecasts are based on various parameters, such as the average temperature during the last few years, purchasing capacity of the customers, age group of the customers, and past trends of consumption. The requirements for such an analysis include:

- A large volume of data
- Historical data, that is, data stored over a period of time

Therefore, analysis services help in data analysis in a BI application. Microsoft SQL Server Analysis Services (SSAS) provide Online Analytical Processing (OLAP) for BI applications. OLAP arranges the data in the data warehouse in an easily accessible format. This technology enables data warehouse to do online analysis of the data.

Reporting Services

Reporting services provide support to generate complete reports on data in the database engine or in the data warehouse. These services provide a set of tools that help in creating and managing different types of reports in different formats. Using these services, you can create centralized reports that can be saved to a common server. Reporting services provide secure and restricted access to these reports.

Microsoft SQL Server Reporting Services (SSRS) help in creating Web-based reports based on the content stored in a variety of data sources. You can also publish these reports in different formats.

Master Data Services

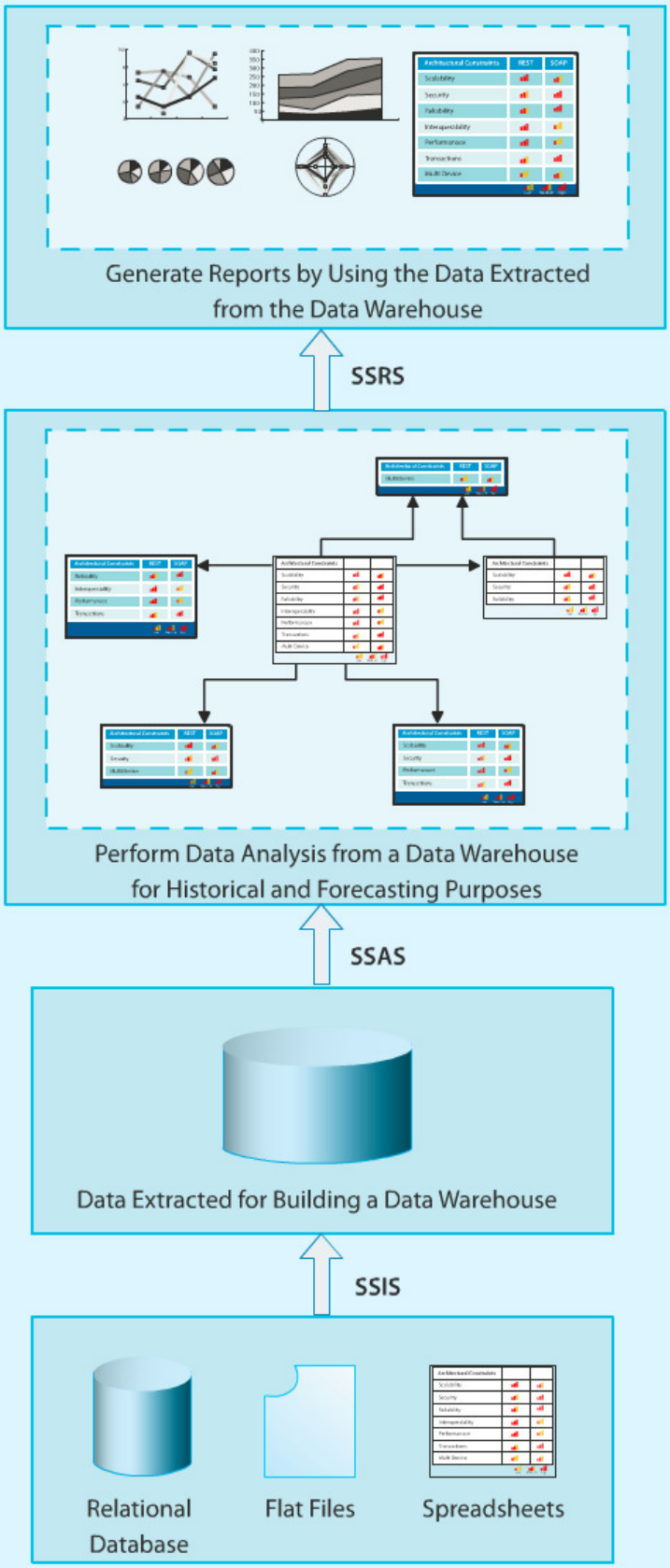
Master Data Services (MDS) is used to manage the master data for an organization. Master Data includes information that is vital for an organization's business operations. It generally consists of data, which is transactional in nature and can be used by multiple users, systems, applications, and processes across an organization. It may include data about customers, products, employees, assets, locations, and vendors, who require continuous quality management, ease of access, and effective sharing of data.

MDS is a Microsoft technology for implementing master data management. It is a part of the Enterprise and Business Intelligence editions of SQL Server. One of the fundamental tasks in MDS implementation is to identify the data that is critical for an organization. This data is referred to as entities. Further, the *attributes* associated with each entity are identified. For example, an entity can be a customer, a student, or a product. The attributes of entities can be the order placed, name, or quantity. MDS implementation ensures that correct values are provided for each attribute of an entity.

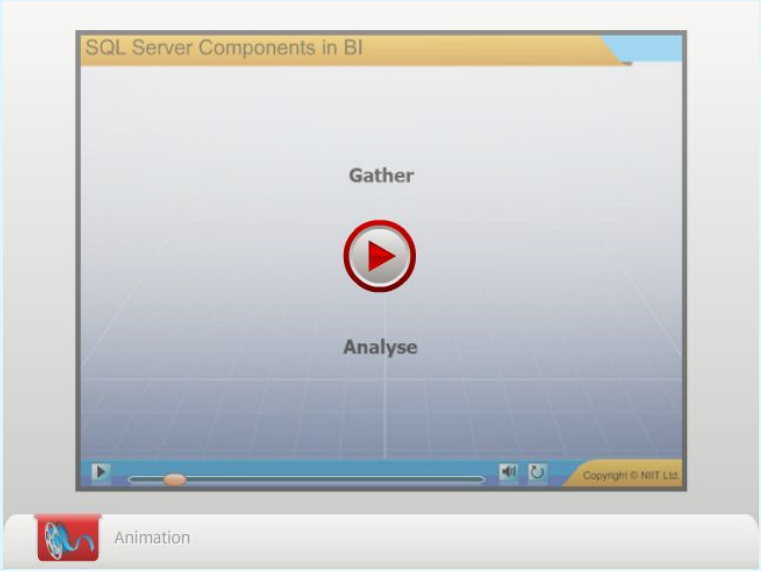
MDS is actually used to formulate a data model. It includes the following features to manage data, and enhances the quality and performance:

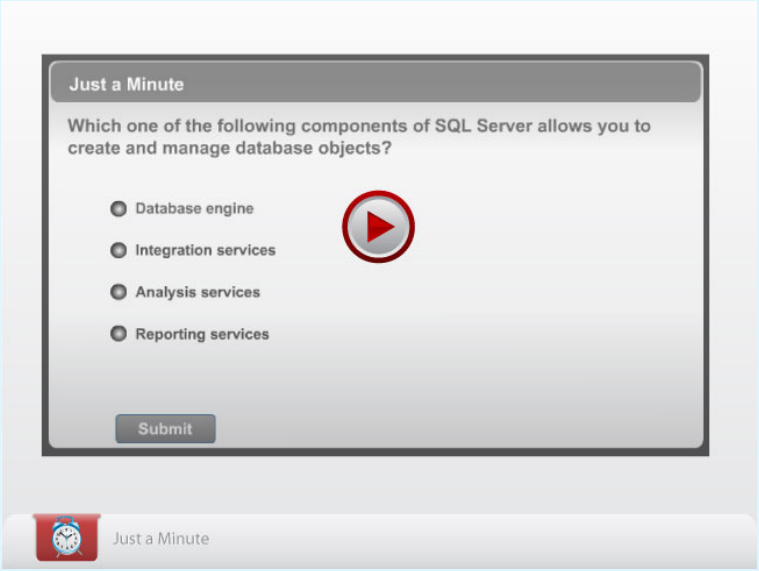
- **Versioning**: Organizations have different views of particular data on the basis of time, business entities, or external changes. For example, an organization wants to view the monthly, weekly, and quarterly sales to calculate the profitability ratio. MDS provides the ability to view the sales record at different time intervals, so that the users can work easily on various versions.
- **Security**: MDS ensures that the sensitive data is protected. The data is protected by limiting user access for performing specific administrative tasks, and viewing or updating selected entities or attributes.
- **Global Access to Master Data**: MDS provides a Web application that acts as a portal to enable users to access master data without installing any client application on their desktops.
- **Business Rules**: MDS enforces business rules to ensure the accuracy of master data by updating values, setting *default* values, and sending an email when some new data is added or removed.

The following figure shows the usage of the various SQL Server core components in a BI application.



The Core Components of SQL Server





SQL Server Editions

SQL Server supports various editions that provide different features targeting different business requirements. SQL Server has the following editions:

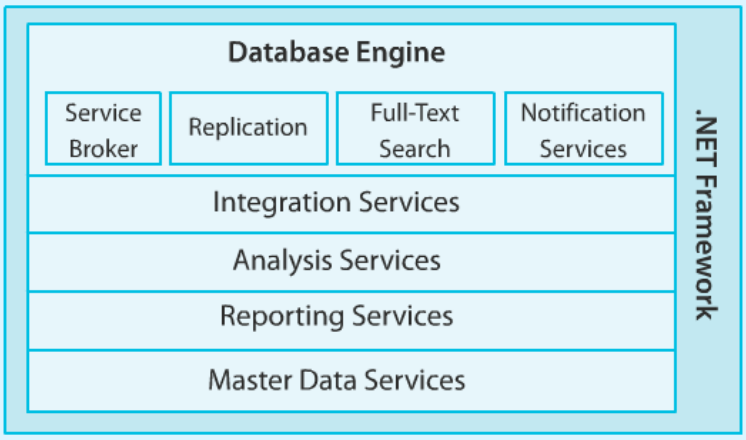
- **Enterprise**: The SQL Server Enterprise edition includes the core database engine and supports the advanced services such as table partitioning, parallel indexing, and indexed views. The Enterprise edition is designed to provide high performance and security, optimized productivity, and high scalability of data.
- **Standard**: The SQL Server Standard edition has core database engine and basic business intelligence capabilities. It differs from the Enterprise Edition in that it does not support all the security and data warehousing features of the Enterprise edition. The Standard edition provides easy integration, improved manageability, and easy extension of data across various platforms, such as .Net or Linux.

- **Business Intelligence**: The BI edition has been introduced in SQL Server 2012. It consists of all the Standard Edition capabilities and supports the business intelligence tools, such as Master Data Services and Data Quality Services. The Business Intelligence edition provides rapid exploration of data and greater consistency, manageability, and security of data.

Data Quality Services helps the database administrator to maintain the quality of the data.

SQL Server Integration with the .NET Framework

Microsoft SQL Server is integrated with the *.NET Framework*, as shown in the following figure.



Integration of SQL Server with the .NET Framework

The .NET Framework is an environment used to build, deploy, and run business applications. These applications can be built in various programming languages supported by the .NET Framework. It has its own collection of services and classes. It exists as a layer between the .NET applications and the underlying operating system.

SQL Server uses various services provided by the .NET Framework. For example, the notification services component is based on the .NET Framework. This component uses the .NET Framework services to generate and send notification messages.

The .NET Framework is also designed to make improvements in code reuse, code specialization, resource management, multilanguage development, security, deployment, and administration. Therefore, it helps bridge the gap of interoperability between different applications.

The .NET Framework consists of the following components:

- Development tools and languages
- Base class library
- *Common Language Runtime (CLR)*

Development Tools and Languages

Development tools and languages are used to create the interface for the Windows forms, Web forms, and console applications. The tools include Visual Studio and Visual C# Developer. The languages that can be used are Visual Basic.NET, C#, or F#. These components are based on the .NET Framework base classes. These tools also enable you to create database-related codes and objects without switching to the SQL Server database engine. This enables you to work with database objects quickly and easily.

Base Class Library

The .NET Framework consists of a class library that acts as a base for any .NET language, such as Visual Basic, .NET, and C#. This class library is built on the object-oriented nature of the runtime. It provides classes that can be used in the code to accomplish a range of common programming tasks, such as string management, data collection, database connectivity, and file access. In addition, it enables you to connect with a database and generate scripts or create queries for accessing data.

CLR

CLR is one of the most essential components of the .NET Framework. It provides an environment for the application to run. CLR or the runtime provides functionalities, such as exception handling, security, debugging, and versioning support to the applications.

Some of the features provided by CLR are:

- **Automatic memory management**: Allocates and de-allocates memory to the application as and when required.
- **Standard type system**: Provides a set of common data types in the form of Common Type System (CTS). This means that the size of integer and long variables is the same across all programming languages.
- **Language interoperability**: Provides the ability of an application to interact with another application written in a different programming language. This also helps maximize code reuse.
- **Platform independence**: Allows execution of a code from any platform that supports the .NET CLR.
- **Security management**: Applies restrictions on the code to access the resources of a computer.

CLR can host a variety of languages. It offers a common set of tools across these languages, ensuring interoperability between the codes. The code developed with a language compiler that targets CLR is called a *managed code*.

Alternatively, the code that is developed without considering the rules and requirements of the common language runtime is called *unmanaged code*. Unmanaged code executes in the common language runtime environment with minimal services. For example,

unmanaged code may run with limited debugging and without the garbage collection process.

With CLR integration in SQL Server, you can implement programming logics that involve complex operations in the database by using the programs written in any of the .NET-supported languages. CLR integration allows you to create objects in a .NET-supported language and embed these objects in the database. You can embed the .NET code in database objects such as stored procedure, function, or trigger. Such a database object is called a *managed database object.*

Features of SQL Server

The components of SQL Server help improve the database management and developer productivity by the following features of SQL Server:

- **Built-in support for *Extensible Markup Language (XML)* data**: Allows you to store and manage XML data in variables or columns of the XML data type. The XML feature of SQL Server enables you to write code to retrieve data from the database in the form of XML. In addition, it allows you to read an XML document and store the XML data in the database.
- **CLR integration**: Allows you to use the CLR features of .NET Framework in the SQL Server database. It enables you to use the code written in any of the .NET supported languages for implementing complex programming logics in the database. For example, you need to write a code for the verification of the credit card number entered by the user. It will be complex to write the code for the verification of the credit card number in T-SQL. However, the same code can be written effectively using a .NET programming language. Therefore, you can write the code in a .NET programming language and use that code in SQL Server to verify the credit card information.
- **Scalability**: Allows you to distribute the data in large tables into several *filegroups*. This enables SQL Server to access all the filegroups simultaneously and retrieve the data quickly. This makes the database scalable and helps improve the performance of queries.
- **Service-oriented architecture**: Provides distributed and asynchronous application framework for large-scale applications. This allows the database clients to send requests to the database server even if the server is not available to process the request immediately.
- **Support for Web services**: Allows you to provide direct access to the data from the Web services by implementing the HTTP endpoints. For example, sales executives of an organization need to access the data on the database server through their Personal Desktop Assistant (PDA) devices. However, providing direct access from a PDA device to the database server involves a high cost. Therefore, organization can implement Web services through which each sales executive can log the sales details online from anywhere using any device.
- **High level of security**: Implements high security by enforcing policies for log on passwords. Administrators can also manage permissions on database objects granted to different users.
- **High availability**: Ensures that the database server is available to all users at all times. This reduces the downtime of the server. In SQL Server, high availability is implemented with the help of database mirroring, failover clustering, and database snapshots.
- **Support for data migration and analysis**: Provides tools to migrate data from different data sources to a common database. In addition, it allows building the data warehouse on this data that can support BI applications for data analysis and decision-making.
- **Intellisense**: Provides the feature of auto completion of code written to create or manipulate database objects in the Query Editor window.
- **Policy-based management**: Used to define a set of policies for configuring and managing SQL Server. For example, you can define a policy to set a naming convention for tables and stored procedures. When a user tries to create a table, the table name must map with the naming convention defined in the policy, else an error will be raised.
- **Resource governor**: Used to manage the workload of SQL Server by allocating and managing the server resources, such as CPU time and memory. The resource pool represents the server resource. You can specify the minimum and maximum values of the CPU and memory utilization in a resource pool. These resources are used for running and performing various assigned tasks in SQL Server.

Types of SQL Statements

As a database developer, you need to manage the database to store, access, and modify data. SQL is the core language used to perform these operations on the data. SQL, pronounced as ‘sequel’, is a language that is used to

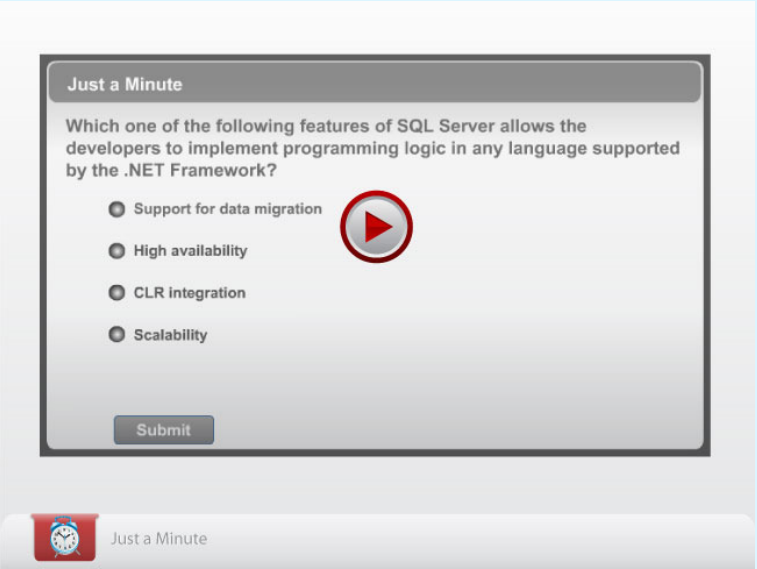
manage data in an RDBMS. This language was developed by IBM in the 1970s. It follows the International Organization for Standardization (ISO) and American National Standards Institute (ANSI) standards.

Most database systems have created customized versions of the SQL language. For example, Transact-SQL (T-SQL) is a scripting language used in SQL Server for programming. The applications may have different user interfaces but have a common way to communicate with SQL Server by sending T-SQL statements to the server.

The SQL statements can be divided into the following categories:

- **Data Definition Language (DDL)**: Is used to define the database, data types, structures, and *constraints* on the data. Some of the DDL statements are:
  - **CREATE**: Used to create a new database object, such as a table.
  - **ALTER**: Used to modify the database objects.
  - **DROP**: Used to delete the objects.
- **Data Manipulation Language (DML)**: Is used to manipulate the data in database objects. Some of the DML statements are:
  - **INSERT**: Used to insert a new data record in a table.
  - **UPDATE**: Used to modify an existing record in a table.
  - **DELETE**: Used to delete a record from a table.
- **Data Control Language (DCL)**: Is used to control the data access in the database. Some of the DCL statements are:
  - **GRANT**: Used to assign permissions to users to access a database object.
  - **REVOKE**: Used to deny permissions to users to access a database object.
- **Data Query Language (DQL)**: Is used to query data from database objects. SELECT is the DQL statement that is used to select data from the database in different ways and formats.

SQL is not a case-sensitive language. Therefore, you can write the statements in any case, lowercase or uppercase. For example, you can use the SELECT statement in lowercase as 'select' or in title case as 'Select'.



Identifying SQL Server Tools

SQL Server provides various tools that help improve the efficiency of database developers. SQL Server Management Studio is one such tool that helps in creating and maintaining database objects. SQL Server Data Tools help in creating and implementing BI solutions. SQL Server also provides tools, such as Database Engine Tuning Advisor and SQL Server Configuration Manager that help the database administrator in configuring the server and optimizing its performance.

Before you start working on SQL Server, it is important to identify the various tools and their features provided by SQL Server.

SQL Server Management Studio

SQL Server Management Studio is a powerful tool associated with SQL Server. It provides a simple and integrated environment for developing and managing the SQL Server database objects. The various components of SQL Server Management Studio, such as query editor, object explorer, and solution explorer are used to create, store, and execute queries. SQL Server Management Studio provides the facility to view the execution plans of queries in different formats.

The following table lists the main components of the SQL Server Management Studio interface.

Components	***Description***
Object Explorer	*An Object Explorer window provides the ability to register, browse, and manage servers. Using Object Explorer, you can also create, browse, and*

	manage server components. The Object Explorer allows you to configure the following components: ❑ ***Security**: Used to create login and users, and to assign permissions.* ❑ ***Replication**: Used to create and manage publishers and subscribers.* ❑ ***SQL Server Agent**: Used to automate administrative tasks by creating and managing jobs, alerts, and operators.* ❑ ***Management**: Used to configure Distributed Transaction Coordinator, Database Mail service, or SQL Server logs. In addition, it is used for managing policies and governing resources of SQL Server.* ❑ ***Server Objects**: Used to create and manage backups, endpoints, and triggers.*
Object Explorer Details	*The Object Explorer Details provide the detailed description of all the objects in SQL Server.*
Registered Servers	*The Registered Servers window displays all the servers registered with the management studio. It also helps record connection information for each registered server including the authentication type, default database, network protocol characteristics, encryption, and time-out parameters.*
Solution Explorer	*The Solution Explorer window provides an organized view of your projects and files. In this explorer, you can right-click on a project or file to manage or set their properties.*
Query Editor	*The Query Editor window provides the ability to execute queries written in T-SQL. It can be invoked by selecting the New Query option from the File menu or the New Query button from the Standard toolbar.*
Template Explorer	*The Template Explorer window provides a set of templates of SQL queries to perform standard database operations. You can use these queries to reduce the time spent in creating queries.*
Dynamic Help	*The Dynamic Help window is available from the Help menu of SQL Server Management Studio. This tool automatically displays links to relevant information while users work in the Management Studio environment.*

The Components of the SQL Server Management Studio Interface

Task 1.1: Opening SQL Server Management Studio

SQL Server Data Tools

SQL Server Data Tools (SSDT) is a replacement for Business Intelligence Development Studio and is used to work with relational databases and BI projects. It allows developers to create, debug, and maintain databases and database objects by using an integrated development environment offered by MS Visual Studio. SQL Server Object Explorer in Visual Studio gives a view of the database objects. This view is similar to as offered by SQL Server management studio. Database objects like tables, views, or procedures can be created, modified, or deleted by using the options available in SQL Server Object Explorer. SSDT offers tools and project types to work with

the Analysis, Reporting, and Integration services. Therefore, SSDT has combined various database development tools into a single IDE.

Database Engine Tuning Advisor

Database Engine Tuning Advisor helps database administrators to analyze and tune the performance of the server. To analyze the performance of the server, the administrator can execute a set of T-SQL statements against a database. After analyzing the performance of these statements, the tool provides recommendations to add, remove, or modify database objects, such as indexes or indexed views to improve performance. These recommendations help in executing the given T-SQL statements in the minimum possible time.

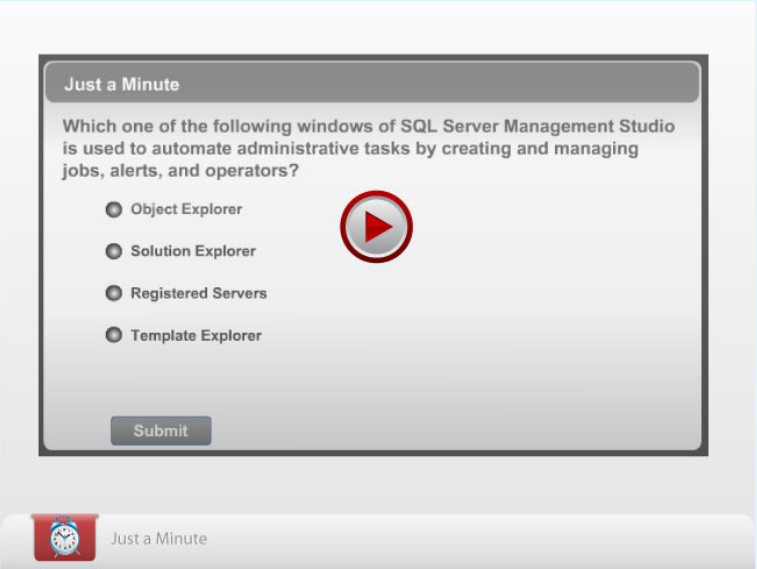
SQL Server Configuration Manager

SQL Server Configuration Manager helps the database administrators to manage the services associated with SQL Server. These services include SQL Server Agent, SQL Server Analysis Services, SQL Server Reporting Services, SQL Server Integration Services, and SQL Server Browser service. Administrators can start, pause, resume, or stop these services by using this tool. In addition, it allows you to configure certain properties, such as server alias, network protocols, and connection parameters.

In addition, the tool allows you to manage the network connectivity configuration from the SQL Server client computers. It allows you to specify the protocols through which the client computers can connect to the server.

SQL Server Profiler

SQL Server profiler helps in monitoring the events, such as login connections, execution of DML statements, stored procedures, *batches*, and security permission checks that are generated within an instance of a database engine. Data captured from these events are used for analysis purposes. In addition, Server Profiler provides an in depth view of the query submission, access of the database against the queries, and return of results after processing of queries. By using SQL Server Profiler, database can be changed and the outcomes of the changes on different database objects are analyzed.



Summary

In this chapter, you learned that:

- A business application can have three elements: user interface, business logic, and data storage.
- A database server is used to store and manage the database in a business application.
- SQL Server consists of the five core components: database engine, integration services, analysis services, reporting services, and master data services.
- A database engine provides support to store, query, process, and secure data on a database server. Integration services allow you to gather and integrate this varied data in a consistent format in a common database called the data warehouse.
- Analysis services assist in determining past trends and formulating future business decisions.
- Reporting services provide support to generate comprehensive reports on the data stored in the database engine or the data warehouse.
- Master Data services provide support to manage master data.
- Microsoft SQL Server is integrated with the .NET Framework.
- The .NET Framework is an environment used to build, deploy, and run business applications.
- The .NET Framework consists of three components: development tools and languages, base class library, and CLR.
- SQL Server provides the following benefits:
  - Built-in support for XML data
  - CLR integration
  - Scalability
  - Service-oriented architecture
  - Support for Web services

- High level of security
- High availability
- Support for data migration and analysis
- Intellisense
- Policy-based management
- Resource governor

- The SQL statements can be divided into the following categories:
  - **DDL**: To create and manage database objects.
  - **DML**: To store and manage data in database objects.
  - **DCL**: To allow or deny access to database objects.
  - **DQL**: To query data from database objects.
- SQL Server provides the following tools to improve the efficiency of the database developers and manage the server:
  - SQL Server Management Studio
  - SQL Server Data Tools
  - Database Engine Tuning Advisor
  - SQL Server Configuration Manager
  - SQL Server Profile

Reference Reading

Introduction to SQL Server

Reference Reading: Books	***Reference Reading: URLs***
Introducing Microsoft SQL Server 2012 by Ross Mistry and Stacia Minser	*http://msdn.microsoft.com/en-us/library/ms144275.aspx*

Identifying SQL Server Tools

Reference Reading: Books	***Reference Reading: URLs***
Beginning Microsoft SQL Server2012 Programming by Paul Atkinson and Robert Vieira	*http://msdn.microsoft.com/en-us/library/hh272686%28v=vs.103%29.aspx*

Chapter 2

Querying Data

As a database developer, you need to regularly retrieve data for various purposes, such as creating reports and manipulating data. You can retrieve data from a database server by using SQL queries.

This chapter explains how to retrieve specific data from database tables by executing SQL queries. Further, it discusses how to incorporate functions to customize the data values returned by the queries. In addition, the chapter explains how to retrieve summarized and grouped data from database tables.

Objectives

In this chapter, you will learn to:

- Retrieve data
- Use functions to customize the *result set*
- Summarize and group data

Retrieving Data

At times, database users might need to retrieve complete or selected data from a table. Depending on the requirements, they might need to extract only selected columns or *rows* from a table. For example, an organization stores the employee data in the Employee table of the SQL Server database. If the HR Manager of the organization needs to send a letter to employee, only selected information, such as name and address details, of the employee needs to be retrieved from the Employee table. However, if the HR Manager has to prepare a report of the employees, all the employee details need to be retrieved from the Employee table.

Depending on these requirements, you will need to run different SQL queries to specify the criteria for selecting data from the tables. Therefore, it is important for you to learn how to query databases to retrieve the required information.

Databases can contain different types of data. Therefore, before querying the data, it is important to identify the various data types.

Identifying Data Types

Data type represents the type of data that a database object can contain. This data can be in the form of characters or numbers. You can store the following types of data in a database:

- **Structured data**: Is the data that has definite structure. It follows a defined format and has a predefined length, such as string or numeric data. The structured data can be easily stored in a table. For example, the roll number of a student is a numeric value and can be easily stored in the RollNo column of the Student table. In addition, you can store the spatial data, such as geographical data with longitude and latitude, and geometrical data with coordinates.
- **Semi-structured data**: Is the data that has an indefinite structure and cannot be easily stored in a database table. The attributes of semi-structured data can change. For example, the details of the customers contain various attributes, such as customer ID and customer name. These attributes change for every customer. Semi-structured data is generally stored in the XML format.
- **Unstructured data**: Is the data that does not have a defined structure. It does not necessarily follow any format or sequence. In addition, it does not follow any rule and is not predictable. For example, text files, image files, streaming videos, and sound clips represent unstructured data.

Depending upon the type of data to be stored, SQL Server supports the various data types. These data types can be associated with the columns, *local variables*, or expressions defined in the database. You need to specify the data type of a column according to the data to be stored in the column. For example, you can specify the character data type to store the employee name, date data type to store the hire date of employees. Similarly, you can specify money data type to store the salary of the employees.

The data types supported in SQL Server can be organized into the following categories:

- Numeric
- Character string
- Date and time
- Binary
- Others

Numeric

The numeric data types allow you to store integer values in the database. The following table lists the numeric data types supported by SQL Server.

Data types	Range	Used to store	Storage (Bytes)
int	–2^31 (–2,147,483,648) to 2^31–1 (2,147,483,647)	Integer data (whole numbers)	4
smallint	–2^15 (–32,768) to 2^15–1 (32,767)	Integer data	2
tinyint	0 to 255	Integer data	1
bigint	–2^63 (–9,223,372,036,854,775,808) to 2^63–1 (9,223,372,036,854,775,807)	Integer data	8
decimal	–10^38 +1 through 10^38–1	Numeric data types, with a fixed precision and scale	5 to 17
numeric	–10^38 +1 through 10^38–1	Numeric data types, with a fixed precision and scale	5 to 17
float	–1.79E+308 to –2.23E–308, 0 and 2.23E–308 to 1.79E+308	Floating precision data	4 to 8
real	–3.40E + 38 to –1.18E–38, 0 and 1.18E–38 to 3.40E + 38	Floating precision number	4
money	–922,337,203,685,477.5808 to 922,337,203,685,477.5807	Monetary data	8
smallmoney	–214,748.3648 to 214,748.3647	Monetary data	4

The Numeric Data Types

Character and String

The character and string data types allow you to store text values in the database. The following table lists the character and string data types supported by SQL Server.

Data types	Range	Used to store	Storage (Bytes)
char(n)	n characters, where n can be 1 to 8000	Fixed length character data	N
varchar(n)	n characters, where n can be 1 to 8000	Variable length character data	n+2
text	Maximum length of 2^31–1 (2,147,483,647) characters	Character string	Variable size
ntext	Maximum length of 2^30–1 (1,073,741,823) characters	Variable length Unicode data	Variable size
nchar	Maximum length of 4000 characters	Fixed length Unicode data	n (length of data) *2
nvarchar	Maximum length of 4000 characters	Variable length Unicode data	n (length of data) *2+2

The Character and String Data Types

The preceding table lists various data types, such as char, varchar, nchar, and nvarchar, which are used for storing character data. However, you must know the difference in using these data types. The following list describes the differences between these data types:

- **char**: It is used to store fixed length character data in a variable. For example, you declare a variable of char data type with 50 character size. This declaration reserves the memory space required to store 50 characters in that variable. If you store 10 characters in that variable, then the remaining 40 character memory space will be wasted.
- **varchar**: It is used to store variable length character data in a variable. It does not reserve the memory space at the time of variable declaration. Instead, it allocates memory as per the size of the data stored in the variable. For example, you declare a variable of varchar data type with 100 character size. Now, you store a word containing 50 characters in that variable. In this case, the system will allocate memory space required to store 50 characters only. The space for the rest 50 characters that can be stored in the variable will be freed. This way, there is no memory loss. It stores only nonunicode characters.
- **nvarchar**: It allocates memory similar to the varchar data type. However, it stores only unicode characters. It requires twice the storage space as required by varchar.
- **nchar**: It supports fixed-length unicode data with the maximum length of 4000 characters. It requires twice as much storage space as required by the char data type.

Unicode is an industry wide computing standard for representing characters. It provides unique number to every character so that these characters are represented in the same way across different platforms and languages.

Date and Time

The date and time data types allow you to store date and time values in the database. The following table lists the date and time data types supported by SQL Server.

Data types	Range	Used to store	Storage (Bytes)
datetime	January 1, 1753, through December 31, 9999	Date and time data	8
smalldatetime	January 1, 1900, through June 6, 2079	Date and time data	4
date	January 1,0001 through December 31, 9999 A.D.	Date data	3
datetime2	January 1,0001 through December 31, 9999 A.D.	Date and time data	6 to 8
datetimeoffset	January 1,0001 through December 31, 9999 A.D.	Time zone aware date and time data	10
time	00:00:00.0000000 through 23:59:59.9999999	Time data	5
timestamp	Maximum storage size of 8 bytes	Unique number in a database that is updated every time a row that contains timestamp is inserted or updated	8

The Date and Time Data Types

Binary

The binary data types allow you to store binary data in the database. The following table lists the binary data types supported by SQL Server.

Data types	Range	Used to store	Storage (Bytes)
bit	0 or 1	Integer data with 0 or 1	Variable size
image	Maximum length of 2^31–1 (2,147,483,647) bytes	Variable length binary data to store images	Variable size
binary	Maximum length of 8000 bytes	Fixed length binary data	n (length of data)
varbinary	Maximum length of 8000 bytes	Variable length binary data	n (length of data) +2

The Binary Data Types

Others

The other data types allow you to store unique identifiers, cursors, table, XML, and spatial data in the database. The following table lists the other data types supported by SQL Server.

Data types	Range	Used to store	Storage (Bytes)
sql_variant	Maximum length of 8016 bytes	Different data types except text, ntext, image, timestamp, and sql_variant	Variable size
uniqueidentifier	Is a 16–byte GUID	Data that uniquely identifies the rows in a table across the database. A column or local variable of the uniqueidentifier data type can be initialized by: ■ Using the NEWID function. ■ Converting from a string constant in the form xxxxxxxx-xxxx-xxxx -xxxx-xxxxxxxxxxxx, where each x is a hexadecimal digit in the range 0-9 or a-f. For example, a unique identifier value is 6F9619FF-8B86-D011-B42D-00C04FC964FF	Variable size
table	Result set to be processed later	Temporary set of rows returned as a result set of a table-valued function	Variable size
xml	xml instances and xml type variables	XML values	Variable size
cursor	Stores variables or stored procedure OUTPUT parameters	Cursor reference	Variable size
geometry	Two-dimensional data	Geometrical shapes, such as points, lines, and polygon	Variable size
geography	Ellipsoidal data	Geographical data, such as the latitude and longitude coordinates that represent points, lines, and polygons on the earth's surface	Variable size
hierarchyid	Hierarchical data	Hierarchy, such as organizational structure, file system, and taxonomy of language terms	Variable size

The Other Data Types

Retrieving Specific Attributes

While retrieving data from tables, you can display one or more columns. For example, the AdventureWorks database stores the employee details, such as EmployeeID, ManagerID, Title, HireDate, and BirthDate in the Employee table. You may want to view all the columns of the Employee table or a few columns, such as EmployeeID and ManagerID. You can retrieve the required data from the database tables by using the SELECT statement.

The SELECT statement is used to access and retrieve data from a database. The syntax of the SELECT statement is:

```
SELECT [ALL | DISTINCT]
select_column_list
[INTO [new_table_name]]
FROM {table_name | view_name}
[WHERE search_condition]
```

where,

`ALL` is represented with an (*) asterisk symbol and displays all the columns of the table.

`DISTINCT` specifies that only the unique rows should appear in the result set.

`select_column_list` is the list of columns or aggregate columns for which the data is to be listed.

`INTO` creates a new table and inserts the resulting rows from the query into it.

`new_table_name` is the name of the new table to be created.

`FROM table_name` is the name of the table from which the data is to be retrieved.

`WHERE` specifies the search condition for the rows returned by the query.

`search_condition` specifies the condition to be satisfied to return the selected rows.

The SELECT statement contains some more clauses, such as GROUP BY, ROLLUP, CUBE, and ORDER BY. These clauses will be explained later in this chapter.

All the examples in this book are based on the AdventureWorks, Inc. case study given in the Appendix of the book, Querying Data Using SQL Server - Activity Book.

For example, the Employee table is stored in the HumanResources *schema* of the AdventureWorks database. To display all the details of the employees, you can use the following query:

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

The following figure shows the output of the preceding query.

	EmployeeID	NationalIDNumber	ContactID	LoginID	ManagerID	Title	BirthDate
1	1	14417807	1209	adventure-works\guy1	16	Production Technician - WC60	1972-05-15 00:
2	2	253022876	1030	adventure-works\kevin0	6	Marketing Assistant	1977-06-03 00:
3	3	509647174	1002	adventure-works\roberto0	12	Engineering Manager	1964-12-13 00:
4	4	112457891	1290	adventure-works\rob0	3	Senior Tool Designer	1965-01-23 00:
5	5	480168528	1009	adventure-works\thierry0	263	Tool Designer	1949-08-29 00:
6	6	24756624	1028	adventure-works\david0	109	Marketing Manager	1965-04-19 00:
7	7	309738752	1070	adventure-works\jolynn0	21	Production Supervisor - WC60	1946-02-16 00:
8	8	690627818	1071	adventure-works\ruth0	185	Production Technician - WC10	1946-07-06 00:
9	9	695256908	1005	adventure-works\gail0	3	Design Engineer	1942-10-29 00:
10	10	912265825	1076	adventure-works\barry0	185	Production Technician - WC10	1946-04-27 00:
11	11	998320692	1006	adventure-works\jossel0	3	Design Engineer	1949-04-11 00:
12	12	245797967	1001	adventure-works\terri0	109	Vice President of Engineering	1961-09-01 00:

The Output Derived After Using the Select Query

The result set displays the records in the same order as they are stored in the source table.

The data in the output window may vary depending on the modifications done on the database.

Schema is a namespace that acts as a container of objects in a database. A schema can be owned by any user, and its ownership is transferable. A single schema can contain objects owned by multiple database users.

If you need to retrieve specific columns from a table, you can specify the column names in the SELECT statement.

For example, to view specific details such as EmployeeID, ContactID, LoginID, and Title of the employees of AdventureWorks, you can specify the column names in the SELECT statement, as shown in the following query:

```
SELECT EmployeeID, ContactID, LoginID,
Title FROM HumanResources.Employee
```

The following figure shows the output of the preceding query.

	EmployeeID	ContactID	LoginID	Title
1	1	1209	adventure-works\guy1	Production Technician - WC60
2	2	1030	adventure-works\kevin0	Marketing Assistant
3	3	1002	adventure-works\roberto0	Engineering Manager
4	4	1290	adventure-works\rob0	Senior Tool Designer
5	5	1009	adventure-works\thierry0	Tool Designer
6	6	1028	adventure-works\david0	Marketing Manager
7	7	1070	adventure-works\jolynn0	Production Supervisor - WC60
8	8	1071	adventure-works\ruth0	Production Technician - WC10
9	9	1005	adventure-works\gail0	Design Engineer
10	10	1076	adventure-works\barry0	Production Technician - WC10
11	11	1006	adventure-works\jossef0	Design Engineer
12	12	1001	adventure-works\terri0	Vice President of Engineering
13	13	1072	adventure-works\sidney0	Production Technician - WC10

The Output Derived After Using the Select Query

In the preceding output, the result set shows the column names in the way they are present in the table definition. You can customize these column names, if required.

Customizing the Display

Sometimes, you may want to change the way data is displayed. For example, if the names of columns are not descriptive, you might need to change the default column headings by creating user-defined headings.

For example, you need to display the Department ID and Department Names from the Department table of the AdventureWorks database. You want that the column headings in the report should be different from those stored in the table, as shown in the following table.

Department Number	***Department Name***

The Report

You can write the query to accomplish the required task in any of the following ways:

1.
```
SELECT 'Department Number'=
DepartmentID, ' Department Name'=
Name FROM
HumanResources.Department
```
2.
```
SELECT DepartmentID 'Department
Number', Name ' Department Name'
FROM HumanResources.Department
```
3.
```
SELECT DepartmentID AS
'Department Number', Name AS '
Department Name' FROM
HumanResources.Department
```

The following figure shows the output of the preceding query.

	Department Number	Department Name
1	12	Document Control
2	1	Engineering
3	16	Executive
4	14	Facilities and Maintenance
5	10	Finance
6	9	Human Resources
7	11	Information Services
8	4	Marketing
9	7	Production
10	8	Production Control
11	5	Purchasing
12	13	Quality Assurance

The Output Derived After Using the Select Query

In the preceding figure, the columns are displayed with user-defined headings, but the original column names in the database table remain unchanged.

Similarly, you might need to make results more explanatory. In such a case, you can add more text to the values displayed in the columns by using *literals*. Literals are string values that are enclosed in single quotes and added to the SELECT statement. The literal values are printed in a separate column as they are written in the SELECT list. Therefore, literals are used for display purpose.

The following SQL query retrieves the employee ID and their titles from the Employee table along with a literal "Designation:":

```
SELECT EmployeeID, 'Designation:',
Title
FROM HumanResources.Employee
```

The literals are created by specifying the string within quotes and placing the same inside the SELECT query. The following figure shows the output of the preceding query.

	EmployeeID	(No column name)	Title
1	1	Designation:	Production Technician - WC60
2	2	Designation:	Marketing Assistant
3	3	Designation:	Engineering Manager
4	4	Designation:	Senior Tool Designer
5	5	Designation:	Tool Designer
6	6	Designation:	Marketing Manager
7	7	Designation:	Production Supervisor - WC60
8	8	Designation:	Production Technician - WC10
9	9	Designation:	Design Engineer
10	10	Designation:	Production Technician - WC10
11	11	Designation:	Design Engineer

The Output Derived After Using the Select Query

In the preceding figure, the result set displays a virtual column with Designation as a value in each row. This column does not physically exist in the database table.

Concatenating the Text Values in the Output

Concatenation is the operation where two strings are joined to make one string. For example, the strings, snow and ball can be concatenated to display the output, snowball.

As a database developer, you have to manage the requirements from various users, who might want to view results in different ways. You may require to display the values of multiple columns in a single column and also add a description with the column value. In such a case, you can use the *concatenation operator*. The concatenation operator is used to concatenate string expressions. It is represented by the + sign. To concatenate two strings, you can use the following query:

```
SELECT 'snow ' + 'ball'
```

The preceding query will display snowball as the output.

The following SQL query concatenates the data of the Name and GroupName columns of the Department table into a single column:

```
SELECT Name + ' department comes under ' + GroupName + ' group' AS Department
FROM HumanResources.Department
```

In the preceding query, literals such as ‘department comes under’ and ‘group’, are concatenated to increase the readability of the output. The following figure shows the output of the preceding query.

	Department
1	Engineering department comes under Research and De...
2	Tool Design department comes under Research and D...
3	Sales department comes under Sales and Marketing gr...
4	Marketing department comes under Sales and Marketin...
5	Purchasing department comes under Inventory Manage...
6	Research and Development department comes under ...
7	Production department comes under Manufacturing group
8	Production Control department comes under Manufactu...
9	Human Resources department comes under Executive ...
10	Finance department comes under Executive General an...

The Output Derived After Using the Select Query

Calculating Column Values

Sometimes, you might also need to show calculated values for the columns. For example, the Orders table stores the order details such as OrderID, ProductID, OrderDate, UnitPrice, and Units. To find the total amount of an order, you need to multiply the UnitPrice of the product with the Units. In such cases, you can apply *arithmetic operators*. Arithmetic operators are used to perform mathematical operations, such as addition, subtraction, division, and multiplication, on numeric columns or on numeric *constants*.

SQL Server supports the following arithmetic operations:

- + (for addition)
- - (for subtraction)
- / (for division)
- * (for multiplication)
- % (for modulo - the modulo arithmetic operator is used to obtain the remainder of two divisible numeric integer values)

All arithmetic operators can be used in the SELECT statement with column names and numeric constants in any combination.

When multiple arithmetic operators are used in a single query, the processing of the operation takes place according to the precedence of the arithmetic operators. The precedence level of arithmetic operators in an expression is multiplication (*), division (/), modulo (%), subtraction (-), and addition (+). You can change the precedence of the operators by using parentheses [()]. When an arithmetic expression uses the same level of precedence, the order of execution is from left to right.

For example, the EmployeePayHistory table in the HumanResources schema contains the hourly rate of the employees. The following SQL query retrieves the per day rate of the employees from the EmployeePayHistory table:

```
SELECT EmployeeID, Rate, Per_Day_Rate = 8 * Rate FROM HumanResources.EmployeePayHistory
```

In the preceding query, Rate is multiplied by 8, assuming that an employee works for 8 hours in a day. The following figure shows the output of the preceding query.

	EmployeeID	Rate	Per_Day_Rate
1	1	12.45	99.60
2	2	13.4615	107.692
3	3	43.2692	346.1536
4	4	8.62	68.96
5	4	23.72	189.76
6	4	29.8462	238.7696
7	5	25.00	200.00
8	6	24.00	192.00
9	6	28.75	230.00
10	6	37.50	300.00

The Output Derived After Using the Select Query

Retrieving Selected Rows

In a given table, a column can contain different values in different records. At times, you might need to view only those records that match a condition. For example, in a manufacturing organization, an employee wants to view a list of products from the Products table that are priced between $ 100 and $ 200. Consider another example, where a teacher wants to view the names and the scores of the students who scored more than 80%. Therefore, the query must select the names and the scores from the table with a condition added to the score column.

To retrieve selected rows based on a specific condition, you need to use the WHERE clause in the SELECT statement. Using the WHERE clause selects the rows that satisfy the condition.

The following SQL query retrieves the department details from the Department table, where the group name is Research and Development:

```
SELECT * FROM HumanResources.Department WHERE GroupName = 'Research and Development'
```

The following figure shows the output of the preceding query.

	DepartmentID	Name	GroupName	ModifiedDate
1	1	Engineering	Research and Development	1998-06-01 00:00:00.000
2	2	Tool Design	Research and Development	1998-06-01 00:00:00.000
3	6	Research and Development	Research and Development	1998-06-01 00:00:00.000

The Output Derived After Using the Select Query

In the preceding figure, the rows containing the Research and Development group name are displayed.

Using Comparison Operators to Specify Conditions

You can specify conditions in the SELECT statement to retrieve selected rows by using various comparison operators. Comparison operators test for similarity between two expressions. They allow row retrieval from a table based on the condition specified in the WHERE clause. Comparison operators cannot be used on text, ntext, or image data type expressions. The syntax for using comparison operators in the SELECT statement is:

```
SELECT column_list
FROM table_name
WHERE expression1 comparison_operator expression2
```

where,

`expression1` and `expression2` are any valid combination of a constant, a variable, a function, or a column-based expression.

In the WHERE clause, you can use a comparison operator to specify a condition. The following SQL query retrieves records from the Employee table where the vacation hour is less than 5:

```
SELECT EmployeeID, NationalIDNumber, Title, VacationHours FROM HumanResources.Employee WHERE VacationHours < 5
```

The preceding query retrieves all the rows that satisfy the specified condition by using the comparison operator, as shown in the following figure.

	EmployeeID	NationalIDNumber	Title	VacationHours
1	3	509647174	Engineering Manager	2
2	12	245797967	Vice President of Engineering	1
3	60	674171828	Production Technician - WC50	1
4	95	431859843	Production Technician - WC50	2
5	131	153288994	Production Technician - WC50	3
6	140	184188301	Chief Financial Officer	0
7	163	370581729	Production Technician - WC50	0
8	165	152085091	Production Technician - WC50	4
9	221	701156975	Production Technician - WC20	4
10	232	113393530	Production Technician - WC20	0
11	239	872923042	Production Technician - WC20	1
12	250	56772045	Production Technician - WC20	2

The Output Derived After Using the Select Query

The following table lists the comparison operators supported by SQL Server.

Operators	***Description***
=	*Equal to*
>	*Greater than*
<	*Less than*
>=	*Greater than or equal to*
<=	*Less than or equal to*
<>, !=	*Not equal to*
!<	*Not less than*
!>	*Not greater than*

The Comparison Operators Supported by SQL Server

Sometimes, you might need to view records for which one or more conditions hold true. Depending on the requirements, you can retrieve records based on the following conditions:

- Records that match one or more conditions
- Records that contain values in a given range
- Records that contain any value from a given set of values
- Records that match a pattern
- Records that contain *NULL* values
- Records to be displayed in a sequence
- Records from the top of a table
- Records without duplication of values

Retrieving Records that Match One or More Conditions

Logical operators are used in the SELECT statement to retrieve records based on one or more conditions. While querying data, you can combine more than one logical operator to apply multiple search conditions. In a SELECT statement, the conditions specified in the WHERE clause is connected by using the logical operators. The syntax for using logical operators in the SELECT statement is:

```
SELECT column_list
FROM table_name
WHERE conditional_expression1 {AND/OR} [NOT]
conditional_expression2
```

where,

`conditional_expression1` and `conditional_expression2` are any conditional expressions.

The three types of logical operators are:

- **OR**: Returns a true value when at least one condition is satisfied. For example, the following SQL query retrieves records from the Department table when the GroupName is either Manufacturing or Quality Assurance:

  ```
  SELECT * FROM HumanResources.Department WHERE GroupName = 'Manufacturing' OR GroupName = 'Quality Assurance'
  ```

- **AND**: Is used to join two conditions and returns a true value when both the conditions are satisfied. To view the details of all the employees of AdventureWorks who are married and working as a Production Technician – WC60, you can use the AND logical operator, as shown in the following query:

  ```
  SELECT * FROM HumanResources.Employee WHERE Title = 'Production Technician – WC60' AND MaritalStatus = 'M'
  ```

- **NOT**: Reverses the result of the search condition. The following SQL query retrieves records from the Department table when the GroupName is not Quality Assurance:

  ```
  SELECT * FROM HumanResources.Department WHERE NOT GroupName = 'Quality Assurance'
  ```

  The preceding query retrieves all the rows except the rows that match the condition specified after the NOT conditional expression.

Retrieving Records That Contain Values in a Given Range

Range operators retrieve data based on a range. The syntax for using range operators in the SELECT statement is:

```
SELECT column_list
FROM table_name
WHERE expression1 range_operator expression2 AND expression3
```

where,

`expression1`, `expression2`, and `expression3` are any valid combination of constants, variables, functions, or column-based expressions.

`range_operator` is any valid range operator.

Range operators are of the following types:

- **BETWEEN**: Specifies an inclusive range to search.

  The following SQL query retrieves records from the Employee table where the number of hours that the employees can avail to go on a vacation is between 20 and 50:

  ```
  SELECT EmployeeID, VacationHours FROM HumanResources.Employee WHERE VacationHours BETWEEN 20 AND 50
  ```

- **NOT BETWEEN**: Excludes the specified range from the result set.

  The following SQL query retrieves records from the Employee table where the number of hours that the employees can avail to go on a vacation is not between 40 and 50:

  ```
  SELECT EmployeeID,VacationHours FROM HumanResources.Employee WHERE VacationHours NOT BETWEEN 40 AND 50
  ```

Retrieving Records That Contain Any Value from a Given Set of Values

Sometimes, you might want to retrieve data after specifying a set of values to check whether the specified value matches any data of the table. This type of operation is performed by using the IN and NOT IN keywords. The

syntax for using the IN and NOT IN keywords in the SELECT statement is:

```
SELECT column_list
FROM table_name
WHERE expression list_operator ('value_list')
```

where,

`expression` is any valid combination of constants, variables, functions, or column-based expressions.

`list_operator` is any valid list operator, IN or NOT IN.

`value_list` is the list of values to be included or excluded in the condition.

The IN keyword selects the values that match any one of the values given in a list. The following SQL query retrieves the records of employees who are Recruiter, Stocker, or Buyer from the Employee table:

```
SELECT EmployeeID, Title, LoginID FROM HumanResources.Employee WHERE Title IN ('Recruiter', 'Stocker', 'Buyer')
```

Alternatively, the NOT IN keyword restricts the selection of values that match any one of the values in a list. The following SQL query retrieves records of employees whose designation is not Recruiter, Stocker, or Buyer:

```
SELECT EmployeeID, Title, LoginID FROM HumanResources.Employee WHERE Title NOT IN ('Recruiter', 'Stocker', 'Buyer')
```

Retrieving Records That Match a Pattern

When retrieving data, you can view selected rows that match a specific pattern. For example, you are asked to create a report that displays the names of all the products of AdventureWorks beginning with the letter P. You can do this by using the LIKE keyword. The LIKE keyword is used to search a string by using wildcards. Wildcards are special characters, such as '*' and '%'. These characters are used to match patterns.

The LIKE keyword matches the given character string with the specified pattern. The pattern can include combination of wildcard characters and regular characters. While performing a pattern match, regular characters must match the characters specified in the character string. However, wildcard characters are matched with fragments of the character string.

For example, if you want to retrieve records from the Department table where the values of Name column begin with 'Pro', you need to use the '%' wildcard character, as shown in the following query:

```
SELECT * FROM HumanResources.Department WHERE Name LIKE 'Pro%'
```

Consider another example, where you want to retrieve the rows from the Department table in which the department name is five characters long and begins with 'Sale', whereas the fifth character can be anything. For this, you need to use the '_' wildcard character, as shown in the following query:

```
SELECT * FROM HumanResources.Department WHERE Name LIKE 'Sale_'
```

The following table describes the wildcard characters that are used with the LIKE keyword in SQL server.

Wildcard	***Description***
%	*Represents any string of zero or more character(s).*
_	*Represents any single character.*
[]	*Represents any single character within the specified range.*
[^]	*Represents any single character not within the specified range.*

The Wildcard Characters Supported by SQL Server

The wildcard characters can be combined into a single expression with the LIKE keyword. The wildcard characters themselves can be searched using the LIKE keyword by putting them into square brackets ([]).

The following table describes the use of the wildcard characters with the LIKE keyword.

Expression	***Returns***
LIKE 'LO%'	*All names that begin with "LO"*
LIKE '%ion'	*All names that end with "ion"*
LIKE '%rt%'	*All names that have the letters "rt" in them*
LIKE '_rt'	*All three letter names ending with "rt"*
LIKE '[DK]%'	*All names that begin with "D" or "K"*
LIKE '[A D]ear'	*All four letter names that end with "ear" and begin with any letter from "A" through "D"*
LIKE 'D[^c]%'	*All names beginning with "D" and not having "c" as the second letter.*

The Use of the Wildcard Characters with the LIKE Keyword

The Like operator is not case-sensitive. For example, Like 'LO%' and Like 'lo%' will return the same result.

Retrieving Records That Contain NULL Values

A NULL value in a column implies that the data value for the column is not available. You might be required to find records that contain null values or records that do not contain NULL values in a particular column. In such a case, you can use the unknown_value_operator in your queries.

The syntax for using the unknown_value_operator in the SELECT statement is:

```
SELECT column_list
FROM table_name
WHERE column_name unknown_value_operator
```

where,

`unknown_value_operator` is either the keyword IS NULL or IS NOT NULL.

The following SQL query retrieves only those rows from the EmployeeDepartmentHistory table for which the value in the EndDate column is NULL:

```
SELECT EmployeeID, EndDate FROM HumanResources.EmployeeDepartmentHistory WHERE EndDate IS NULL
```

No two NULL values are equal. You cannot compare one NULL value with another.

NULL values are always the first item to be displayed in the output that is sorted in an ascending order.

At times, you might need to handle the null values in a table quiet differently. For example, the contact details of the employees are stored in the following Contact table.

EmployeeID	Residence	Office	Mobile Number
1	Null	945673561	Null
2	23456	999991111	Null
3	Null	Null	912345678
4	Null	Null	908087657

The Contact Table

The contact details contain the residential, office, and mobile number of an employee. If the employee does not have any of the contact numbers, it is substituted with a null value. Now, you want to display a result set by substituting all the null values with zero. To perform this task, you can use the ISNULL() function. The ISNULL() function replaces the null values with the specified replacement value. The syntax for using the ISNULL() function in the SELECT statement is:

```
ISNULL ( check_expression , replacement_value )
```

where,

`check_expression` is the expression to be checked for null.

`replacement_value` is the value that replaces the null values in the resultset.

For example, the following SQL query replaces the null values with zero in the query output:

```
SELECT EmployeeID, ISNULL(Residence, 0) AS Residence, ISNULL(Office, 0.00) AS Office, ISNULL(Mobile Number, 0.00) AS Mobile Number FROM Contact
```

The following figure displays the output of the preceding query.

	EmployeeID	Residence	Office	Mobile_Number
1	1	0	945673561	0
2	2	23456	999991111	0
3	3	0	0	912345678
4	4	0	0	908087676

The Output Derived After Using the ISNULL()

Consider another example, the following SQL query replaces the null values with zero in the SalesQuota column in the query output:

```
SELECT SalesPersonID, ISNULL (SalesQuota, 0.00) AS 'Sales Quota' FROM Sales.SalesPerson
```

The following figure displays the output of the preceding query.

	SalesPersonID	Sales Quota
1	268	0.00
2	275	300000.00
3	276	250000.00
4	277	250000.00
5	278	250000.00
6	279	300000.00
7	280	250000.00
8	281	250000.00
9	282	250000.00
10	283	250000.00
11	284	0.00
12	285	250000.00

The Output Derived After Using the ISNULL()

At times, you might need to retrieve the first non null value from a list of values in each row. For example, the Contact table contains three columns to store the residence, office, and mobile number details of employees. You want to generate a report that displays the first non null contact number in each row for every employee, as shown in the following table.

EmployeeID	*Contact_Number*
1	*945673561*
2	*23456*
3	*912345678*
4	*908087657*

The Report

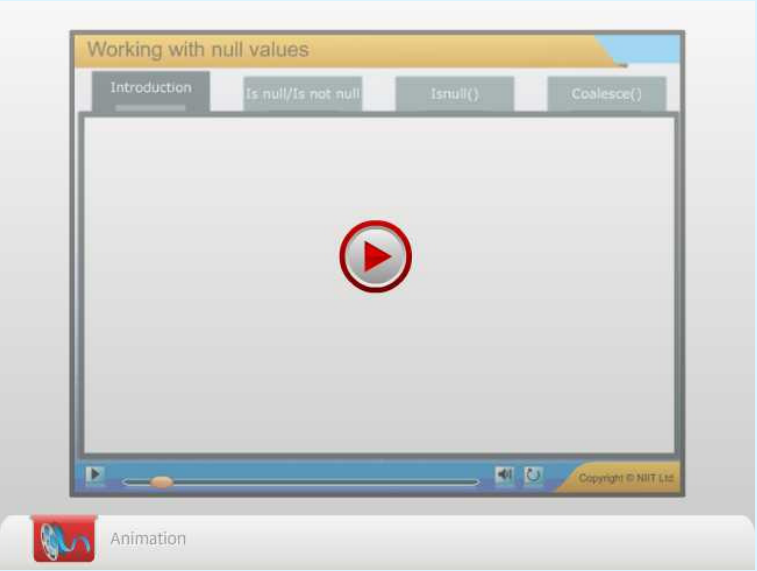
In the preceding table, the employee with ID, 1, has only the office number. Therefore, it is displayed in the Contact_Number column. The employee with ID, 2, has the residence and office numbers. Therefore, the residence number, as it is coming first in the row, is displayed in the result set.

To display such type of reports, you can use the COALESCE() function. The COALESCE() function checks the values of each column in a list and returns the first non null contact number. The null value is returned only if all the values in a list are null. The syntax for using the COALESCE() function is:

```
COALESCE ( column_name [ ,...n ] )
```

For example, you can use the following SQL query to display the very first contact number of the employees in the Contact table:

```
SELECT EmployeeID, COALESCE(Residence,
Office ,Mobile_Number)AS
Contact_Number FROM Contact
```



Retrieving Records to be Displayed in a Sequence

You can use the ORDER BY clause of the SELECT statement to display the data in a specific order.Data can be displayed in the ascending or descending order of values in a given column.

The syntax for using the ORDER BY clause in the SELECT statement is:

```
SELECT select_list
FROM table_name
[ORDER BY order_by_expression [ASC|
DESC]
[, order_by_expression [ASC|DESC]...]
```

where,

`order_by_expression` is the column name on which the sort operation is to be performed.

`ASC` specifies that the values need to be sorted in ascending order.

`DESC` specifies that the values need to be sorted in descending order.

The following SQL query retrieves the records from the Department table by setting ascending order on the Name column:

```
SELECT DepartmentID, Name FROM
HumanResources.Department ORDER BY
Name ASC
```

Optionally, you can sort the result set based on more than one column. For this, you need to specify the sequence of the sort columns in the ORDER BY clause, as shown in the following query:

```
SELECT GroupName, DepartmentID, Name
FROM HumanResources.Department ORDER
BY GroupName, DepartmentID
```

The preceding query sorts the Department table in ascending order of GroupName, and then ascending order of DepartmentID, as shown in the following figure.

	GroupName	DepartmentID	Name
1	Executive General and Administration	9	Human Resources
2	Executive General and Administration	10	Finance
3	Executive General and Administration	11	Information Services
4	Executive General and Administration	14	Facilities and Maintenance
5	Executive General and Administration	16	Executive
6	Inventory Management	5	Purchasing
7	Inventory Management	15	Shipping and Receiving
8	Manufacturing	7	Production
9	Manufacturing	8	Production Control
10	Quality Assurance	12	Document Control
11	Quality Assurance	13	Quality Assurance
12	Research and Development	1	Engineering

The Output Derived After Using the ORDER BY Clause

If you do not specify the ASC or DESC keywords with the column name in the ORDER BY clause, the records are sorted in ascending order.

The ORDER BY clause does not sort the table physically.

Retrieving Records from the Top of a Table

You can use the TOP keyword to retrieve only the first set of rows from the top of a table. This set of records can be either a number of records or a percent of rows that will be returned from a query result.

For example, you want to view the product details from the product table, where the product price is more than $ 50. There might be various records in the table, but you want to see only the top 10 records that satisfy the condition. In such a case, you can use the TOP keyword.

The syntax for using the TOP keyword in the SELECT statement is:

```
SELECT [TOP n [PERCENT] [WITH TIES]]
column_name [,column_name...]
FROM table_name
WHERE search_conditions
[ORDER BY [column_name
[,column_name...]]]
```

where,

`n` is the number of rows that you want to retrieve. If the PERCENT keyword is used, then “n” percent of the rows are returned.

`WITH TIES` specifies that result set includes all the additional rows that matches the last row returned by the TOP clause. It is used along with the ORDER BY clause.

The following query retrieves the top 10 rows of the Employee table:

```
SELECT TOP 10 * FROM
HumanResources.Employee
```

The following query retrieves the top 10% rows of the Employee table:

```
SELECT TOP 10 PERCENT * FROM
HumanResources.Employee
```

In the output of the preceding query, 29 rows will be returned where the total number of rows in the Employee table is 290.

If the SELECT statement including TOP has an ORDER BY clause, then the rows to be returned are selected after the ORDER BY clause has been applied.

For example, you want to retrieve the top three records from the Employee table where the HireDate is greater than or equal to 1/1/98 and less than or equal to 12/31/98. Further, the record should be displayed in the ascending order based on the SickLeaveHours column. To accomplish this task, you can use the following query:

```
SELECT TOP 3 * FROM
HumanResources.Employee WHERE HireDate
>= '1/1/98' AND HireDate <= '12/31/98'
ORDER BY SickLeaveHours ASC
```

Consider another example, where you want to retrieve the details of top 10 employees who have the highest sick leave hours. In addition, the result set should include all those employees whose sick leave hours matches the lowest sick leave hours included in the result set:

```
SELECT TOP 10 WITH TIES EmployeeID,
Title, SickLeaveHours FROM
HumanResources.Employee
ORDER BY SickLeaveHours DESC
```

The following figure displays the output of the preceding query.

	EmployeeID	Title	SickLeaveHours
1	4	Senior Tool Designer	80
2	72	Stocker	69
3	109	Chief Executive Officer	69
4	141	Production Technician - WC50	69
5	179	Production Technician - WC50	69
6	224	Production Technician - WC10	69
7	262	Production Technician - WC10	69
8	245	Production Technician - WC10	68
9	252	Production Technician - WC10	68
10	195	Stocker	68
11	107	Production Technician - WC50	68
12	34	Stocker	68
13	69	Production Technician - WC50	68

The Output Derived After Using the WITH TIES Clause

In the preceding figure, the number of rows returned is 13. The last three rows has the sick leave hours as 68, which is the same as the sick leave hours for the 10th row, which is the 1ast row returned by the TOP clause.

Retrieving Records from a Particular Position

At times, you might need to retrieve a specific number of records starting from a particular position in a table. For example, you may want to retrieve only five records at a time, starting from a specified position, and in a particular order.

With the ORDER BY clause, you can retrieve the records in a specific order but cannot limit the number of records returned.

With the TOP clause, you can limit the number of records returned but cannot retrieve the records from a specified position.

In such a case, you can use the OFFSET and FETCH clause to retrieve a specific number of records, starting from a particular position, in the result set. The result set should be sorted on a particular column.

The syntax for using the OFFSET-FETCH clause in the SELECT statement is:

```
SELECT select_list
FROM table_name
[ORDER BY order_by_expression [ASC|
DESC]
[, order_by_expression [ASC|DESC]...]
[ <offset_fetch> ]


<offset_fetch> ::=
    OFFSET    [    integer_constant    |
offset_row_count_expression ] [ ROW |
    ROWS]
    FETCH    [    FIRST    |    NEXT    ]
[integer_constant |
    fetch_row_count_expression ] [ ROW
| ROWS ] ONLY
```

where,

`OFFSET` specifies the number of rows to be excluded before the query execution. The value can be an integer constant or an expression.

`FETCH` specifies the number of rows to be returned in the query execution. The value can be an integer constant or an expression. FIRST and NEXT are similar. You can use either of them with the FETCH clause. Similarly, ROW and ROWS are same, use either of them.

For example, you want to retrieve the records of employees from the Employee table. But, you do not want to include the first 15 records in the result set. In such a case, you can use the OFFSET clause to exclude the first 15 records from the result set, as shown in the following query:

```
Select
EmployeeID,NationalIDNumber,ContactID
,HireDate from HumanResources.Employee
Order By EmployeeID
OFFSET 15 ROWS
```

The following figure displays the output of the preceding query.

	EmployeeID	NationalIDNumber	ContactID	HireDate
1	16	446466105	1068	1998-03-30 00:00:00.000
2	17	565090917	1074	1998-04-11 00:00:00.000
3	18	494170342	1069	1998-04-18 00:00:00.000
4	19	9659517	1075	1998-04-29 00:00:00.000
5	20	443968955	1129	1999-01-02 00:00:00.000
6	21	277173473	1231	1999-01-02 00:00:00.000
7	22	835460180	1172	1999-01-03 00:00:00.000
8	23	687685941	1173	1999-01-03 00:00:00.000
9	24	498138869	1113	1999-01-03 00:00:00.000
10	25	360868122	1054	1999-01-04 00:00:00.000

The Output Derived After Using the OFFSET Clause

You may want to retrieve the 10 records from the Employee table, excluding the first 15 records. In such a case, you can use the FETCH clause along with the OFFSET clause, as shown in the following query:

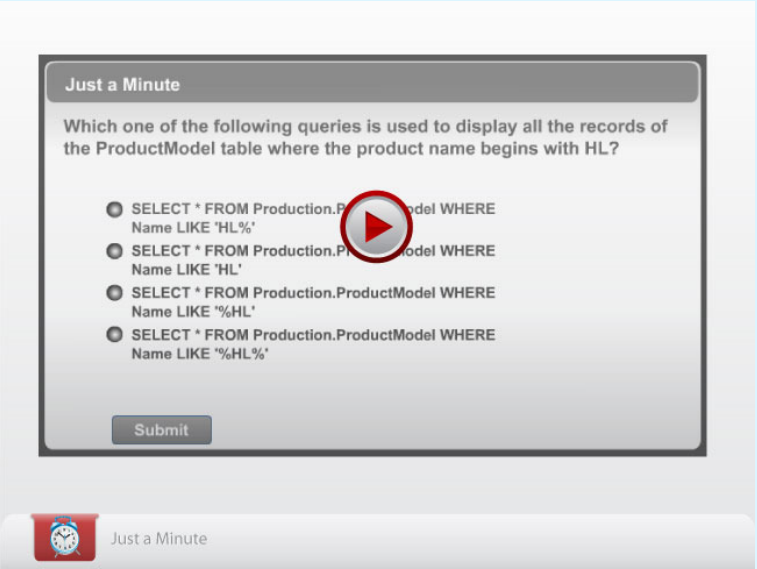
```
Select
EmployeeID,NationalIDNumber,ContactID
,HireDate
from HumanResources.Employee
Order By EmployeeID
OFFSET 15 ROWS
FETCH NEXT 10 ROWS ONLY
```

	EmployeeID	NationalIDNumber	ContactID	HireDate
1	16	446466105	1068	1998-03-30 00:00:00.000
2	17	565090917	1074	1998-04-11 00:00:00.000
3	18	494170342	1069	1998-04-18 00:00:00.000
4	19	9659517	1075	1998-04-29 00:00:00.000
5	20	443968955	1129	1999-01-02 00:00:00.000
6	21	277173473	1231	1999-01-02 00:00:00.000
7	22	835460180	1172	1999-01-03 00:00:00.000
8	23	687685941	1173	1999-01-03 00:00:00.000
9	24	498138869	1113	1999-01-03 00:00:00.000
10	25	360868122	1054	1999-01-04 00:00:00.000

The Output Derived After Using the OFFSET and FETCH CLAUSES

It is mandatory to use ORDER BY clause with the OFFSET and FETCH clause.

FETCH clause can be used only with the OFFSET clause.



Retrieving Records Without Duplication of Values

You can use the DISTINCT keyword when you need to eliminate rows with duplicate values in a column. The DISTINCT keyword eliminates the duplicate rows from the result set. The syntax of the DISTINCT keyword is:

```
SELECT [ALL|DISTINCT] column_names
FROM table_name
WHERE search_condition
```

where,

`DISTINCT` keyword specifies that only the records containing non-duplicated values in the specified column are displayed.

The following SQL query retrieves all the Titles beginning with PR from the Employee table:

```
SELECT DISTINCT Title FROM HumanResources.Employee WHERE Title LIKE 'PR%'
```

The execution of the preceding query displays a title only once.

If the DISTINCT keyword is followed by more than one column name, then it is applied to all the columns. You can specify the DISTINCT keyword only before the select list.

Task 2.1: Displaying Query Results in Different Formats

Activity 2.1: Retrieving Data

Using Functions to Customize the Result Set

While querying data from SQL Server, you can use various in-built functions provided by SQL Server to customize the result set. Customization includes changing the format of the string or date values or performing calculations on the numeric values in the result set. For example, if you need to display all the text values in upper case, you can use the upper() string function. Similarly, if you need to calculate the square of the integer values, you can use the power() mathematical function.

Depending on the utility, the in-built functions provided by SQL Server are categorized as string functions, date functions, mathematical functions, ranking functions, and system functions.

Using String Functions

You can use the string functions to manipulate the string values in the result set. For example, to display only the first eight characters of the values in a column, you can use the left() string function.

String functions are used with the char and varchar data types. SQL Server provides string functions that can be used as a part of any character expression. These functions are used for various operations on strings.

The syntax for using a function in the SELECT statement

is:

```
SELECT function_name (parameters)
```

where,

`function_name` is the name of the function.

`parameters` are the required parameters for the string function.

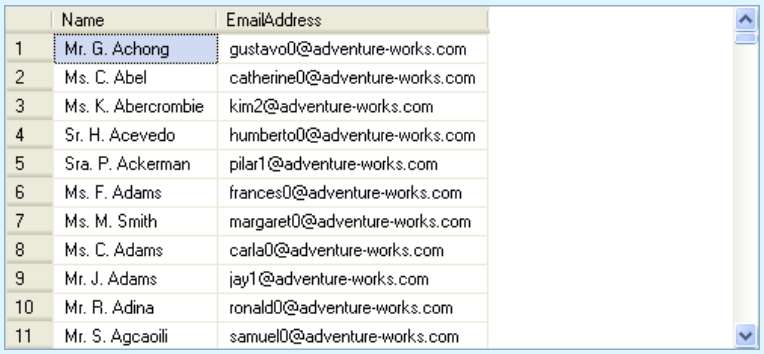
For example, you want to retrieve the Name, DepartmentID, and GroupName columns from the Department table and the data of the Name column should be displayed in uppercase with a user-defined heading, Department Name. For this, you can use the upper() string function, as shown in the following query:

```
SELECT 'Department Name'= upper(Name), DepartmentID, GroupName FROM HumanResources.Department
```

The following SQL query uses the left() string function to extract the specified characters from the left side of a string:

```
SELECT Name = Title + ' ' + left (FirstName,1) + '. ' + LastName, EmailAddress FROM Person.Contact
```

The following figure shows the output of the preceding query.



	Name	EmailAddress
1	Mr. G. Achong	gustavo0@adventure-works.com
2	Ms. C. Abel	catherine0@adventure-works.com
3	Ms. K. Abercrombie	kim2@adventure-works.com
4	Sr. H. Acevedo	humberto0@adventure-works.com
5	Sra. P. Ackerman	pilar1@adventure-works.com
6	Ms. F. Adams	frances0@adventure-works.com
7	Ms. M. Smith	margaret0@adventure-works.com
8	Ms. C. Adams	carla0@adventure-works.com
9	Mr. J. Adams	jay1@adventure-works.com
10	Mr. R. Adina	ronald0@adventure-works.com
11	Mr. S. Agcaoili	samuel0@adventure-works.com

The Output Derived After Using the Query

The following table lists the string functions provided by SQL Server.

Function name	***Example***	***Description***
Ascii (character_expression)	`SELECT ascii ('ABC')`	*Returns 65, the ASCII code of the leftmost character 'A'.*
Char (integer _expression)	`SELECT char (65)`	*Returns 'A', the character equivalent of the ASCII code value.*
Charindex (‘pattern’, expression)	`SELECT charindex ('E','HELLO')`	*Returns 2, the starting position of the specified pattern in the expression. The pattern should be a string literal without using any wildcard characters.*
Difference (character_expression1, character_expression2)	`SELECT difference ('HELLO', 'hell')`	*Returns 4, the DIFFERENCE function compares two strings and evaluates the similarity between them, returning a value from 0 through 4. The value 4 is the best match.*
Left (character_expression, integer_expression)	`SELECT left ('RICHARD',4)`	*Returns 'RICH', which is a part of the character string equal in size to the integer_expression characters from the left.*
Len (character_expression)	`SELECT len ('RICHARD')`	*Returns 7, the number of characters in the character_expression.*
Lower (character_expression)	`SELECT lower ('RICHARD')`	*Returns 'richard', after converting character_expression to lower case.*
Ltrim (character_expression)	`SELECT ltrim (' RICHARD')`	*Returns 'RICHARD' without leading spaces. It removes leading blanks from the character expression.*
Replace (string_expression, string_pattern, string_replacement)	`SELECT REPLACE ('University','ity','e');`	*Returns ‘Universe’ after replacing ‘ity’ with ‘e’.*
Replicate (string_expression ,integer_expression)	`SELECT Replicate ('University',2)`	*Returns ‘UniversityUniversity’. It repeats the string value for a specified number of times.*
Patindex (‘% pattern%’, expression)	`Select patindex ('%BOX %','ACTIONBOX')`	*Returns 7, the starting position of the first occurrence of the pattern in the*

		specified expression, or zeros if the pattern is not found. It uses wildcard characters to specify the pattern to be searched.
Reverse (character_expression)	`SELECT reverse ('ACTION')`	*Returns 'NOITCA', the reverse of the character_expression.*
Right (character_expression, integer_expression)	`SELECT right ('RICHARD',4)`	*Returns 'HARD', a part of the character string, after extracting from the right the number of characters specified in the integer_expression.*
Rtrim (character_expression)	`SELECT rtrim ('RICHARD ')`	*Returns 'RICHARD', after removing any trailing blanks from the character expression.*
Space (integer_expression)	`SELECT 'RICHARD'+ space (2) +'HILL'`	*Returns 'RICHARD HILL'. Two spaces are inserted between the first and second word.*
Str (float_expression, [length, [decimal]])	`SELECT str (123.45,6,2)`	*Returns '123.45'. It converts numeric data to character data where the length is the total length, including the decimal point, the sign, the digits, and the spaces and the decimal is the number of places to the right of the decimal point.*
Stuff (character_expression1, start, length, character_expression2)	`SELECT stuff ('Weather', 2,2, 'i')`	*Returns 'Wither'. It deletes the number of characters as specified in the character_expression1 from the start and then inserts char_expression2 into character_expression1 at the start position.*
Substring (expression, start, length)	`SELECT substring ('Weather', 2,2)`	*Returns 'ea', which is part of the character string. It returns the part of the source character string from the start position of the expression.*
Upper (character_expression)	`SELECT upper ('Richard')`	*Returns 'RICHARD'. It converts lower case characters to upper case.*
Concat (string_value1, string_value2 [, string_valueN])	`SELECT concat ('Richard', 'Hill')`	`Returns 'RichardHill'. It concatenates two or more strings into a single string.`

The String Functions Provided by SQL Server

Using Conversion Functions

You can use the conversion functions to convert data from one type to another. For example, you want to convert a string value into a numeric format. You can use the parse() function to convert string values to numeric or date time format, as shown in the following query:

```
SELECT parse('219' as decimal)
```

The following table lists the conversion functions provided by SQL Server.

Function	Parameter	Example	Remarks
Parse	(string_value AS data_type)	SELECT parse('2009-01-09' as Datetime2)	Returns 2009-01-09 00:00:00.0000000, after converting date to the datetime2 format. This function generates an error if the conversion from one data type to another is not possible.
Try_Parse	(string_value AS data_type)	SELECT try_parse ('2010-1-9' as int)	Returns Null. This function returns NULL if the conversion is not possible but will not throw an exception.
convert	(datatype [(length)], expression [, style])	SELECT convert (char(10), HireDate,2) FROM Human Resources.Employee	Converts the hire date from the date data type to the character data type and then displays it in the yy.mm.dd format. The third argument, style, specifies the method of representing the date when converting the date data type into the character data type.
TRY_CONVERT()	data_type [(length)], expression [, style])	SELECT Title, try_convert (int,HireDate,2) AS 'Hire Date' FROM Human Resources.Employee	Returns Null. This function is similar to the convert() function but it returns null if the conversion is not possible.

The Conversion Functions Provided by SQL Server

The following table lists the style values for displaying datetime expressions in different formats.

Style value	***Format***
1	*mm/dd/yyyy*
2	*yy.mm.dd*
3	*dd/mm/yyyy*
4	*dd.mm.yy*
5	*dd-mm-yy*

The Style Values

The TRY_CONVERT() may give an error at the time of execution that it is not a recognized built-in function name. Perform the following steps to resolve this problem: Right-click the AdventureWorks database in the Object-Explorer window. Select ***Properties****. The* ***Database Properties-AdventureWorks*** *dialog box is displayed. Select* ***Options*** *in the* ***Select a page*** *pane of the dialog box. Select* ***SQL Server 2012(110)*** *in the* ***Compatibility level*** *drop-down list. Click* ***OK*** *to close the dialog box.*

Using Date Functions

You can use the date functions of SQL Server to manipulate date and time values. You can either perform arithmetic operations on date values or parse the date values. Date parsing includes extracting components, such as the day, the month, and the year from a date value.

You can also retrieve the system date and use the value in the date manipulation operations. To retrieve the current system date, you can use the getdate() function. The following query displays the current date:

```
SELECT getdate()
```

The datediff() function is used to calculate the difference between two dates. For example, the following SQL query uses the datediff() function to calculate the age of the employees:

```
SELECT datediff (yy, BirthDate, getdate()) AS 'Age'
FROM HumanResources.Employee
```

The preceding query calculates the difference between the current date and the date of birth of employees, whereas, the date of birth of employees is stored in the BirthDate column of the Employee table in the AdventureWorks database.

The following table lists the date functions provided by SQL Server.

Function name	***Example***	***Description***
Dateadd (date part, number, date)	`SELECT dateadd(mm, 3,'2009-01-01')`	*Returns 2009-04-01, adds 3 months to the date.*
Datediff (date part, date1, date2)	`SELECT datediff (year, convert (datetime,'2005-06-06'), convert (datetime, '2009-01-01' ))`	*Returns 4, calculates the number of date parts between two dates.*
Datename (date part, date)	`SELECT datename (month, convert (datetime,'2005-06-06'))`	*Returns June, date part from the listed date as a character value.*
Datepart (date part, date)	`SELECT datepart(mm, '2009-09-01' )`	*Returns 9, the month, from the listed date as an integer.*
Getdate()	`SELECT getdate()`	*Returns the current date and time.*
Day (date)	`SELECT day ('2009 01-05')`	*Returns 5, an integer, which represents the day.*
Getutcdate()	`SELECT getutcdate()`	*Returns the current date of the system in Universal Time Coordinate (UTC) time. UTC time is also known as the Greenwich Mean Time (GMT).*
Month (date)	`SELECT month ('2009 01 05')`	*Returns 1, an integer, which represents the month.*
Year (date)	`SELECT year ('2009-01-05 ')`	*Returns 2009, an integer, which represents the year.*
Datefromparts (year, date, month)	`Select datefromparts (2009,01,05)`	*Returns 2009-01-05, which is the value for the specified year, month, and day.*
Datetimefromparts (year, month, day, hour, minute, seconds,	`SELECT DATETIMEFROMPARTS (2009, 01, 09,`	*Returns 2009-01-09 12:25:45.050, which is the full*

milliseconds)	`12,25, 45, 50 )`	*datetime value.*
Eomonth (start_date [, month_to_add])	`SELECT EOMONTH ('2009-01-09', 2)`	*Returns 2009-03-31.It displays the last date of the month specified in start_date. The month_to_add arguments specifies the number of months to add to start_date.*

The Date Functions Provided by SQL Server

To parse the date values, you can use the datepart() function in conjunction with the date functions. For example, the datepart() function retrieves the year when an employee was hired, along with the employee title, from the Employee table, as shown in the following query:

```
SELECT Title, datepart (yy, HireDate)
AS 'Year of Joining'
FROM HumanResources.Employee
```

SQL Server provides the abbreviations and values of the of the first parameter in the datepart() function, as shown in the following table.

Date part	*Abbreviation*	*Values*
Year	*yy, yyyy*	*1753-9999*
Quarter	*qq, q*	*1-4*
Month	*mm, m*	*1-12*
day of year	*dy, y*	*1-366*
Day	*dd, d*	*1-31*
Week	*wk, ww*	*0-51*
Weekday	*Dw*	*1-7(1 is Sunday)*
Hour	*Hh*	*(0-23)*
Minute	*mi, n*	*(0-59)*
Second	*ss, s*	*0-59*
Millisecond	*Ms*	*0-999*

The Abbreviations Used to Extract Different Parts of a Date

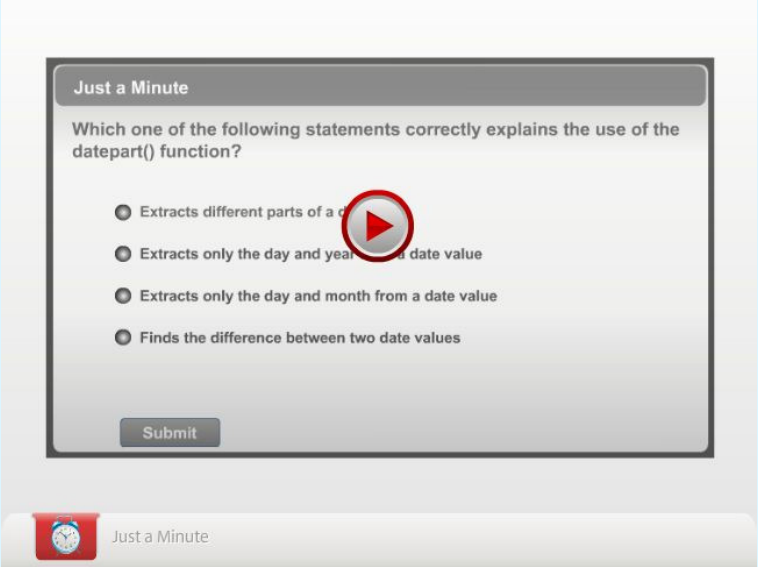
The following SQL query uses datename() and datepart() functions to retrieve the month name and year from a given date:

```
SELECT EmployeeID,datename(mm,
hiredate)+ ', ' + convert
(varchar,datepart(yyyy, hiredate)) as
'Joining'
FROM HumanResources.Employee
```

The following figure shows the output of the preceding query.

	EmployeeID	Joining
1	1	July, 1996
2	2	February, 1997
3	3	December, 1997
4	4	January, 1998
5	5	January, 1998
6	6	January, 1998
7	7	January, 1998
8	8	February, 1998
9	9	February, 1998
10	10	February, 1998
11	11	February, 1998

The Output Derived After Using the Query



Using Mathematical Functions

You can use mathematical functions to manipulate the numeric values in a result set. You can perform various numeric and arithmetic operations on the numeric values. For example, you can calculate the absolute value of a number or you can calculate the square or square root of a value.

The following table lists the mathematical functions provided by SQL Server.

Function name	*Example*	*Description*
Abs (numeric_expression)	`SELECT abs (-87)`	*Returns 87, an absolute value.*
Ceiling (numeric_expression)	`SELECT ceiling (14.45)`	*Returns 15, the smallest integer greater than or equal to the specified value.*
Exp (float_expression)	`SELECT exp (4.5)`	*Returns 90.017131300521 8, the exponential value of the*

		specified value.
Floor (numeric_expression)	`SELECT floor (14.45)`	*Returns14, the largest integer less than or equal to the specified value.*
Log (float_expression)	`SELECT log (5.4)`	*Returns 1.6863989535702 3, the natural logarithm of the specified value.*
log10 (float_expression)	`SELECT log10 (5.4)`	*Returns the base-10 logarithm of the specified value.*
Pi ()	`SELECT pi()`	*Returns the constant value of 3.1415926535897 93.*
Power (numeric_expression, y)	`SELECT power (4,3)`	*Returns 64, which is 4 to the power of 3.*
Radians (numeric_expression)	`SELECT radians(180)`	*Returns 3, converts from degrees to radians.*
Rand ([seed])	`SELECT rand ()`	*Returns a random float number between 0 and 1.*
Round (numeric_expression, length)	`SELECT round (15.789, 2)`	*Returns 15.790, a numeric expression rounded off to the length specified as an integer expression.*
Sign (numeric_expression)	`SELECT sign (-15)`	*Returns positive, negative, or zero. Here, it returns -1.*
Sqrt (float_expression)	`SELECT sqrt (64)`	*Returns 8, the square root of the specified value.*

The Mathematical Functions Provided by SQL Server

For example, to calculate the round off value of any number, you can use the round() mathematical function. The round() mathematical function calculates and returns the numeric value based on the input values provided as an argument.

The syntax of the round() function is:

```
round(numeric_expression, length)
```

where,

`numeric_expression` is the numeric expression to be rounded off.

`length` is the precision to which the expression is to be rounded off.

The following SQL query retrieves the EmployeeID and Rate for the specified employee ID from the EmployeePayHistory table:

```
SELECT EmployeeID, 'Hourly Pay Rate' =
round(Rate,2)
FROM HumanResources.EmployeePayHistory
```

The following figure shows the output of the preceding query.

	EmployeeID	Hourly Pay Rate
1	1	12.45
2	2	13.46
3	3	43.27
4	4	8.62
5	4	23.72
6	4	29.85
7	5	25.00
8	6	24.00
9	6	28.75
10	6	37.50
11	7	25.00

The Output Derived After Executing the Query

In the preceding figure, the value of the Hourly Pay Rate column is rounded off to two decimal places.

While using the round() function, if the length is positive, then the expression is rounded to the right of the decimal point. If the length is negative, then the expression is rounded to the left of the decimal point.

The following table lists the usage of the round() function provided by SQL Server.

Function	***Output***
round (1234.567,2)	*1234.570*
round (1234.567,1)	*1234.600*
round (1234.567,0)	*1235.000*
round (1234.567,-1)	*1230.000*
round (1234.567,-2)	*1200.000*
round (1234.567,-3)	*1000.000*

The Usage of the round() Function

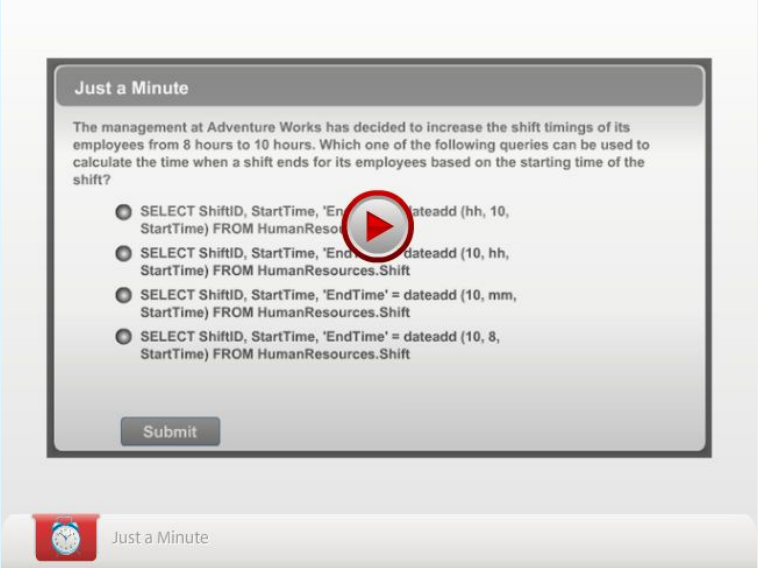
Using Logical Functions

You can use the logical functions to perform logical operations on a result set. For example, to return a value from a list of values, you can use the Choose() function.

The logical functions return a Boolean value as output. The following table lists the logical functions provided by SQL Server.

Function name	Parameters	Examples	Description
Choose	(index, val_1, val_2 [, val_n])	SELECT CHOOSE (1, 'Trigger', 'Procedure', 'Index')	Returns Trigger, which is the value at index position, 1.It returns the item at the specified index from a list of values.
IIF	(boolean_expression, true_value, false_value)	Select iif(5>7,'True','False')	Returns False. It returns true_value if boolean_expression evaluates to true. It returns false_value if boolean_expression evaluates to false .

The Logical Functions



Using Ranking Functions

You can use ranking functions to generate sequential numbers for each row or to give a rank based on specific criteria. For example, in a manufacturing organization, the management wants to rank the employees based on their salary. To rank the employees, you can use the rank() function.

Consider another example, where a teacher wants to see the names and scores of all students according to their ranks. Ranking functions return a ranking value for each row. However, based on the criteria, more than one row can get the same rank. You can use the following functions to rank the records:

- row_number()
- rank()
- dense_ rank()
- ntile()

All these functions make use of the OVER clause. This clause determines the ascending or descending sequence in which rows are assigned a rank. You can use the PARTITION BY clause with the OVER clause to partition the rows on which the ranking is performed. The PARTITION BY clause divides the result set returned into partitions based on the specified condition, and then the RANK function is applied to each partition.

The row_number() Function

The row_number() function returns the sequential numbers, starting at 1, for the rows in a result set based on a column.

For example, the following SQL query displays the sequential number on a column by using the row_number () function:

```
SELECT EmployeeID, Rate, row_number() OVER(ORDER BY Rate desc) AS RANK FROM HumanResources.EmployeePayHistory
```

The following figure displays the output of the preceding query.

	EmployeeID	Rate	RANK
1	109	125.50	1
2	148	84.1346	2
3	273	72.1154	3
4	12	63.4615	4
5	140	60.0962	5
6	158	50.4808	6
7	42	50.4808	7
8	140	48.5577	8
9	268	48.101	9
10	284	48.101	10
11	288	48.101	11

The Output Derived After Using the row_number() Function

The EmployeeID and Rate column are retrieved from the EmployeePayHistory table, where the Rate column is ranked by using the row_number() function. The ORDER BY keyword in the OVER clause specifies that the result set will appear in the descending order of the Rate column.

The rank() Function

The rank() function returns the rank of each row in a result set based on the specified criteria. For example, you want to rank the products based on the sales made during a year. For this, you can use the rank() function. This function will consider the ORDER BY clause and the record with maximum value will get the highest rank if the ORDER BY clause is asc.

For example, you want to create the report of all the employees with their salary rates. The salary rates should also contain the rank of the salary. The highest ranked employee should be given the rank as 1. In addition, if two employees have the same salary rate, they should be given the same rank.

To perform this task, you need to use the following query:

```
SELECT EmployeeID, Rate, rank() OVER (ORDER BY Rate desc) AS Rank FROM HumanResources.EmployeePayHistory
```

The following figure displays the output of the preceding query.

	EmployeeID	Rate	Rank
1	109	125.50	1
2	148	84.1346	2
3	273	72.1154	3
4	12	63.4615	4
5	140	60.0962	5
6	158	50.4808	6
7	42	50.4808	6
8	140	48.5577	8
9	268	48.101	9
10	284	48.101	9
11	288	48.101	9

The Output Derived After Using the rank() Function

In the preceding output, the salary rates for the employee ID 158 and 42 are same. Therefore, both of them have been ranked 6 but the rank for the employee ID 140 is 8, not 7. To provide the consecutive ranks, you need to use the dense_rank() function.

The dense_rank() Function

The dense_rank() function is used where consecutive ranking values need to be given based on the specified criteria. It performs the same ranking task as the rank() function, but provides consecutive ranking values to an output. For example, you want to rank the products based on the sales done for that product during a year. If two products A and B have same sale values, both will be assigned a common rank. The next product in the order of sales values will be assigned the next rank value.

If in the preceding example of the rank() function, you need to give the same rank to the employees with the same salary rate and the consecutive rank to the next one, you need to use the following query:

```
SELECT EmployeeID, Rate, dense_rank()
OVER(ORDER BY Rate desc) AS Rank FROM
HumanResources.EmployeePayHistory
```

The following figure displays the output of the preceding query.

	EmployeeID	Rate	Rank
1	109	125.50	1
2	148	84.1346	2
3	273	72.1154	3
4	12	63.4615	4
5	140	60.0962	5
6	158	50.4808	6
7	42	50.4808	6
8	140	48.5577	7
9	268	48.101	8
10	284	48.101	8
11	288	48.101	8

The Output Derived After Using the dense_rank()

Consider another example, where you want to rank the sales person‘s data according to the value of the sale amount. However, you want that the ranking should be performed for each sales territory. You can perform this task by using the PARTITION BY clause on the sales territory column, as shown in the following query:

```
SELECT SalesPersonID, TerritoryID,
SalesYTD,
dense_rank() OVER(PARTITION BY
TerritoryID ORDER BY SalesYTD desc)
as Rank FROM Sales.SalesPerson where
TerritoryID IS NOT NULL
```

The following figure displays the output of the preceding query.

	SalesPersonID	TerritoryID	SalesYTD	Rank
1	283	1	3587378.4257	1
2	287	1	1931620.1835	2
3	280	1	0.00	3
4	275	2	4557045.0459	1
5	277	3	3857163.6332	1
6	276	4	5200475.2313	1
7	281	4	3018725.4858	2
8	279	5	2811012.7151	1
9	282	6	3189356.2465	1
10	278	6	1764938.9859	2
11	286	7	3827950.238	1
12	289	8	2241204.0424	1
13	290	9	1758385.926	1
14	285	10	5015682.3752	1

The Output Derived After Using the PARTITION BY Clause

In the preceding figure, first the sales territory are partitioned on the basis of TerritoryID, and then ranking is applied on the SalesYTD column for each territory.

While using the dense_rank() function, the ranks assigned to the rows are consecutive and depend on the distinct values of the column on which ranking is performed. However, if you need to divide the rows into the specific number of rank groups, you can use the ntile() function.

The ntile() Function

The ntile() function is used to divide the result set into a specific number of groups. This function accepts a positive integer to distribute the rows into the number of groups. The distributed groups are numbered, starting from 1. For each row retrieved from the database, the ntile () function returns the number of group to which the row belongs.

For example, you want to retrieve the details of those employees who are hired after April 2001. The employees need to be grouped into four groups, and then ranked based on their birth date. You can perform this task by using the ntile() function with 4 as parameter, as shown in the following query:

```
SELECT EmployeeID, BirthDate,
HireDate, ntile(4) OVER(ORDER BY
BirthDate) AS Rank
FROM HumanResources.Employee
WHERE datepart(mm, HireDate) >= 04 AND
datepart(yy, HireDate) >= 2001
```

The following figure displays the output of the preceding query.

	EmployeeID	BirthDate	HireDate	Rank
1	284	1947-10-22 00:00:00.000	2002-05-18 00:00:00.000	1
2	277	1952-09-29 00:00:00.000	2001-07-01 00:00:00.000	1
3	282	1954-01-11 00:00:00.000	2001-07-01 00:00:00.000	1
4	281	1958-04-10 00:00:00.000	2001-07-01 00:00:00.000	1
5	285	1958-04-18 00:00:00.000	2002-07-01 00:00:00.000	2
6	275	1959-01-26 00:00:00.000	2001-07-01 00:00:00.000	2
7	290	1961-04-18 00:00:00.000	2003-07-01 00:00:00.000	2
8	279	1964-02-19 00:00:00.000	2001-07-01 00:00:00.000	2
9	283	1964-03-14 00:00:00.000	2001-07-01 00:00:00.000	3
10	280	1965-01-06 00:00:00.000	2001-07-01 00:00:00.000	3
11	288	1965-02-11 00:00:00.000	2003-04-15 00:00:00.000	3
12	278	1965-03-07 00:00:00.000	2001-07-01 00:00:00.000	3
13	289	1965-08-09 00:00:00.000	2003-07-01 00:00:00.000	4
14	286	1965-10-31 00:00:00.000	2002-07-01 00:00:00.000	4
15	287	1968-02-06 00:00:00.000	2002-11-01 00:00:00.000	4
16	276	1970-03-30 00:00:00.000	2001-07-01 00:00:00.000	4

The Output Derived After Using the ntile() Function

In the preceding figure, the records in the Employee table are divided into four groups after arranging the data in the BirthDate column in ascending order. Each group contains four rows and each row in a group is given the same ranking.

The number of rows in a group depends on the total number of rows in the table. If the number of rows in a table is not divisible by the number specified in the ntile() function, the groups will be of two sizes, one larger or smaller and the rest equal. Larger groups come before smaller groups in the order specified by the OVER clause.

For example, if the number of rows in the table is 16 and the result set is partitioned into 5 groups, the first group will contain 4 records. However, the rest of the groups will contain 3 records. Continuing with the preceding example, change the number of groups to 5 in the ntile() function, as shown in the following query:

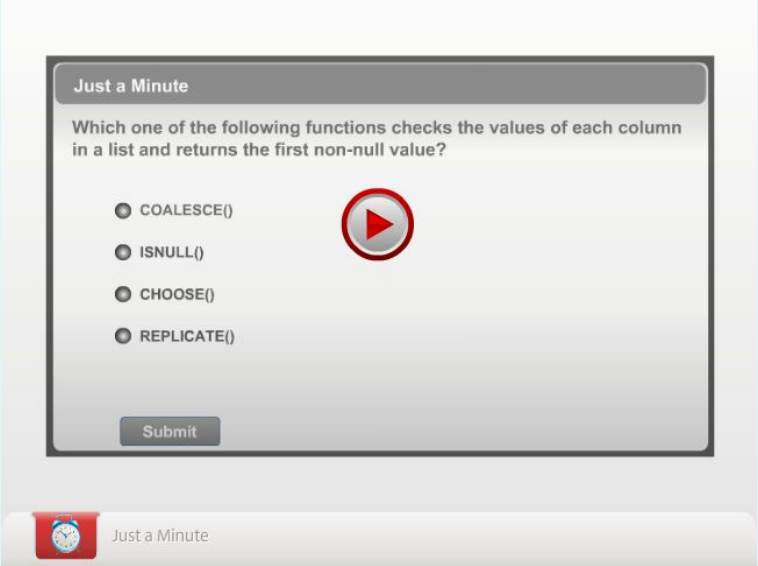
```
SELECT EmployeeID, BirthDate,
HireDate, ntile(5) OVER(ORDER BY
BirthDate) AS Rank
FROM HumanResources.Employee
WHERE datepart(mm, HireDate) >= 04 and
datepart(yy, HireDate) >= 2001
```

The following figure displays the output of the preceding query.

	EmployeeID	BirthDate	HireDate	Rank
1	284	1947-10-22 00:00:00.000	2002-05-18 00:00:00.000	1
2	277	1952-09-29 00:00:00.000	2001-07-01 00:00:00.000	1
3	282	1954-01-11 00:00:00.000	2001-07-01 00:00:00.000	1
4	281	1958-04-10 00:00:00.000	2001-07-01 00:00:00.000	1
5	285	1958-04-18 00:00:00.000	2002-07-01 00:00:00.000	2
6	275	1959-01-26 00:00:00.000	2001-07-01 00:00:00.000	2
7	290	1961-04-18 00:00:00.000	2003-07-01 00:00:00.000	2
8	279	1964-02-19 00:00:00.000	2001-07-01 00:00:00.000	3
9	283	1964-03-14 00:00:00.000	2001-07-01 00:00:00.000	3
10	280	1965-01-06 00:00:00.000	2001-07-01 00:00:00.000	3
11	288	1965-02-11 00:00:00.000	2003-04-15 00:00:00.000	4
12	278	1965-03-07 00:00:00.000	2001-07-01 00:00:00.000	4
13	289	1965-08-09 00:00:00.000	2003-07-01 00:00:00.000	4
14	286	1965-10-31 00:00:00.000	2002-07-01 00:00:00.000	5
15	287	1968-02-06 00:00:00.000	2002-11-01 00:00:00.000	5
16	276	1970-03-30 00:00:00.000	2001-07-01 00:00:00.000	5

The Output Derived After Using the ntile() Clause

As shown in the preceding figure, the first group contains four records. However, the rest of the records contain three records.



Using Analytical Functions

At times, you may want to perform analytical comparison of the data stored in a table and compute some aggregate result value like average or percentage. For example, consider the following Emp_SalesData table.

YEAR	MONTH	PRODUCTID	AMOUNT
2012	2	P1	20000
2012	2	P2	12000
2012	3	P1	15000
2012	3	P2	8000
2012	4	P1	12000
2012	4	P2	14000
2012	5	P3	14000

The Emp_SalesData Table

The preceding table stores the month wise sales details for the products, P1, P2, and P3. You want to display the sales amount for each month as well as for the following and preceding months to analyze the increase or decrease in the sales. SQL Server provides the analytical functions to perform such analysis. Unlike aggregate functions, analytical functions can return multiple rows for each group specified. Some of the analytical functions are:

- lead()
- lag()
- first_value()
- last_value()

lead()

The lead() function returns the values from the subsequent rows in the same result set. This analytical function is used to compare values in the current row with the values in the subsequent rows. The syntax of the lead() function is:

```
lead ( scalar_expression [ ,offset ] ,
[ default ] )
  OVER   (   [   partition_by_clause   ]
order_by_clause )
```

where,

`scalar_expression` is the value to be returned based on the specified offset.

`offset` is the number of rows to be forwarded from the current row. If the offset is not specified, it will take the default value, 1.

`Default` is the value to return when scalar_expression at offset is NULL. If it is not specified, it will take the default value, NULL.

`OVER ( [ partition_by_clause ] order_by_clause)` specifies the order of the data before the function is applied.

For example, you want to prepare a sales report that

displays the comparison of sales happened in a particular month with the next month sales. For this, you need to display the total sales amount for each month along with the total sales amount for the following month. To perform this task, you can use the following query:

```
SELECT
MONTH, SUM(amount) AS TotalSale,lead
(sum(amount), 1) OVER (ORDER BY month)
AS Next_Month_Sales
FROM EMP_SALESDATA
GROUP BY MONTH
ORDER BY MONTH;
```

The following figure displays the output of the preceding query.

	MONTH	TotalSale	Next_Month_Sales
1	2	32000	23000
2	3	23000	26000
3	4	26000	14000
4	5	14000	NULL

The Output Derived by Using the lead() Function

lag()

The lag() function returns the values from the preceding rows in the same result set. This analytical function is used to compare the values in the current row with the values in the previous rows. The syntax of the lag() function is:

```
lag (scalar_expression [,offset]
[,default])
   OVER ( [ partition_by_clause ]
order_by_clause )
```

For example, you want to prepare a sales report that displays the comparison of sales happened in the current month with the previous month sales. For this, you need to display the total sales amount for each month along with the total sales amount for the previous month. To perform this task, you can use the following query:

```
SELECT
MONTH, SUM(amount) AS TotalSale,lag
(sum(amount), 1) OVER (ORDER BY month)
AS Previous_Month_Sales
FROM EMP_SALESDATA
GROUP BY MONTH
ORDER BY MONTH;
```

The following figure displays the output of the preceding query.

	MONTH	TotalSale	Previous_Month_Sales
1	2	32000	NULL
2	3	23000	32000
3	4	26000	23000
4	5	14000	26000

The Output Derived by Using the lag() Function

first_value()

The first_value() function returns the first value in an ordered set of records. For example, you may want to retrieve the first month of the year when the sales amount for each product was collected for the first time. To perform this task, you can use the first_value() function. The syntax of this function is:

```
first_value ( [scalar_expression )
    OVER ( [ partition_by_clause ]
order_by_clause [
rows_range_clause ] )
```

where,

`rows_range_clause` limits the rows within the partition by specifying the start and end points.

For example, consider the following query:

```
SELECT DISTINCT ProductID, first_value
(Month) OVER (PARTITION BY ProductID
ORDER BY Month ) AS
'FirstMonthOfCollection' FROM
EMP_SALESDATA
```

The preceding query retrieves the first month of the year when the sales amount for each product was collected for the very first time, as shown in the following figure.

	ProductID	Amount	FirstMonthOfCollection
1	P1	20000	2
2	P1	15000	2
3	P1	12000	2
4	P2	12000	2
5	P2	8000	2
6	P2	14000	2
7	P3	14000	5

The Output Derived by Using the first_value() Function

last_value()

The last_value() function returns the last value in an ordered set of records. For example, you may want to retrieve the most recent month of the year when the sales amount for each product was collected. To perform this task, you can use the last_value() function. The syntax of using this function is:

```
last_value ( [scalar_expression ) OVER
```

```
( [ partition_by_clause ] order_by_clause
[ rows_range_clause ] )
```

For example, consider the following query:

```
SELECT DISTINCT ProductID, last_value (Month) OVER (PARTITION BY ProductID ORDER BY Month ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING) AS 'RecentMonthOfCollection' FROM EMP_SALESDATA
```

The preceding query retrieves the most recent month of the year when the sales amount for each product was collected, as shown in the following figure.

	ProductID	Amount	RecentMonthOfCollection
1	P1	12000	4
2	P1	15000	4
3	P1	20000	4
4	P2	14000	4
5	P2	8000	4
6	P2	12000	4
7	P3	14000	5

The Output Derived by Using the last_value() Function

Using System Functions

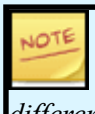
The system functions are used to query the system tables. System tables are a set of tables that are used by SQL Server to store information about users, databases, tables, and security. The system functions are used to access the SQL Server databases or user-related information. For example, to view the host ID of the terminal on which you are logged on, you can use the following query:

```
SELECT host_id() AS 'HostID'
```

The preceding query displays the output as shown in the following figure.

	HostID
1	4988

The Output Derived After Executing the host_id()



The system functions may display different output on different computers.

The following table lists the system functions provided by SQL Server.

Functions	***Description***	***Example***
host_id()	*Returns the current host process ID number of a client process.*	`SELECT host_id() AS 'HostID'`
host_name ()	*Returns the current host computer name of a client process.*	`SELECT host_name() AS 'HostName'`
suser_sid ([‘login_name’])	*Returns the Security Identification (SID) number corresponding to the log on name of the user.*	`SELECT suser_sid ('sa') AS SID`
suser_id ([‘login_name’])	*Returns the log on Identification (ID) number corresponding to the log on name of the user.*	`SELECT suser_id ('sa') AS USERID`
suser_sname ([server_user_id])	*Returns the log on name of the user corresponding to the security identification number.*	`SELECT suser_sname (0x01) AS SUSER`
user_id ([‘name_in_db’])	*Returns the database identification number corresponding to the user name.*	`SELECT user_id ('Robert') AS USERID`
user_name ([user_id])	*Returns the user name corresponding to the database identification number.*	`SELECT user_name (13)AS UserName`
db_id (['db_name'])	*Returns the database identification number of the database.*	`SELECT db_id ('AdventureWorks') AS DatabaseID`
db_name ([db_id])	*Returns the database name.*	`SELECT db_name(17) AS DatabaseName`
object_id (‘objname’)	*Returns the database object ID number.*	`SELECT object_id ('AdventureWorks.HumanResources.Employee') AS ObjectID`
object_name	*Returns the*	`SELECT`

('obj_id')	*database object name.*	`object_name (901578250) AS ObjectName`
cast (expression AS data_type [(length)])	*Converts an expression of one data type to the specified data type.*	`SELECT cast (NationalIDNumber AS CHAR(20)) AS IDNUMBER FROM HumanResources.Employee`
convert (data_type [(length)], expression [, style])	*Converts an expression of one data type to the specified data type.*	`SELECT convert(CHAR(20), NationalIDNumber) AS IDNUMBER FROM HumanResources.Employee`
datalength (expression)	*Returns the number of bytes used to represent the specified expression.*	`SELECT datalength ('Test Expression') AS LengthOfData`

The System Functions Provided by SQL Server

Activity 2.2: Customizing the Result Set

Summarizing and Grouping Data

At times, the users need to view a summary of the data. Summary of the data contains aggregated values that help in data analysis at a broader level. For example, to analyze the sales, the users might want to view the average sales or total sales for a specified time period. SQL Server provides aggregate functions to generate summarized data. The users might also want to view the summarized data in different groups based on specific criteria. For example, the users want to view the average sales data regionwise or productwise. In such a case, the sales data of each region will be displayed together. You can group the data by using the GROUP BY clause of the SELECT statement. You can also use aggregate functions to summarize data when grouping it.

Summarizing Data by Using Aggregate Functions

At times, you need to calculate the summarized values of a column based on a set of rows. For example, the salary of employees is stored in the Rate column of the EmployeePayHistory table and you need to calculate the average salary earned by the employees.

The aggregate functions, on execution, summarize the values of a column or a group of columns, and produce a single value. The syntax of an aggregated function is:

```
SELECT        aggregate_function([ALL|
DISTINCT] expression)
FROM table_name
```

where,

`ALL` specifies that the aggregate function is applied to all the values in the specified column.

`DISTINCT` specifies that the aggregate function is applied to only unique values in the specified column.

`expression` specifies a column or an expression with operators.

You can calculate summary values by using the following aggregate functions:

- **Avg()**: Returns the average of values in a numeric expression, either all or distinct.
  The following SQL query retrieves the average value from the Rate column of the EmployeePayHistory table with a user-defined heading:
  ```
  SELECT 'Average Rate' = avg
  (Rate) FROM
  HumanResources.EmployeePayHistory
  ```
- **Count()**: Returns the number of values in an expression, either all or distinct. The following SQL query retrieves the unique rate values from the EmployeePayHistory table with a user-defined heading:
  ```
  SELECT 'Unique Rate' = count
  (DISTINCT Rate)
  FROM
  HumanResources.EmployeePayHistory
  ```
  The count() function also accepts (*) as its parameter, but it counts the number of rows returned by the query.
- **Min()**: Returns the lowest value in the expression. The following SQL query retrieves the minimum value from the Rate column of the EmployeePayHistory table with a user-defined heading:
  ```
  SELECT 'Minimum Rate' = min
  (Rate)
  FROM
  HumanResources.EmployeePayHistory
  ```
- **Max()**: Returns the highest value in the expression. The following SQL query retrieves the maximum value from the Rate column of the

EmployeePayHistory table with a user-defined heading:

```
SELECT 'Maximum Rate' = max (Rate)
FROM HumanResources.EmployeePayHistory
```

- **Sum()**: Returns the sum total of values in a numeric expression, either all or distinct. The following SQL query retrieves the sum value of all the unique rate values from the EmployeePayHistory table with a user-defined heading:

```
SELECT 'Sum' = sum (DISTINCT Rate) FROM HumanResources.EmployeePayHistory
```

Grouping Data

At times, you need to view data matching specific criteria to be displayed together in the result set. For example, you want to view a list of all the employees with details of employees of each department displayed together.

You can group the data by using the GROUP BY, ROLLUP, CUBE, and PIVOT clauses of the SELECT statement.

GROUP BY

The GROUP BY clause summarizes the result set into groups, as defined in the SELECT statement, by using aggregate functions. The HAVING clause further restricts the result set to produce the data based on a condition. The syntax of the GROUP BY clause is:

```
SELECT column_list
FROM table_name
WHERE condition
[GROUP BY [ALL] expression [, expression]
[HAVING search_condition]
```

where,

`ALL` is a keyword used to include those groups that do not meet the search condition.

`expression` specifies the column name(s) or expression(s) on which the result set of the SELECT statement is to be grouped.

`search_condition` is the conditional expression on which the result is to be produced.

The following SQL query returns the minimum and maximum values of vacation hours for the different types of titles where the number of hours that the employees can avail to go on a vacation is greater than 80:

```
SELECT Title, Minimum = min (VacationHours), Maximum = max (VacationHours) FROM HumanResources.Employee WHERE VacationHours > 80 GROUP BY Title
```

The following figure displays the output of the preceding query.

	Title	Minimum	Maximum
1	Chief Executive Officer	99	99
2	Facilities Administrative Assistant	87	87
3	Facilities Manager	86	86
4	Janitor	88	91
5	Maintenance Supervisor	92	92
6	Production Supervisor - WC60	81	82
7	Production Technician - WC10	83	99
8	Production Technician - WC45	81	87
9	Production Technician - WC50	88	99
10	Quality Assurance Supervisor	81	81
11	Quality Assurance Technician	82	85

The Output Derived After Executing the Query

The GROUP BY...HAVING clause is same as the SELECT...WHERE clause. The GROUP BY clause collects data that matches the condition and summarizes it into an expression to produce a single value for each group. The HAVING clause eliminates all those groups that do not match the specified condition.

The following query retrieves all the titles along with their average vacation hours when the vacation hours are more than 30 and the group average value is greater than 55:

```
SELECT Title, 'Average Vacation Hours' = avg(VacationHours) FROM HumanResources.Employee WHERE VacationHours > 30 GROUP BY Title HAVING avg(VacationHours) >55
```

The GROUP BY clause can be applied on multiple fields. You can use the following query to retrieve the average value of the vacation hours that is grouped by Title and ManagerID in the Employee table:

```
SELECT Title, 'Manager ID' = ManagerID, Average = avg (VacationHours) FROM HumanResources.Employee GROUP BY Title, ManagerID
```

The following figure displays the output of the preceding query.

	Title	Manager ID	Average
1	Chief Executive Officer	NULL	99
2	Design Engineer	3	5
3	Research and Development Manager	3	61
4	Senior Design Engineer	3	3
5	Senior Tool Designer	3	27
6	Marketing Assistant	6	42
7	Marketing Specialist	6	46
8	Production Technician - WC60	7	36
9	Engineering Manager	12	2
10	Production Technician - WC50	14	10
11	Production Technician - WC60	16	19

The Output Derived After Executing the Query

If you want to display all those groups that are excluded by the WHERE clause, then you can use the ALL keyword along with the GROUP BY clause.

For example, the following query retrieves the records for the employee titles that are eliminated in the WHERE condition:

```
SELECT Title, VacationHours = sum (VacationHours) FROM HumanResources.Employee WHERE Title IN ('Recruiter', 'Stocker', 'Design Engineer') GROUP BY ALL Title ORDER BY sum (VacationHours)DESC
```

The following figure displays the output of the preceding query.

	Title	VacationHours
1	Stocker	291
2	Recruiter	99
3	Design Engineer	15
4	Document Control Assistant	NULL
5	Document Control Manager	NULL
6	Engineering Manager	NULL
7	European Sales Manager	NULL
8	Facilities Administrative Assistant	NULL
9	Facilities Manager	NULL
10	Finance Manager	NULL
11	Human Resources Administrative Assistant	NULL

The Output Derived After Executing the Query

If you write the preceding query using GROUP BY clause, it will display three records.

The GROUPING SETS clause is used to combine the result generated by multiple GROUP BY clauses into a single result set. For example, the employee details of the organization are stored in the following EmpTable table.

	EmpName	Region	Department	sal
1	Max	North America	Information Technology	25000.00
2	Andrew	South America	Information Technology	28000.00
3	Maria	North America	Human Resources	36000.00
4	Stephen	Middle East Asia	Information Technology	40000.00
5	Steve	Middle East Asia	Human Resources	60000.00

The EmpTable Table

You want to view the average salary of the employees combined for each region and department. You also want to view the average salary of the employees regionwise and departmentwise. To view the average salary of the employees combined for each region and department, you can use the following query:

```
SELECT Region, Department, avg(sal) AverageSalary
FROM EmpTable
GROUP BY
Region, Department
```

To view the average salary of the employees for each region, you can use the following query:

```
SELECT Region, avg(sal) AverageSalary
FROM EmpTable
GROUP BY
Region
```

To view the average salary of the employees for each department, you can use the following query:

```
SELECT Department, avg(sal) AverageSalary
FROM EmpTable
GROUP BY
Department
```

Using the preceding queries, you can view the average salary of the employees based on different grouping criteria. If you want to view the results of all the previous three queries in a single result set, you need to perform the union of the results generated from the preceding queries. However, instead of performing the union of the results, you can use the GROUPING SET clause, as shown in the following query:

```
SELECT Region, Department, AVG(sal) AverageSalary
FROM EmpTable
GROUP BY
    GROUPING SETS
    (
        (Region, Department),
        (Region),
        (Department)
    )
```

The following figure displays the output of the preceding query.

	Region	Department	AverageSalary
1	Middle East Asia	Human Resources	60000.00
2	North America	Human Resources	36000.00
3	NULL	Human Resources	48000.00
4	Middle East Asia	Information Technology	40000.00
5	North America	Information Technology	25000.00
6	South America	Information Technology	28000.00
7	NULL	Information Technology	31000.00
8	Middle East Asia	NULL	50000.00
9	North America	NULL	30500.00
10	South America	NULL	28000.00

The Output Derived After Using the GROUPING SET Clause

In the preceding figure, the rows that do not have NULL values represent the average salary of the employees grouped for each region and department. The rows that contain NULL values in the Department column represent the average salary of the employees for each region. The rows that contain NULL values in the Region column represent the average salary of the employees for each department.

Working with the ROLLUP and CUBE Operators

Consider a scenario, where the management of Tebisco, Inc. wants to view the details of the sales data of previous years. The management wants to view a report that shows the total sales amount earned by each employee during the previous years and the total sales amount earned by all the employees in the previous years. In addition, the management wants to view a report that shows the yearwise sum of sales amount earned by each employee during the previous years.

To generate the result sets required in the preceding scenario, you need to apply multiple levels of aggregation. You can use the ROLLUP and CUBE operators to apply multiple levels of aggregation on result sets and generate the required report.

Using the ROLLUP Operator

The ROLLUP operator is an extension of the GROUP BY clause. This operator can be used to apply multiple levels of aggregation on results retrieved from a table. The ROLLUP operator generates a result set that contains a subtotal for each group and a grand total of all the groups.

Consider a scenario. The sales data of Tebisco, Inc. is stored in the SalesHistory table. The data stored in the SalesHistory table is shown in the following table.

EmployeeID	***YearOfSale***	***SalesAmount***
101	*2007*	*120000.00*
101	*2008*	*140000.00*
101	*2009*	*250000.00*
102	*2007*	*150000.00*
102	*2008*	*120000.00*
102	*2009*	*110000.00*
103	*2007*	*105000.00*
103	*2008*	*180000.00*
103	*2009*	*160000.00*
104	*2007*	*170000.00*
104	*2008*	*120000.00*
104	*2009*	*150000.00*

The Data Stored in the SalesHistory Table

You have been assigned the task to generate a report that shows the total sales amount earned by each employee during the previous years and the total sales amount earned by all the employees in the previous years.

To generate the report required to accomplish the preceding task, you need to apply the following levels of aggregation:

- Sum of sales amount earned by each employee in the previous years
- Sum of sales amount earned by all the employees in the previous years

Therefore, you can use the ROLLUP operator to apply the preceding levels of aggregation and generate the required result set, as shown in the following query:

```
SELECT EmployeeID, YearOfSale, SUM
(SalesAmount) AS SalesAmount
FROM SalesHistory
GROUP BY ROLLUP(EmployeeID,
YearOfSale)
```

In the preceding query, the EmployeeID and YearOfSale columns are specified with the ROLLUP operator because the result is to be generated for each employee as well as for each year of sale.

The following figure shows the output of the preceding query.

	EmployeeID	YearOfSale	SalesAmount
1	101	2007	120000.00
2	101	2008	140000.00
3	101	2009	250000.00
4	101	NULL	510000.00
5	102	2007	150000.00
6	102	2008	120000.00
7	102	2009	110000.00
8	102	NULL	380000.00
9	103	2007	105000.00
10	103	2008	180000.00
11	103	2009	160000.00
12	103	NULL	445000.00
13	104	2007	170000.00
14	104	2008	120000.00
15	104	2009	150000.00
16	104	NULL	440000.00
17	NULL	NULL	1775000.00

The Output Derived After Using the ROLLUP Operator

The preceding figure displays the sum of the sales amount earned by each employee during the previous years. The amount earned by employee 101 is displayed in row number 4. The NULL in this row represents that it contains the sum of the preceding rows. Similarly, the amount earned by employee 102 is displayed in row number 8, by employee 103 is displayed in row number 12, and by employee 104 is displayed in row number 16. In addition, the output displays the grand total of the sales amount earned by all the employees during the previous years in row number 17.

Using the CUBE Operator

The CUBE operator is also used to apply multiple levels of aggregation on the result retrieved from a table. However, this operator extends the functionality of the ROLLUP operator and generates a result set with all the possible combination of the records retrieved from a table.

For example, you have to generate a report by retrieving data from the SalesHistory table. The generated report should show the yearwise total amount of sale by all the employees, total amount of sale of all the previous years, and total amount of sale earned by each employee during all the previous years.

To generate a result set required to accomplish the preceding task, you need to apply the following levels of aggregation:

- Sum of sales amount for each year
- Sum of sales amount for all years
- Sum of sales amount for each employee in all years

Therefore, you can use the CUBE operator to apply the preceding levels of aggregation and generate the required result set, as shown in the following query:

```
SELECT EmployeeID, YearOfSale, SUM
(SalesAmount) AS SalesAmount
FROM SalesHistory
GROUP BY CUBE(EmployeeID, YearOfSale)
```

The following figure shows the result of the preceding query.

	EmployeeID	YearOfSale	SalesAmount
1	101	2007	120000.00
2	102	2007	150000.00
3	103	2007	105000.00
4	104	2007	170000.00
5	NULL	2007	545000.00
6	101	2008	140000.00
7	102	2008	120000.00
8	103	2008	180000.00
9	104	2008	120000.00
10	NULL	2008	560000.00
11	101	2009	250000.00
12	102	2009	110000.00
13	103	2009	160000.00
14	104	2009	150000.00
15	NULL	2009	670000.00
16	NULL	NULL	1775000.00
17	101	NULL	510000.00
18	102	NULL	380000.00
19	103	NULL	445000.00
20	104	NULL	440000.00

The Output Derived After Using the CUBE Operator

The preceding figure displays the sum of the sales amount earned by all the employees during each year. The amount earned by all the employees in year 2007 is displayed in row number 5. Similarly, amount earned by all the employees in year 2008 and 2009 is displayed in row number 10 and 15, respectively. Further, it displays the

grand total of the sales amount earned by all the employees during the previous years in row number 16. In addition, the output displays the sum of sales amount earned by each employee during the previous years in row numbers 17 to 20. The output in the last four rows is similar to the output in row numbers, 4, 8, 12, and 16 displayed by using the ROLLUP operator.

Differences Between the ROLLUP and CUBE Operators

Both the ROLLUP and CUBE operators are used to apply multiple levels of aggregation on the result set retrieved from a table. However, these operators are different in terms of result sets they produce. The differences between the ROLLUP and CUBE operators are:

- For each value in the columns on the right side of the GROUP BY clause, the CUBE operator reports all possible combinations of values from the columns on the left side. However, the ROLLUP operator does not report all such possible combinations.
- The number of groupings returned by the ROLLUP operator equals the number of columns specified in the GROUP BY clause plus one. However, the number of groupings returned by the CUBE operator equals the double of the number of columns specified in the GROUP BY clause.

PIVOT

The database users might need to view data in a user-defined format. These reports might involve summarizing data on the basis of various criteria. SQL Server allows you to generate summarized data reports using the PIVOT clause of the SELECT statement.

The PIVOT clause is used to transform a set of columns into values. PIVOT rotates a table-valued expression by turning the unique values from one column in the expression into multiple columns in the output. In addition, it performs aggregations on the remaining column values if required in the output. The syntax of the PIVOT operator is:

```
SELECT * from table_name
PIVOT              (aggregation_function
(value_column)
FOR pivot_column
IN (column_list)
) table_alias
```

Consider an example, where you want to display the number of purchase orders placed by certain employees, laid down with the vendors. The following query provides this report:

```
SELECT VendorID, [164] AS Emp1, [198]
AS Emp2, [223] AS Emp3, [231] AS Emp4,
[233] AS Emp5
FROM
(SELECT  PurchaseOrderID,  EmployeeID,
VendorID
FROM Purchasing.PurchaseOrderHeader) p
PIVOT
(
count (PurchaseOrderID)
FOR EmployeeID IN
( [164], [198], [223], [231], [233] )
) AS pvt
ORDER BY VendorID
```

The following figure displays the output of the preceding query.

	VendorID	Emp1	Emp2	Emp3	Emp4	Emp5
1	1	4	3	5	4	4
2	2	4	1	5	5	5
3	3	4	3	5	4	4
4	4	4	2	5	5	4
5	5	5	1	5	5	5
6	6	0	0	0	1	0
7	7	5	2	5	5	5
8	8	4	2	5	4	5
9	10	4	1	5	4	5
10	11	4	2	6	4	4
11	12	5	2	5	5	4
12	13	5	1	5	5	5
13	15	4	3	5	4	4

The Output Derived After Using the PIVOT Operator

Unpivot

The UNPIVOT operator allows database users to normalize the data that has earlier been pivoted. This operator transforms the multiple column values of a record into multiple records with the same values in a single column. For example, consider a table named Applicant

that stores the details of applicants. This table stores the applicant's name and grade secured by the applicants in matriculation, higher secondary, and graduation.
The following table shows the data stored in the Applicant table.

ApplicantName	Matriculation	HigherSecondary	Graduation
Anderson	A	A	B
Samuel	A	B	A
Sandra	B	B	B

The Data Stored in the Applicant Table

In the preceding table, the Matriculation, HigherSecondary, and Graduation columns show the grade secured by the applicants. However, if you want to display different rows for each qualification secured by the applicants, you can use the UNPIVOT operator, as shown in the following query:

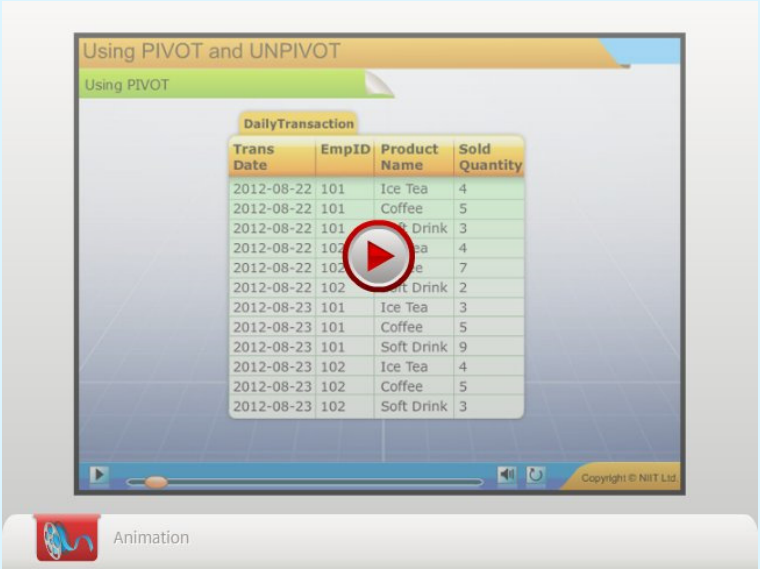
```
SELECT ApplicantName, Qualification,
Grade
FROM
    (SELECT ApplicantName,
Matriculation, HigherSecondary,
Graduation
    FROM Applicant) Ap
UNPIVOT
    (Grade FOR Qualification IN
      (Matriculation, HigherSecondary,
Graduation)
) AS unpivotedtbl
```

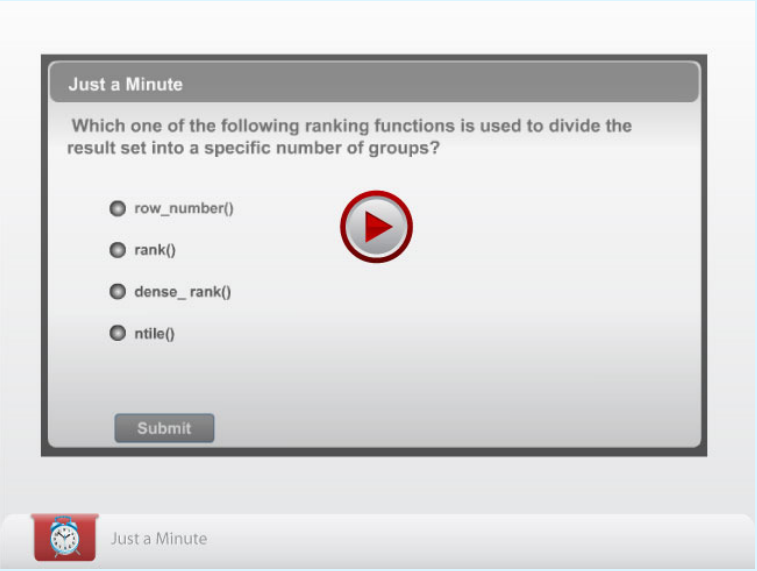
The following figure shows the output of the preceding query.

	ApplicantName	Qualification	Grade
1	Anderson	Matriculation	A
2	Anderson	HigherSecondary	A
3	Anderson	Graduation	B
4	Samuel	Matriculation	A
5	Samuel	HigherSecondary	B
6	Samuel	Graduation	A
7	Sandra	Matriculation	B
8	Sandra	HigherSecondary	B
9	Sandra	Graduation	B

The Output Derived After Using the UNPIVOT Operator

In the preceding figure, the table contains different rows for each qualification that the applicant has completed.





Activity 2.3: Summarizing and Grouping Data

Summary

In this chapter, you learned that:

- Data type represents the type of data that a database object can contain.
- You can store the following types of data in a database:
  - Structured data
  - Semi-structured data
  - Unstructured data
- Data can be retrieved from a database by using the SELECT statement.
- Data of all the columns of a table can be retrieved by specifying * in the SELECT statement.
- Data that has to be retrieved based on a condition is specified by adding the WHERE clause.

- Synonyms are created to refer to a database object easily.
- Literals and user-defined headings are added to change the display.
- The concatenation operator is used to concatenate a string expression.
- Arithmetic operators are used to perform mathematical operations.
- Comparison operators test the similarity between two expressions.
- Logical operators are used in the SELECT statement to retrieve records based on one or matching conditions. The logical operators are AND, OR, and NOT.
- Range operators retrieve data based on a range. There are two types of range operators, BETWEEN and NOT BETWEEN.
- The IN keyword allows the selection of values that match any one of the values in a list.
- The NOT IN keyword restricts the selection of values that match any one of the values in a list.
- The LIKE keyword is used to specify the pattern search.
- The IS NULL keyword is used to retrieve missing values.
- The ORDER BY clause is used to retrieve data in a specific order.
- The TOP keyword retrieves only the first set of rows, which can either be a number or a percent of rows that will be returned from a query result.
- The OFFSET-FETCH clause is used to retrieve a number of records from anywhere in the result set.
- The DISTINCT keyword eliminates duplicate rows.
- The string functions are used to format data in the result set.
- The date functions are used to manipulate date values.
- The mathematical functions are used to perform numerical operations.
- The conversion functions are used to convert a data from one type to another.
- The ranking functions are used to generate sequential numbers for each row or to give a rank based on specific criteria.
- You can use the following functions to rank the records:
  - row_number()
  - rank()
  - dense_ rank()
  - ntile()
- The analytical functions are used to perform comparisons between rows in the same table.
- The system functions are used to query system tables.
- The aggregate functions, such as avg(), count(), min(), max(), and sum() are used to retrieve summarized data.
- The GROUP BY, GROUP BY ALL, and PIVOT clauses are used to group the result set.
- The GROUPING SETS clause is used to combine the results generated by multiple GROUP BY clauses into a single result set.
- The ROLLUP and CUBE operators are used to apply multiple levels of aggregation on result sets.
- The UNPIVOT operator allows to normalize the data that has earlier been pivoted.

Reference Reading

Retrieving Data

Reference Reading: Books	***Reference Reading: URLs***
Sams Teach Yourself SQL in 24 Hours by Ryan Stephens and Ron Plew	*http://msdn.microsoft.com/en-us/library/ms189499.aspx*

Using Functions to Customize the Result Set

Reference Reading: Books	***Reference Reading: URLs***
Microsoft SQL Server 2012 Step by Step by Patrick LeBlanc	*http://msdn.microsoft.com/en-us/library/ms174318.aspx*

Summarizing and Grouping Data

Reference Reading: Books	***Reference Reading: URLs***
Sams Teach Yourself SQL in 24 Hours by Ryan Stephens and Ron Plew *Microsoft SQL Server 2012 Step by Step by Patrick LeBlanc*	*http://msdn.microsoft.com/en-us/library/ms191500%28v=SQL.105%29.aspx*

Chapter 3

Querying Data by Using Joins and Subqueries

In a normalized database, data related to an entity may be stored in multiple tables. When you need to view data from related tables together, you can *join* the tables with the help of common attributes.

You can also use subqueries where the result of a query is used as an input for the condition of another query. In addition, you can store the result of a query as a temporary result set by using *Common Table Expressions (CTE)*. This temporary result set exists only within the scope of a single SQL statement. In addition you can also use Derived Tables to simplify the queries involving complex aggregate calculations.

This chapter discusses how to query data from multiple tables by applying various types of joins, such as inner join, *outer join*, *cross join*,*equi join*, or *self join*. Further, it explains how to use subqueries. Lastly, this chapter discusses about temporary result sets created by using CTE.

Objectives

In this chapter, you will learn to:

- Query data by using joins
- Query data by using subqueries
- Manage result sets

Querying Data by Using Joins

When an SQL query executes, it returns a result set. A result set is a set of rows retrieved from a table. As a database developer, you may need to retrieve data from more than one table and display it in a single result set. In such a case, different columns in the result set can obtain data from different tables. To retrieve data from multiple tables, SQL Server allows you to apply *joins*. Joins allow you to view data from related tables in a single result set. You can join more than one table based on a common attribute.

Depending on the requirements to view data from multiple tables, you can apply different types of joins, such as inner join, outer join, cross join, equi join, or self join.

Using an Inner Join

An *inner join* retrieves records from multiple tables after comparing values present in a common column. When an inner join is applied, only the rows with values satisfying the join condition in the common column are displayed. The rows in both the tables that do not satisfy the join condition are not displayed.

For example, the details of movies are stored in the following Movies table.

MovieID	***MovieName***	***YearMade***
1	*My Fair Lady*	*1964*
2	*Unforgiven*	*1992*
3	*Time Machine*	*1997*

The Movies Table

The details of actors are stored in the following Actors table.

MovieID	***FirstName***	***LastName***
1	*Rex*	*Harrison*
1	*Audrey*	*Hepburn*
2	*Clint*	*Eastwood*
5	*Humphrey*	*Bogart*

The Actors Table

If you join the Movies table and Actors table using an inner join, the result set will be retrieved as shown in the following table.

MovieID	MovieName	YearMade	FirstName	LastName
1	My Fair Lady	1964	Rex	Harrison
1	My Fair Lady	1964	Audrey	Hepburn
2	Unforgiven	1992	Clint	Eastwood

The Result Set Obtained After Joining the Movies Table and the Actors Table

Notice that the last row of the Actors table is not present in the result set because the corresponding matching record is not found in the Movies table.

A join is implemented by using the SELECT statement, where the SELECT list contains the names of the columns to be retrieved from the tables. The FROM clause contains the names of the tables from which combined data is to be retrieved. The WHERE clause specifies the condition, with a comparison operator, based on which the tables will be joined.

The syntax of applying an inner join in the SELECT statement is:

```
SELECT column_name, column_name
[,column_name]
 FROM table1_name JOIN table2_name
 ON table1_name.ref_column_name
join_operator
table2_name.ref_column_name
```

[WHERE search_condition]

where,

table1_name and table2_name are the names of the tables that are joined.

join_operator is the comparison operator based on which the join is applied.

table1_name.ref_column_name and table2_name.ref_column_name are the names of the columns on which the join is applied.

WHERE specifies the search condition for the rows returned by the query.

search_condition specifies the condition to be satisfied to return the selected rows.

An inner join is the default join. Therefore, you can also apply an inner join by using the JOIN keyword. In addition, you can use the INNER JOIN keyword.

Whenever a column is mentioned in a join condition, it should be referred by prefixing it with the table name to which it belongs or with a table alias. A table alias is used to refer to the table with another name or to uniquely identify the table. The table alias is defined in the FROM clause of the SELECT statement.

When listing the column names in the SELECT statement, it is mandatory to use a table name or a table alias if an ambiguity arises due to duplicate column names in multiple tables.

The following query displays the Employee ID and Title for each employee from the Employee table, and the Rate and PayFrequency columns from the EmployeePayHistory table:

```
SELECT e.EmployeeID,e.Title,
eph.Rate,eph.PayFrequency
 FROM HumanResources.Employee e JOIN
HumanResources.EmployeePayHistory eph
ON e.EmployeeID = eph.EmployeeID
```

In the preceding query, the Employee table and the EmployeePayHistory table are joined on the common column, EmployeeID. The query also assigns e as an alias of the Employee table and eph as an alias of the EmployeePayHistory table. The column names are also listed with the table alias names.

The following figure displays the data of the Employee table.

	EmployeeID	NationalIDNumber	ContactID	LoginID	ManagerID	Title
1	1	14417807	1209	adventure-works\guy1	16	Production Technician - WC60
2	2	253022876	1030	adventure-works\kevin0	6	Marketing Assistant
3	3	509647174	1002	adventure-works\roberto0	12	Engineering Manager
4	4	112457891	1290	adventure-works\rob0	3	Senior Tool Designer
5	5	480168528	1009	adventure-works\thierry0	263	Tool Designer
6	6	24756624	1028	adventure-works\david0	109	Marketing Manager
7	7	309738752	1070	adventure-works\jolynn0	21	Production Supervisor - WC60
8	8	690627818	1071	adventure-works\ruth0	185	Production Technician - WC10
9	9	695256908	1005	adventure-works\gail0	3	Design Engineer
10	10	912265825	1076	adventure-works\barry0	185	Production Technician - WC10

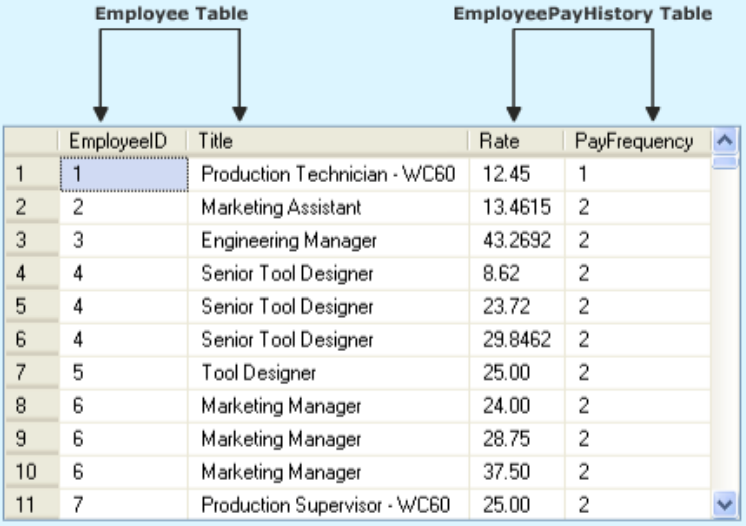
The Employee Table

The following figure displays the data of the EmployeePayHistory table.

	EmployeeID	RateChangeDate	Rate	PayFrequency	ModifiedDate
1	1	1996-07-31 00:00:00.000	12.45	1	2004-07-31 00:00:00.000
2	2	1997-02-26 00:00:00.000	13.4615	2	2004-07-31 00:00:00.000
3	3	1997-12-12 00:00:00.000	43.2692	2	2004-07-31 00:00:00.000
4	4	1998-01-05 00:00:00.000	8.62	2	1997-12-22 00:00:00.000
5	4	2000-07-01 00:00:00.000	23.72	2	2000-06-16 00:00:00.000
6	4	2002-01-15 00:00:00.000	29.8462	2	2002-01-01 00:00:00.000
7	5	1998-01-11 00:00:00.000	25.00	2	2004-07-31 00:00:00.000
8	6	1998-01-20 00:00:00.000	24.00	2	1998-01-06 00:00:00.000
9	6	1999-08-16 00:00:00.000	28.75	2	1999-08-02 00:00:00.000
10	6	2002-06-01 00:00:00.000	37.50	2	2002-05-18 00:00:00.000
11	7	1998-01-26 00:00:00.000	25.00	2	2004-07-31 00:00:00.000

The EmployeePayHistory Table

The following figure displays the output of the preceding inner join query.



	EmployeeID	Title	Rate	PayFrequency
1	1	Production Technician - WC60	12.45	1
2	2	Marketing Assistant	13.4615	2
3	3	Engineering Manager	43.2692	2
4	4	Senior Tool Designer	8.62	2
5	4	Senior Tool Designer	23.72	2
6	4	Senior Tool Designer	29.8462	2
7	5	Tool Designer	25.00	2
8	6	Marketing Manager	24.00	2
9	6	Marketing Manager	28.75	2
10	6	Marketing Manager	37.50	2
11	7	Production Supervisor - WC60	25.00	2

The Output Derived by Executing the Inner Join Query

Based on the relationship between tables, you need to select the common column to set the join condition. In the preceding inner join query, the common column is EmployeeID, which are the primary key of the Employee table and the foreign key of the EmployeePayHistory table.

Observe the result set after the join. The record of the employee with EmployeeID 1 from the Employee table is joined with the record of the employee with EmployeeID 1 from the EmployeePayHistory table.

While applying joins, you can also check for other conditions. The following query retrieves the employee ID and the designation from the Employee table for all the employees whose pay rate is greater than 40:

```
SELECT 'Employee ID' = e.EmployeeID,
'Designation' = e.Title
 FROM HumanResources.Employee e INNER
JOIN HumanResources.EmployeePayHistory
eph
 ON e.EmployeeID = eph.EmployeeID
WHERE eph.Rate>40
```

In the preceding query, the tables are joined based on the

EmployeeID column, which is common in both the tables. In addition, only those records are selected where the value in the Rate column of the EmployeePayHistory table is greater than 40.

Using an Outer Join

In comparison to an inner join, an *outer join* displays the result set containing all the rows from one table and the matching rows from the other table. For example, if you create an outer join on table A and table B, it will show you all the records of table A and only those records from table B for which the condition on the common column holds true.

An outer join displays NULL for the columns of the related table where it does not find any matching records. The syntax of applying an outer join is:

```
SELECT column_name, column_name
[,column_name]
 FROM table1_name [LEFT | RIGHT| FULL]
OUTER JOIN table2_name
 ON table1_name.ref_column_name
join_operator
table2_name.ref_column_name
 [WHERE search_condition]
```

An outer join is of the following types:

- Left outer join
- Right outer join
- Full outer join

Using a Left Outer Join

A left outer join returns all rows from the table specified on the left side of the LEFT OUTER JOIN keyword and the matching rows from the table specified on the right side. It displays NULL for the columns of the table specified on the right side where it does not find any matching records.

For example, you need to display the details of all the movies and the corresponding details of the actors who acted in those movies. To perform this task, you can join the Movies table and the Actors table by using a left outer join. The Movies table will be on the left side of the LEFT OUTER JOIN keyword in the query. The following figure displays the result set of using the left outer join between the Movies table and the Actors table.

MovieID	MovieName	YearMade	FirstName	LastName
1	My Fair Lady	1964	Rex	Harrison
1	My Fair Lady	1964	Audrey	Hepburn
2	Unforgiven	1992	Clint	Eastwood
3	Time Machine	1997	NULL	NULL

The Result Set Obtained After Joining the Movies Table and the Actors Table

In the preceding figure, the FirstName and LastName columns have the value NULL against the movie, Time Machine. This is because there are no records of actors in the Actors table for the Time Machine movie.

Consider another example from the AdventureWorks, Inc. The SpecialOfferProduct table contains a list of products that are on special offer and the SalesOrderDetail table stores the details of all the sales transactions. The users at AdventureWorks, Inc. need to view the transaction details of the special offer products. In addition, they want to view the ProductID of the special offer products for which no transaction has been done.

To perform this task, you can use the LEFT OUTER JOIN keyword, as shown in the following query:

```
SELECT DISTINCT (p.ProductID),
p1.SpecialOfferID, p1.SalesOrderID,
p1.OrderQty, p1.UnitPrice
 FROM Sales.SpecialOfferProduct p LEFT
OUTER JOIN
 [Sales].[SalesOrderDetail] p1 ON
p.ProductID =p1.ProductID
 ORDER BY p.ProductID
```

The following figure displays the output of the preceding query.

	ProductID	SpecialOfferID	SalesOrderID	OrderQty	UnitPrice
1	680	NULL	NULL	NULL	NULL
2	706	NULL	NULL	NULL	NULL
3	707	1	43665	1	20.1865
4	707	1	43668	2	20.1865
5	707	1	43673	4	20.1865
6	707	1	43677	1	20.1865
7	707	1	43678	1	20.1865
8	707	1	43680	4	20.1865
9	707	1	43681	1	20.1865
10	707	1	43683	3	20.1865
11	707	1	43692	4	20.1865

The Output Derived After Executing the Query

The preceding figure displays NULL for the ProductIDs for which no transaction was performed.

Using a Right Outer Join

A right outer join returns all the rows from the table specified on the right side of the RIGHT OUTER JOIN keyword and the matching rows from the table specified on the left side.

For example, you need to display the details of all the actors and the corresponding details of the movies in which they have acted. To perform this task, you can join the Movies table and the Actors table by using a right outer join. The Actors table will be on the right side of the RIGHT OUTER JOIN keyword in the query.

The following figure displays the result set of using the right outer join between the Movies table and the Actors

table.

MovieID	MovieName	YearMade	FirstName	LastName
1	My Fair Lady	1964	Rex	Harrison
1	My Fair Lady	1964	Audrey	Hepburn
2	Unforgiven	1992	Clint	Eastwood
NULL	NULL	NULL	Humphrey	Bogart

The Result Set Obtained After Joining the Movies Table and the Actors Table

In the preceding figure, NULL is displayed against the MovieID, MovieName, and YearMade columns for the actor, Humphrey Bogart. This is because there are no records for the actor, Humphrey Bogart in the Movies table.

Consider the example of AdventureWorks, Inc. You want to display the IDs of all the sales persons along with the territory assigned to them. In addition, you need to display the IDs of all those sales persons who have not been assigned any territory. To perform this task, you can apply a right outer join between the Sales.SalesTerritory table and the Sales.SalesPerson table, as shown in the following query:

```
SELECT st.Name AS Territory,
sp.SalesPersonID
 FROM Sales.SalesTerritory st
 RIGHT OUTER JOIN Sales.SalesPerson sp
 ON st.TerritoryID = sp.TerritoryID
```

The following figure displays the output of the preceding query.

	Territory	SalesPersonID
1	NULL	268
2	Northeast	275
3	Southwest	276
4	Central	277
5	Canada	278
6	Southeast	279
7	Northwest	280
8	Southwest	281
9	Canada	282
10	Northwest	283
11	NULL	284
12	United Kingdom	285

The Output Derived After Executing the Query

In the preceding figure, NULL is displayed in the Territory column for those sales persons who have not been assigned any territory.

Using a Full Outer Join

A full outer join is a combination of left outer join and right outer join. This join returns all the matching and non-matching rows from both the tables. However, the matching records are displayed only once. In case of non-matching rows, a NULL value is displayed for the columns for which data is not available.

For example, you need to display the details of all the movies along with all the actors combined in a single result set. To perform this task, you can join the Movies table and the Actors table by using a full outer join. The following figure displays the result set of using the full outer join between the Movies table and the Actors table.

MovieID	MovieName	YearMade	FirstName	LastName
1	My Fair Lady	1964	Rex	Harrison
1	My Fair Lady	1964	Audrey	Hepburn
2	Unforgiven	1992	Clint	Eastwood
3	Time Machine	1997	NULL	NULL
NULL	NULL	NULL	Humphrey	Bogart

The Result Set Obtained After Joining the Movies Table and the Actors Table

Consider another example of AdventureWorks, Inc. You want to display the IDs of all the sales persons along with the territory assigned to them. In addition, you need to display the IDs of all those sales persons who have not been assigned any territory. Similarly, you need to display all those territories which have not been assigned to any sales person. To perform this task, you can apply a full outer join between the Sales.SalesTerritory table and the Sales.SalesPerson table, as shown in the following query:

```
SELECT st.Name AS Territory,
sp.SalesPersonID
 FROM Sales.SalesTerritory st
 FULL OUTER JOIN Sales.SalesPerson sp
 ON st.TerritoryID = sp.TerritoryID
```

The following figure displays the output of the preceding query.

	Territory	SalesPersonID
1	Northwest	280
2	Northwest	283
3	Northwest	287
4	Northeast	275
5	Central	277
6	Southwest	276
7	Southwest	281
8	Southeast	279
9	Canada	278
10	Canada	282
11	France	286

The Output Derived After Executing the Query

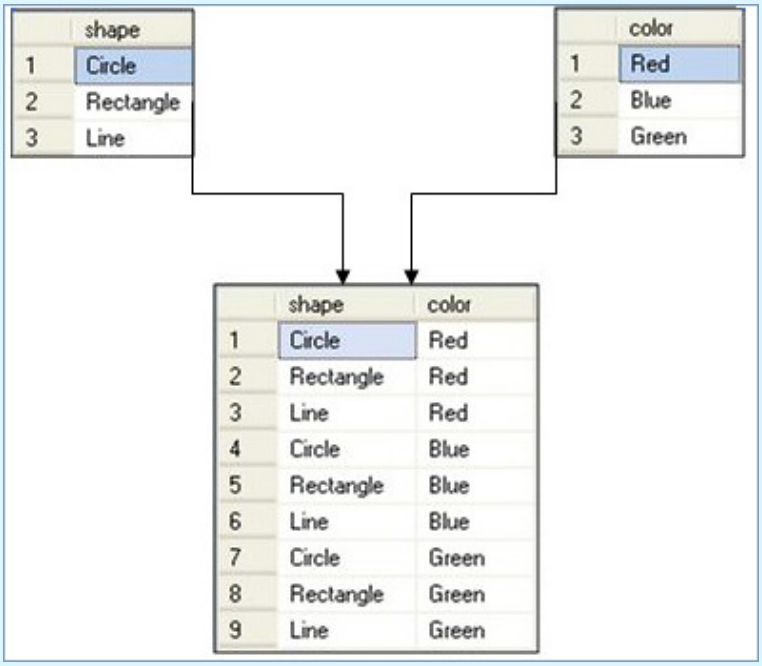
The preceding query will display NULL values in the Territory column for those sales persons who have not been assigned any territory. Similarly, it will display NULL values in the SalesPersonID column for those territories which have not been assigned to any sales person.

Using a Cross Join

A *cross join* is also known asthe *Cartesian Product*. It joins each row of one table with each row of the other table. The number of rows in the result set is equal to the number of rows in the first table multiplied by the number of rows in the second table. This implies that if table A has 10 rows and table B has 5 rows, then all the 10 rows of table A are joined with all the 5 rows of table B. Therefore, the result set will contain 50 rows.

For example, table A contains three shapes: Circle, Rectangle, and Line. Table B contains three colors: Red, Blue, and Green.

If you join these two tables using the cross join, the result set will contain nine rows, as shown in the following figure.



The Result Set Obtained After Cross Joining the Shape Table and Color Table

Consider another example of a storehouse that sells computers. As a database developer, you have saved the configuration and price details of computers in the ComputerDetails table, as shown in the following figure.

	CompID	CompDescription	Price
1	1	Pentium D 3 GHtz, 1 GB RAM, nVdia G Force	2000.00
2	2	Pentium 4 2.6 GHtz, 512 MB RAM, Intel 911	600.00

The ComputerDetails Table

The store also sells peripheral devices. You have saved the details of these devices in the AddOnDetails table, as shown in the following figure.

	AddOnID	AddOnDescription	Price
1	1	Creative Speakers, Joystick, LCD Screen	200.00
2	2	Sony Speakers, LCD Screen	160.00
3	3	LCD Screen, Wireless mouse	120.00

The AddOnDetails Table

To identify the total price of a computer with all the combinations of add-on devices, you can use the CROSS JOIN keyword, as shown in the following query:

```
SELECT A.CompDescription,
B.AddOnDescription, A.Price + B.Price
AS 'Total Cost' FROM ComputerDetails A
CROSS JOIN AddOnDetails B
```

The preceding query combines the records of both the tables to display the total price of a computer with all the possible combinations, as shown in the following figure.

	CompDescription	AddOnDescription	Total Cost
1	Pentium D 3 GHtz, 1 GB RAM, nVdia G Force	Creative Speakers, Joystick, LCD Screen	2200.00
2	Pentium D 3 GHtz, 1 GB RAM, nVdia G Force	Sony Speakers, LCD Screen	2160.00
3	Pentium D 3 GHtz, 1 GB RAM, nVdia G Force	LCD Screen, Wireless mouse	2120.00
4	Pentium 4 2.6 GHtz, 512 MB RAM, Intel 911	Creative Speakers, Joystick, LCD Screen	800.00
5	Pentium 4 2.6 GHtz, 512 MB RAM, Intel 911	Sony Speakers, LCD Screen	760.00
6	Pentium 4 2.6 GHtz, 512 MB RAM, Intel 911	LCD Screen, Wireless mouse	720.00

The Output Derived After Executing the Query

Using an Equi Join

An *equi join* is the same as an inner join and joins tables with the help of a foreign key. However, in an equi join, only the equality operator is used to specify the join condition, whereas you can use conditional operators to specify the join condition in an inner join.

Consider an example where you apply an equi join between the EmployeeDepartmentHistory table, the Employee table, and the Department table by using a common column, EmployeeID. To perform this task, you can use the following query:

```
SELECT * FROM HumanResources.EmployeeDepartmentHistory d JOIN HumanResources.Employee e ON d.EmployeeID = e.EmployeeID JOIN HumanResources.Department p ON p.DepartmentID = d.DepartmentID
```

The output of the preceding query displays the EmployeeID column from all the tables, as shown in the following figure.

	EmployeeID	DepartmentID	ShiftID	StartDate	EndDate	ModifiedDate	EmployeeID
1	1	7	1	1996-07-31 00:00:00.000	NULL	1996-07-30 00:00:00.000	1
2	2	4	1	1997-02-26 00:00:00.000	NULL	1997-02-25 00:00:00.000	2
3	3	1	1	1997-12-12 00:00:00.000	NULL	1997-12-11 00:00:00.000	3
4	4	1	1	1998-01-05 00:00:00.000	2000-06-30 00:00:00.000	2000-06-28 00:00:00.000	4
5	4	2	1	2000-07-01 00:00:00.000	NULL	2000-06-30 00:00:00.000	4
6	5	2	1	1998-01-11 00:00:00.000	NULL	1998-01-10 00:00:00.000	5
7	6	5	1	1998-01-20 00:00:00.000	1999-08-15 00:00:00.000	1999-08-13 00:00:00.000	6
8	6	4	1	1999-08-16 00:00:00.000	NULL	1999-08-15 00:00:00.000	6
9	7	7	3	1998-01-26 00:00:00.000	NULL	1998-01-25 00:00:00.000	7

The Output Derived After Executing the Query

Using a Self Join

In a *self join*, a table is joined with itself. As a result, one row in a table correlates with other rows in the same table. In a self join, a table name is used twice in the query. Therefore, to differentiate between the two instances of a single table, the table is given two alias names.

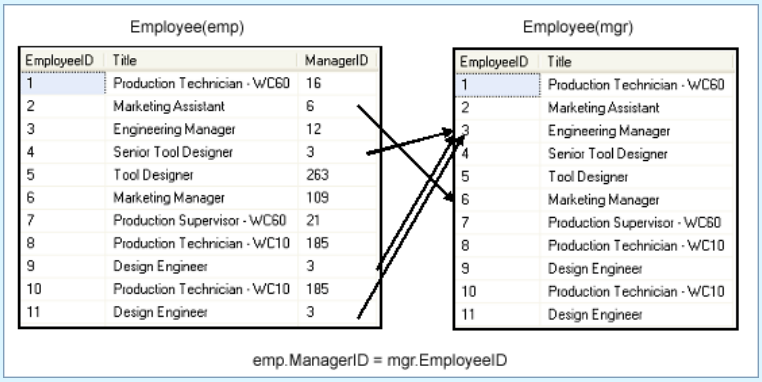
Consider the example of the Employee table. The Employee table contains records as shown in the following figure.

EmployeeID	Title	ManagerID
1	Production Technician - WC60	16
2	Marketing Assistant	6
3	Engineering Manager	12
4	Senior Tool Designer	3
5	Tool Designer	263
6	Marketing Manager	109
7	Production Supervisor - WC60	21
8	Production Technician - WC10	185
9	Design Engineer	3
10	Production Technician - WC10	185

The Employee Table

You want to display the employee details along with their manager details, such as manager ID and manager designation. However, in the preceding table, designation of the manager is not displayed. In this case, the manager is also an employee. Therefore, the designation of the manager can be retrieved from the title column, which stores the designation of an employee. Therefore, you need to join the table with itself to obtain the required result.

To perform a self join, you need to divide the physical table into two logical tables, as shown in the following figure.



The Physical Table Divided into Logical Tables

In the preceding figure, the Employee (emp) table represents the EmployeeID, title, and ManagerID of the employee, whereas the Employee (mgr) table represents the EmployeeID and title of the managers. Using the preceding figure, you can easily identify the designations of the managers.

The following query joins the Employee table with itself to display the employee IDs and the designations of all the employees with the employee IDs and designations of their managers:

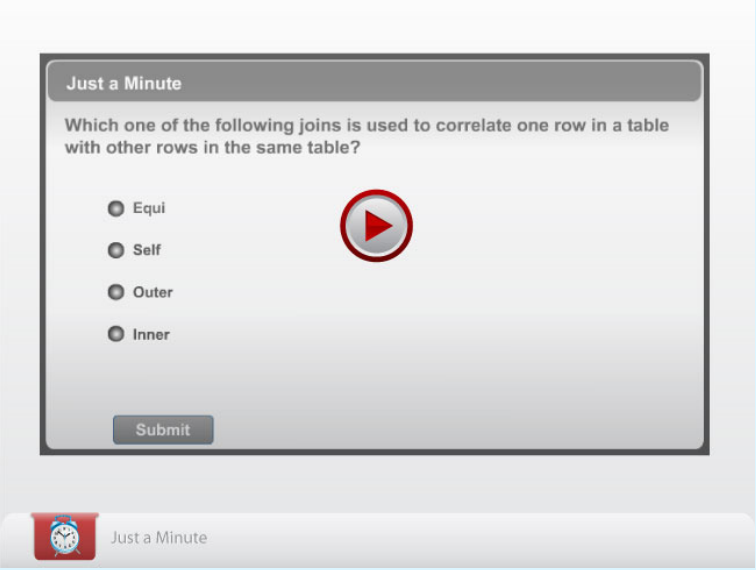
```
SELECT emp.EmployeeID, emp.Title AS
Employee_Designation, emp.ManagerID,
mgr.Title AS Manager_Designation FROM
HumanResources.Employee emp,
HumanResources.Employee mgr
 WHERE emp.ManagerID = mgr.EmployeeID
```

The following figure shows the output of the preceding query.

	EmployeeID	Employee_Designation	ManagerID	Manager_Designation
1	4	Senior Tool Designer	3	Engineering Manager
2	9	Design Engineer	3	Engineering Manager
3	11	Design Engineer	3	Engineering Manager
4	158	Research and Development Manager	3	Engineering Manager
5	263	Senior Tool Designer	3	Engineering Manager
6	267	Senior Design Engineer	3	Engineering Manager
7	270	Design Engineer	3	Engineering Manager
8	271	Marketing Specialist	6	Marketing Manager
9	272	Marketing Assistant	6	Marketing Manager
10	269	Marketing Assistant	6	Marketing Manager
11	203	Marketing Specialist	6	Marketing Manager
12	2	Marketing Assistant	6	Marketing Manager

The Output Derived After Executing the Query

Activity 3.1: Using Joins



Querying Data by Using Subqueries

While querying data from multiple tables, you might need to use the result of one query as an input for the condition of another query. For example, in the AdventureWorks database, you need to view the designation of all the employees who earn more than the average salary. For this, you have to perform two steps. First, you have to find the average salary. Second, you have to find the employees who earn more than the average salary. In such cases, you can combine more than one statement in a *subquery*.

A *subquery* is an SQL statement that is used within another SQL statement. Subqueries are nested inside the WHERE or HAVING clause of the SELECT, INSERT, UPDATE, and DELETE statements. The query that represents the parent query is called the outer query, and the query that represents the subquery is called the*inner query*. The database engine executes the inner query first and returns the result to the outer query to calculate the result set.

For example, the EmployeeDetails table contains five columns: EmployeeID, EmpName, Designation, Salary, and DeptNo. The following figure represents the EmployeeDetails table.

	EmployeeID	EmpName	Designation	Salary	DeptNo
1	1	John	Executive	25000	10
2	2	Smith	Manager	45000	10
3	3	Robert	Clerk	10000	10
4	4	Jack	Clerk	12000	20
5	5	Sylvia	Executive	32000	20

The EmployeeDetails Table

You want to find all the employees who have the same designation as John. Notice that the designation of John is not known in advance. Therefore, first you have to execute the following query to find the designation of John:

```
SELECT Designation FROM
EmployeeDetails
 WHERE EmpName = 'John'
```

Consider that the output of the preceding query is ? Executive?. Then, you have to execute the following query to get the desired output:

```
SELECT * from EmployeeDetails
 WHERE Designation = 'Executive'
```

Instead of executing two queries, you can combine these two queries to get the desired output, as shown in the following query:

```
SELECT * FROM EmployeeDetails
 WHERE Designation = (SELECT
Designation FROM EmployeeDetails
 WHERE EmpName = 'John')
```

In the preceding query, the inner query is executed first and returns the designation of John to the outer query. Then, the outer query is executed using the return value

and displays the desired output, as shown in the following figure.

	EmployeeID	EmpName	Designation	Salary	DeptNo
1	1	John	Executive	25000	10
2	5	Sylvia	Executive	32000	20

The Output Derived After Executing the Query

Depending on the output generated by the subquery and the purpose for which it is to be used in the outer query, you can use different keywords, operators, and functions in the subqueries.

Using the IN and EXISTS Keywords

A subquery returns values that are used by the outer query. A subquery can return one or more values. Depending on the requirement, you can use these values in different ways in the outer query.

For example, in the AdventureWorks database, you need to display the department name for an employee whose EmployeeID is 46. To perform this task, you can use the following query:

```
SELECT Name FROM
HumanResources.Department
 WHERE DepartmentID =
 (SELECT DepartmentID FROM
HumanResources.EmployeeDepartmentHisto
ry
 WHERE EmployeeID = 46 AND EndDate IS
NULL)
```

In the preceding query, the inner subquery returns the DepartmentID column of the employee with EmployeeID as 46. Using this DepartmentID, the outer query returns the name of the department from the Department table. The EndDate is NULL signifying that you need to extract the ID of the department where the employee is currently working.

In the preceding query, the subquery returns a single value. However, at times, you need to return more than one value from the subquery. In addition, you might need to use a subquery only to check the existence of some records and based on that you need to execute the outer query.

You can specify different kinds of conditions on subqueries by using the following keywords:

- IN
- EXISTS

Using the IN Keyword

If a subquery returns more than one value, you might need to match a column value with any of the values in the list returned by the inner query. To perform this task, you need to use the IN keyword.

The syntax for using the IN keyword is:

```
SELECT column, column [,column]
 FROM table_name
 WHERE column [ NOT ] IN
 ( SELECT column FROM table_name
[WHERE conditional_expression])
```

For example, you need to retrieve the EmployeeID attribute of all the employees who live in Bothell from the EmployeeAddress table. To perform this task, you need to use a query to obtain the AddressID of all the addresses that contain the word Bothell. You can then obtain the EmployeeID from the Employee table where the AddressID matches any of the AddressIDs returned by the previous query.

To perform this task, you can use the following query:

```
SELECT EmployeeID FROM
HumanResources.EmployeeAddress WHERE
AddressID IN (SELECT AddressID FROM
Person.Address WHERE City = 'Bothell')
```

The following figure shows the output of the preceding query.

	EmployeeID
1	67
2	179
3	288
4	5
5	41
6	167
7	181
8	35
9	222
10	164
11	72
12	248

The Output Derived After Executing the Query

Using the EXISTS Keyword

You can also use a subquery to check if a set of records exists. For this, you need to use the EXISTS clause with a subquery. The EXISTS keyword always returns a TRUE or FALSE value.

The EXISTS clause checks for the existence of rows according to the condition specified in the inner query and passes the existence status to the outer query. The inner query in this case returns the result set in the form of a table. Therefore, such queries are also called table-valued subqueries. The EXISTS keyword returns a TRUE value if the result of the inner query contains any row.

The query introduced with the EXISTS keyword differs from other queries. The EXISTS keyword is not preceded by any column name, constant, or other expression; and it contains an asterisk (*) in the SELECT list of the inner query.

The syntax for using the EXISTS keyword in the SELECT statement is:

```
SELECT column, column [,column]
 FROM table_name
 WHERE EXISTS ( SELECT column FROM
table_name                    [WHERE
conditional_expression] )
```

For example, the users of AdventureWorks, Inc. need a list containing the employee ID and title of all the employees who have worked in the marketing department at any point in time. The department ID of the marketing department is 4.

To generate the required list, you can write the following query by using the EXISTS keyword:

```
SELECT    EmployeeID,    Title    FROM
HumanResources.Employee
 WHERE EXISTS
 (SELECT            *            FROM
HumanResources.EmployeeDepartmentHisto
ry      WHERE      EmployeeID      =
HumanResources.Employee.EmployeeID  AND
DepartmentID = 4)
```

The following figure displays the output generated by the preceding query.

	EmployeeID	Title
1	2	Marketing Assistant
2	6	Marketing Manager
3	46	Marketing Specialist
4	106	Marketing Specialist
5	119	Marketing Specialist
6	203	Marketing Specialist
7	269	Marketing Assistant
8	271	Marketing Specialist
9	272	Marketing Assistant
10	274	Purchasing Manager

The Output Derived After Executing the Query

A subquery must be enclosed within parentheses and cannot use the ORDER BY or the COMPUTE BY clause.

Using Modified Comparison Operators

While using subqueries, you can use the =, >, and < comparison operators to create a condition that checks the value returned by the subquery. When a subquery returns more than one value, you might need to apply the operators to all the values returned by the subquery. To perform this task, you can modify the comparison operators in the subquery. SQL Server provides the ALL and ANY keywords that can be used to modify the existing comparison operators.

The ALL keyword returns a TRUE value if all the values that are retrieved by the subquery satisfy the comparison operator. It returns a FALSE value if only some values satisfy the comparison operator or if the subquery does not return any rows to the outer statement.

The ANY keyword returns a TRUE value if any value that is retrieved by the subquery satisfies the comparison operator. It returns a FALSE value if no values in the subquery satisfy the comparison operator or if the subquery does not return any rows to the outer statement.

The following table shows the operators that can be used with the ALL and ANY keywords.

Operator	***Description***
>ALL	*Means greater than the maximum value in the list. The expression | column_name >ALL (10, 20, 30) means 'greater than 30'.*
>ANY	*Means greater than the minimum value in the list. The expression | column_name >ANY (10, 20, 30) means 'greater than 10'.*
=ANY	*Means any of the values in the list. It acts in the same way as the IN clause. The expression | column_name =ANY (10, 20, 30) means 'equal to either 10 or 20 or 30'.*
<>ANY	*Means not equal to any value in the list. The expression | column_name <>ANY (10, 20, 30) means 'not equal to 10 or 20 or 30'.*
<>ALL	*Means not equal to all the values in the list. It acts in the same way as the NOT IN clause. The expression | column_name <>ALL (10, 20, 30) means 'not equal to 10 and 20 and 30'.*

The Operators that can be used with the ALL and ANY Keywords

The following query displays the employee ID and the title of all the employees whose vacation hours are more than the vacation hours of employees designated as

Recruiter:

```
SELECT EmployeeID, Title
 FROM HumanResources.Employee
 WHERE VacationHours >ALL (SELECT
VacationHours
 FROM HumanResources.Employee WHERE
Title ='Recruiter')
```

In the preceding query, the inner query returns the vacation hours of all the employees who are titled as Recruiter. The outer query uses the '>ALL' comparison operator. It retrieves the details of those employees who have vacation hours greater than all the employees titled as Recruiter.

The following figure shows the output of the preceding query.

	EmployeeID	Title
1	3	Engineering Manager
2	7	Production Supervisor - WC60
3	8	Production Technician - WC10
4	10	Production Technician - WC10
5	13	Marketing Manager
6	14	Production Supervisor - WC50
7	15	Production Technician - WC10
8	16	Production Supervisor - WC60
9	17	Production Technician - WC10
10	18	Production Supervisor - WC60
11	19	Production Technician - WC10

The Output Derived After Executing the Query

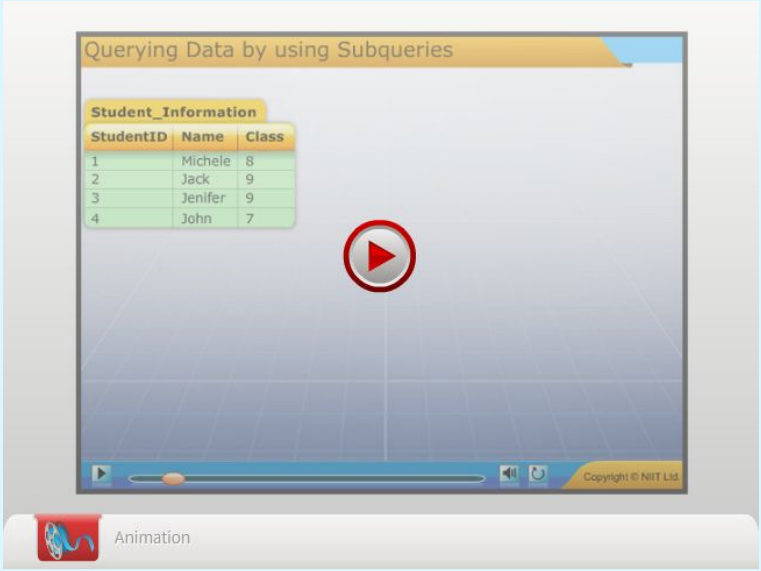
Consider another example of the EmployeeDetails table. You want to display the details of employees whose salary is more than the lowest salary of department having ID 10. To perform this task, you can use the following query:

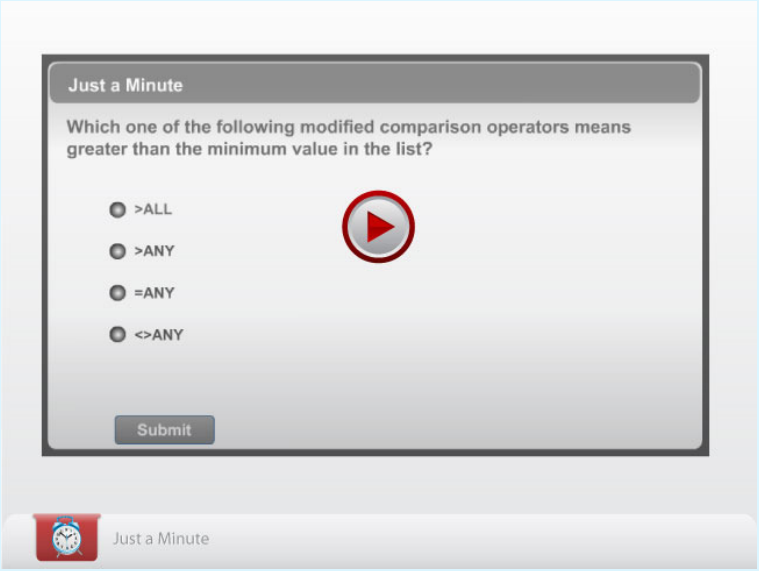
```
SELECT * FROM EmployeeDetails
 WHERE Salary > ANY ( SELECT DISTINCT
Salary FROM EmployeeDetails
 WHERE DeptNo = 10)
```

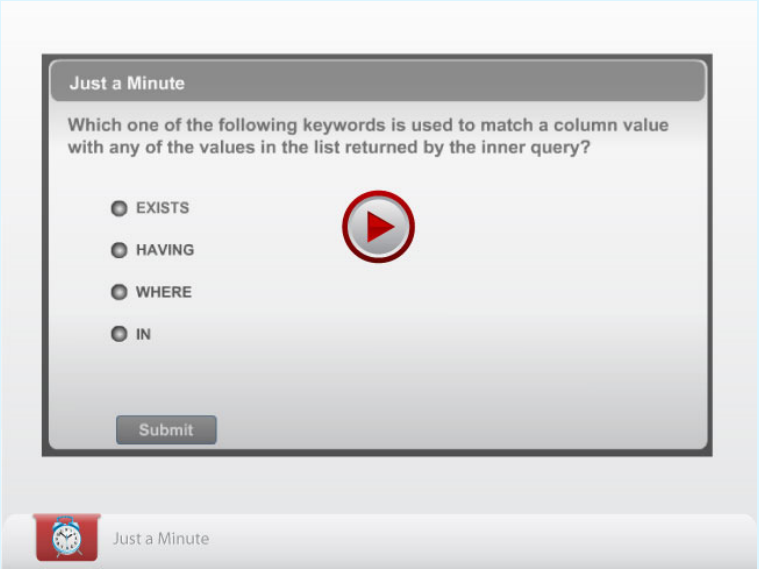
In the preceding query, the ANY operator compares the salary of each employee to each value returned by the subquery. The following figure displays the output of the preceding query.

	EmployeeID	EmpName	Designation	Salary	DeptNo
1	1	John	Executive	25000	10
2	2	Smith	Manager	45000	10
3	4	Jack	Clerk	12000	20
4	5	Sylvia	Executive	32000	20

The Output Derived After Executing the Query







Using Aggregate Functions

While using subqueries, you can also use aggregate functions in the subqueries to generate aggregated values

from the inner query. For example, in a manufacturing organization, the management wants to view the sales records of the items whose sales records are higher than the average sales record of a particular product. For this, you first need to obtain the average sales record of a particular product, and then find all the records whose sales records exceed the average value. For this, you can use aggregate functions inside the subquery.

The following query displays the employee ID of the employees whose vacation hours are greater than the average vacation hours of employees with title as Marketing Assistant:

```
SELECT EmployeeId FROM HumanResources.Employee
 WHERE VacationHours >(SELECT AVG (VacationHours) FROM HumanResources.Employee
 WHERE Title = 'Marketing Assistant')
```

In the preceding query, the inner query returns the average vacation hours of all the employees who are titled as Marketing Assistant. The outer query uses the comparison operator '>' to retrieve the employee ID of all those employees who have vacation hours more than the average vacation hours assigned for a Marketing Assistant.

The following figure shows the output of the preceding query.

	EmployeeId
1	4
2	7
3	8
4	10
5	13
6	14
7	15
8	16
9	17
10	18
11	19

The Output Derived After Executing the Query

Using Nested Subqueries

A subquery can contain one or more subqueries. Subqueries are used when the condition of a query is dependent on the result of another query, which, in turn, is dependent on the result of another subquery.

For example, you need to view the department ID of an employee whose e-mail address is taylor0@adventure-works.com. To perform this task, you can use the following query:

```
SELECT DepartmentID FROM HumanResources.EmployeeDepartmentHistory
 WHERE EmployeeID = /* Level 1 inner query */
 (SELECT EmployeeID FROM HumanResources.Employee
 WHERE ContactID = /* Level 2 inner query */
 (SELECT ContactID FROM Person.Contact WHERE EmailAddress = 'taylor0@adventure-works.com')
 ) AND EndDate IS NULL
```

In the preceding query, two queries are nested within another query. The level 2 inner query returns the contact ID of an employee based on the e-mail address of the employee from the Person table. The value in EndDate column of the HumanResources.EmployeeDepartmentHistory table will be NULL for that department ID in which the employee is currently working.

The level 1 inner query uses this contact ID to search for the employee ID of the employee with the given e-mail address. The main query uses the employee ID returned by the level 1 inner query to search for the department ID from the EmployeeDepartmentHistory table.

The following figure shows output of the preceding query.

	DepartmentID
1	7

The Output Derived After Executing the Query

You can implement subqueries upto 32 levels. However, the number of levels that can be used depends on the memory available on the database server.

Consider another example where you want to display the details of employees whose salary is greater than the highest salary in the Admin department. To perform this task, you need to execute the following query:

```
SELECT * FROM EmployeeDetails
 WHERE Salary > ( SELECT max(Salary) FROM EmployeeDetails
 WHERE DeptNo = ( SELECT DeptNo from DeptDetails
 WHERE DeptName = 'Admin'))
```

In the preceding query, the level 2 inner query returns the department number of the admin department. The level 1 inner query uses this department number to retrieve the highest salary in this department. The main query uses the highest salary returned by the level 1 inner query to display the employee details. The preceding query displays the output as shown in the following figure.

	EmployeeID	EmpName	Designation	Salary	DeptNo
1	2	Smith	Manager	45000	10

The Output Derived After Executing the Query

Using Correlated Subqueries

A correlated subquery can be defined as a query that depends on the outer query for its evaluation. In a normal nested subquery, the inner query executes only once. The main query is executed using the value returned by the inner query. On the other hand, in a correlated subquery, the inner query is driven by the outer query. In this case, the inner query executes once for each row returned by the outer query. For example, if the outer query returns 10 rows, the inner query will be executed 10 times.

In a correlated subquery, the WHERE clause references a table in the FROM clause of the outer SELECT statement. This means that the inner query is evaluated for each row of the table specified in the outer query.

For example, you can use the following query to find the employees who earn the highest salary in their department:

```
SELECT * FROM EmployeeDetails e
 WHERE Salary = (SELECT max(Salary)
FROM EmployeeDetails
 WHERE DeptNo = e.DeptNo)
```

The preceding query can be identified as a correlated subquery since the e.DeptNo column in the WHERE clause of the inner query is referencing the table specified in the FROM clause of the outer SELECT statement. The preceding query will display the output as shown in the following figure.

	EmployeeID	EmpName	Designation	Salary	DeptNo
1	5	Sylvia	Executive	32000	20
2	2	Smith	Manager	45000	10

The Output Derived After Executing the Query

Consider another example of AdventureWorks database. The following query displays the employee ID, designation, and number of hours spent on vacation for all the employees whose vacation hours are greater than the average vacation hours identified for their title:

```
SELECT EmployeeID, Title,
VacationHours
 FROM HumanResources.Employee e1 WHERE
e1.VacationHours >
 (SELECT AVG(e2.VacationHours)
 FROM HumanResources.Employee e2 WHERE
e1.Title = e2.Title)
```

In the preceding query, the inner query returns the titlewise average vacation hours from the Employee table. The outer query retrieves the employee ID, title, and vacation hours of all the employees whose vacation hours are greater than the average vacation hours retrieved by the inner query. The following figure shows the output of the preceding query.

	EmployeeID	Title	VacationHours
1	216	Accountant	59
2	201	Accounts Payable Specialist	64
3	130	Accounts Receivable Specialist	62
4	176	Application Specialist	74
5	149	Application Specialist	73
6	164	Buyer	59
7	198	Buyer	60
8	231	Buyer	57
9	238	Buyer	58

The Output Derived After Executing the Query

Using the APPLY Operator

Sometimes, you need to combine the result set of two queries such that for each row of the first query, the second query is evaluated to determine if any rows are returned. For example, you need to know the dividends paid to each employee of the organization. The dividends are required to be paid on the basis of the salary of the employees. The Employee table contains the records of all the employees. The Pay table contains the salary details of each employee. To display the required data, you need to retrieve employee details from the Employee table. For each row of the Employee table, you need to calculate the dividend based on salary from the Pay table. Therefore, you need to combine the result sets of two queries to display the required data. SQL Server provides you with the *APPLY operator* to perform this task in a single SQL statement.

There are two types of APPLY operators supported by SQL Server. These are:

- CROSS APPLY
- OUTER APPLY

The syntax for using the APPLY operator is:

```
left_table_expression {CROSS | OUTER}
APPLY right_table_expression
```

where,

`left_table_expression` represents the outer result set.

`right_table_expression` represents the inner result set, which is evaluated for each record of the outer result set.

CROSS APPLY

The CROSS APPLY operator returns only those rows from the outer result set that matches with the inner result set.

For example, customers can deposit their money in a bank. The information regarding the deposits made by the customers is stored in the Depositor table, as shown in the following figure.

	Customer_name	Acc_num
1	Nora	101
2	Robin	103
3	James	107
4	Jennifer	109

The Depositor Table

The bank also grants loans to the customers for their personal use. This information is stored in the Borrower table, as shown in the following figure.

	Customer_name	Loan_num
1	Nora	301
2	Robin	305
3	James	306
4	Jenne	308

The Borrower Table

You want to view the name, account number, and loan number of the customers who are borrowers as well as depositors. The following query uses the CROSS APPLY operator on the Depositor table and the Borrower table to view the required result:

```
SELECT d.Customer_name, d.Acc_num,
br.Loan_num FROM Depositor d
 CROSS APPLY
 (SELECT * FROM Borrower b WHERE
d.Customer_name = b.Customer_name) br
```

The following figure displays the output of the preceding query.

	Customer_name	Acc_num	Loan_num
1	Nora	101	301
2	Robin	103	305
3	James	107	306

The Output Derived After Executing the Query

In the preceding figure, the record of Jenne is not displayed as it does not match any row in the outer result set.

OUTER APPLY

The OUTER APPLY operator returns all rows from the outer result set even if the corresponding row is not found in the inner result set. Therefore, the final result set obtained after performing the OUTER APPLY operation contains NULL values when the inner result set does not return a matching value for each or either of the rows from the outer result set.

For example, you want to view the details of all the customers who are depositors. In addition, you want to view the loan numbers of the customers who are borrowers also. You can view the desired result by using the following query:

```
SELECT d.Customer_name, d.Acc_num,
br.Loan_num FROM Depositor d
 OUTER APPLY
 (SELECT * FROM Borrower b WHERE
d.Customer_name = b.Customer_name)br
```

The following figure displays the output of the preceding query.

	Customer_name	Acc_num	Loan_num
1	Nora	101	301
2	Robin	103	305
3	James	107	306
4	Jennifer	109	NULL

The Output Derived After Executing the Query

In the preceding figure, Jennifer does not have a loan account with the bank. Therefore, NULL is displayed against the loan number for Jennifer in the result set.

Activity 3.2: Using Subqueries

Managing Result Sets

In an organization, data related to different entities is stored in multiple tables. If you want to view data from the multiple tables together, you need to combine or compare the result sets. SQL Server provides the various operators, such as UNION, EXCEPT, and INTERSECT, to combine as well as compare the result sets.

Sometimes, you need to reference the result set of the previous query to manipulate the data in other tables. However, the result set of a query is not stored in a database. Therefore, if you want to use the result set of a query, you need to first store that result set in a table and then use that table to manipulate the data in other tables. This task has been made easy by using CTEs provided by SQL Server.

Combining Result Sets

As a database developer, you often need to create the result sets after combining data from multiple tables. While combining data from multiple tables, you may need to either store only the common records from the combined tables or exclude some records from the result sets based on different criteria.

SQL Server provides you with the UNION operator to combine data from multiple tables. In addition, you can use the EXCEPT and INTERSECT operators to exclude

records from the result sets based on different criteria.
You must follow some basic rules while using the UNION, EXCEPT, and INTERSECT operators to combine the result sets. These rules are:

- The number and the sequence of the columns must be the same in all queries.
- The data types of the columns in all the queries must be compatible.

Using the UNION Operator

The UNION operator is used to combine data of two or more queries into a single result set. The syntax of the UNION operator is:

```
<query_expression>
 UNION [ ALL ]
 <query_expression>
 [ UNION [ ALL ]
 <query_expression>
 [ ...n ] ]
```

where,

`query_expression` is the query expression that returns the data to be combined with the data from another query expression. The definitions of the columns that are part of a UNION operation do not have to be the same, but they must be compatible through implicit conversion.

`UNION` specifies that result sets from multiple queries need to be combined and returned as a single result set.

`ALL` specifies that all the rows need to be incorporated into the result set. This can result in duplicate rows. If it is not specified, duplicate rows are removed.

For example, you can use the following query to view the details of all customers who are either depositors or borrowers with the bank:

```
SELECT Customer_name FROM Depositor
 UNION
 SELECT Customer_name FROM Borrower
```

The following figure displays the output of the preceding query.

	Customer_name
1	James
2	Jenne
3	Jennifer
4	Nora
5	Robin

The Output Derived After Executing the Query

In the preceding figure, there are no duplicate rows in the result set. Now, consider the following query:

```
SELECT Customer_name FROM Depositor
 UNION ALL
 SELECT Customer_name FROM Borrower
```

The following figure displays the output of the preceding query.

	Customer_name
1	Nora
2	Robin
3	James
4	Jennifer
5	Nora
6	Robin
7	James
8	Jenne

The Output Derived After Executing the Query

In the preceding figure, the result set contains all the rows from both the tables. This resulted in duplicate rows in the result set.

Using the EXCEPT and INTERSECT Operators

Sometimes, you need to display the result set after comparing the results of two queries. You can use the EXCEPT and INTERSECT operators to perform this task. The EXCEPT operator compares two result sets and returns the data from the first result set that is not found in the second result set. The number and order of columns should be same in the result sets on which the EXCEPT operator is applied.

However, the INTERSECT operator returns the common rows after comparing two result sets. If the common rows are not found, then NULL value is returned. The syntax for using the INTERSECT and EXCEPT operators is:

```
<query_expression>
 EXCEPT | INTERSECT
 <query_expression>
```

where,

`query_expression` specifies the query that returns the data to be compared with the data from another query expression.

For example, you can use the following query to return the name of those customers who are depositors but not borrowers:

```
SELECT Customer_name FROM Depositor
 EXCEPT
 Select Customer_name FROM Borrower
```

The following figure displays the output of the preceding query.

	Customer_name
1	Jennifer

The Output Derived After Executing the Query

The preceding figure returns the name of the customer who is not a borrower.

Similarly, you can use the following query to find the name of the customers who are depositors and taken loan from the bank as well:

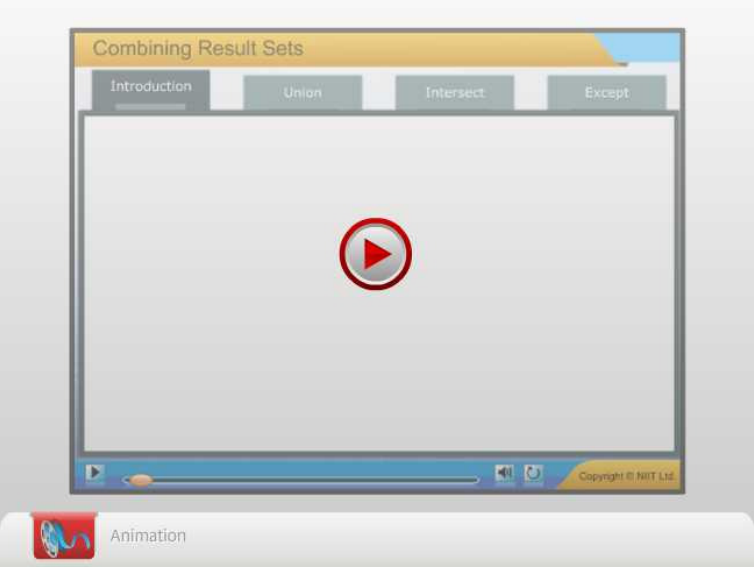
```
SELECT Customer_name FROM Depositor
```

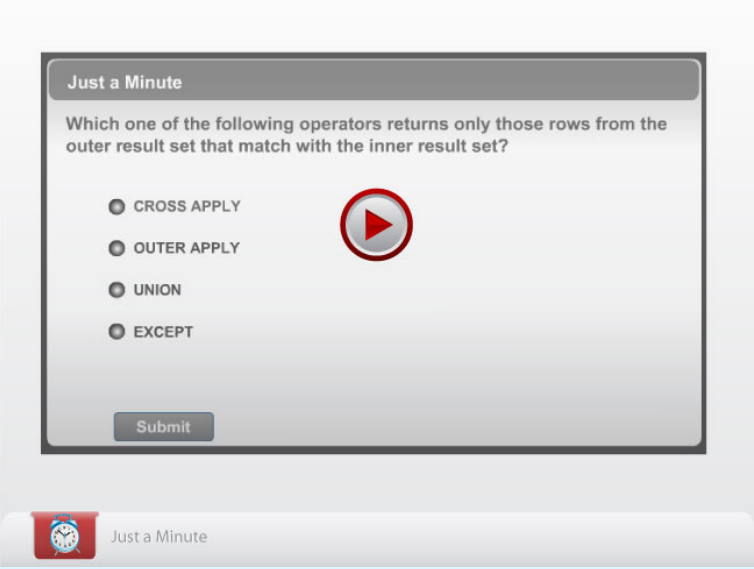
```
INTERSECT
SELECT Customer_name FROM Borrower
```

The following figure displays the output of the preceding query.

	Customer_name
1	James
2	Nora
3	Robin

The Output Derived After Executing the Query





Working with Temporary Result Sets

Sometimes, a user wants to create queries on the result sets that are obtained earlier. The result sets returned after executing the queries are known as temporary result sets. These temporary result sets are not stored in the database and remain in the memory till the execution of the query. After the execution of the query, the table containing the temporary result set is automatically dropped.

However, SQL Server allows you to access the temporary result sets by using CTEs.

A CTE is used to create a temporary named result set. It is defined within the execution scope of a single SQL statement. The syntax for using CTE is:

```
WITH expression_name [(column_name1
[,...n] )]
 AS
 (CTE_Query_defnition)
```

where,

`expression_name` is the name of the CTE.

`(column_name1 [,...n] )` specifies the name of columns included in the temporary result set.

`CTE_Query_defnition` specifies the query whose result set populates the CTE.

For example, you want to prepare a report that displays the top 10 rates of employees. In addition, the report should also display the maximum rate among these top 10 rates. For this, you need to use the CTE, as shown in the following statements:

```
WITH RateCTE(Rate)
 AS
 (
 SELECT TOP 10 Rate = Rate FROM
HumanResources.EmployeePayHistory
 )
 SELECT Rate, Max_Rate = (SELECT max
(Rate) FROM RateCTE)
 FROM RateCTE
```

The following figure displays the output of the preceding statements.

	Rate	Max_Rate
1	12.45	43.2692
2	13.4615	43.2692
3	43.2692	43.2692
4	8.62	43.2692
5	23.72	43.2692
6	29.8462	43.2692
7	25.00	43.2692
8	24.00	43.2692
9	28.75	43.2692
10	37.50	43.2692

The Output Derived After Executing the Statements

Sometimes, you need to reference a CTE multiple times in the same query. Such queries are referred to as recursive CTEs. In a recursive CTE, result sets of more than one query are combined to populate the CTE.

A recursive CTE consists of the following members to produce the required result:

- **Anchor query**: Includes one or more query expressions joined by the UNION ALL, UNION, EXCEPT, or INTERSECT operators. As these

query expressions produce the base result set for the recursive CTE, they are referred to as anchor query.

- **Recursive query**: Includes one or more query expressions joined by the UNION ALL operator that uses the CTE itself to produce the result. It takes the result of the anchor query as an input to produce an output. This output further acts as an input for the recursive query to produce another output. This process continues until an empty result set is produced.

The syntax of recursive CTE is:

```
WITH expression_name [(column_name1 [,...n] )]
 AS
 (
 (CTE_Query_defnition) --anchor query
 UNION ALL
 (Recursive_Query_Expression)
 )
```

where,

`CTE_Query_defnition` represents the anchor query that is used as an input for the recursive query.

`Recursive_Query_Expression` represents the recursive query.

The process of execution of a recursive CTE is:

1. The anchor query is executed first to create the first invocation or base result set (T0).
2. The recursive query is executed with Ti as an input and Ti+1 as an output. Here, i starts from 0.
3. Step 2 is repeated until an empty set is returned.
4. The final result set is displayed. This result set is a UNION ALL of T0 to Tn.

For example, consider the following statements:

```
WITH List_CTE(i)
 AS
 (
 SELECT i = CONVERT(varchar (8000),'Welcome')
 UNION ALL
 SELECT i + 'A' FROM List_CTE WHERE LEN(i) < 10
 )
 SELECT i FROM List_CTE
 ORDER BY i
```

In the preceding statements, the anchor query populates the CTE with the result, Welcome. The recursive query uses this result and produces the output, WelcomeA. This output further acts as an input for the recursive query to produce the result, WelcomeAA. This process continues until the recursive query produces no rows.

The following figure displays the output of the preceding statements.

	i
1	Welcome
2	WelcomeA
3	WelcomeAA
4	WelcomeAAA

The Output Derived After Executing the Statements

Consider another example. The Employee table of the AdventureWorks database stores the employee details along with the information about their respective supervisors. You want to find the hierarchy of employees starting from the top level employee. To accomplish this task, you can execute the following statements:

```
WITH Rec_CTE(LoginID, ManagerID, EmployeeID) AS
 (
 SELECT LoginID, ManagerID, EmployeeID
 FROM HumanResources.Employee
 WHERE ManagerID IS NULL
 UNION ALL
 SELECT e.LoginID, e.ManagerID, e.EmployeeID
 FROM HumanResources.Employee e
 INNER JOIN Rec_CTE d
 ON e.ManagerID = d.EmployeeID
 )
 SELECT *
 FROM Rec_CTE
```

The following steps describe the execution of the preceding statements:

1. The anchor query returns the top level employee where the ManagerID is null, as shown in the following figure.

	LoginID	ManagerID	EmployeeID
1	adventure-works\ken0	NULL	109

The Top Level Employee

2. This result is referenced by the recursive query to find the direct subordinates at the next level, 109. The direct subordinates at the next level are retrieved by performing the JOIN operation between the Employee table and the CTE.
3. The following figure shows the direct subordinates at the next level.

	LoginID	ManagerID	EmployeeID
1	adventure-works\david0	109	6
2	adventure-works\terri0	109	12
3	adventure-works\jean0	109	42
4	adventure-works\laura1	109	140
5	adventure-works\james1	109	148
6	adventure-works\brian3	109	273

The Direct Subordinates at the Next Level

4. Retrieval of direct subordinates at the next level occurs recursively, until an empty result set is returned by the JOIN operation. The final result

set returned by the running query is the union of all result sets generated by the anchor and recursive queries. The following figure displays the final result set of the preceding query.

	LoginID	ManagerID	EmployeeID
1	adventure-works\ken0	NULL	109
2	adventure-works\david0	109	6
3	adventure-works\terri0	109	12
4	adventure-works\jean0	109	42
5	adventure-works\laura1	109	140
6	adventure-works\james1	109	148
7	adventure-works\brian3	109	273
8	adventure-works\stephen0	273	268
9	adventure-works\amy0	273	284
10	adventure-works\syed0	273	288
11	adventure-works\lynn0	288	290
12	adventure-works\jae0	284	285

The Final Result Set Obtained After Executing the Statement

In the preceding figure, the hierarchical representation of employees is displayed.

Working with Derived Tables

Sometimes, obtaining data in a result set is a complicated process as it might include some complex aggregate calculations, involving fields of one or more tables. A simple query might not help in such a case. Therefore, you may need to simplify the process by using temporary tables or views. For example, the SalesOrderDetail table of the AdventureWorks database contains the details of the orders placed by the customers. You want to find out the total quantity ordered and the total orders placed for a particular product. To accomplish this task, you need to first use the following SQL statement to retrieve the total quantity ordered for the product:

```
SELECT ProductID, SUM(OrderQty) AS
TotalQuantityOrdered FROM
Sales.SalesOrderDetail
 GROUP BY ProductId
```

The following figure displays the output of the preceding statement.

	ProductID	TotalQuantityOrdered
1	925	625
2	902	36
3	710	90
4	879	249
5	733	90
6	856	1616
7	756	346
8	779	2394
9	802	190
10	971	322

The Output Derived After Executing the Statement

Now, to view the total orders placed for a particular product, you need to execute the following statement:

```
SELECT ProductID,SUM(OrderQty) AS
TotalSalesOrdersPlaced FROM
Sales.SalesOrderDetail
 GROUP BY ProductId,SalesOrderId
```

The following figure displays the output of the preceding statement.

	ProductID	TotalSalesOrdersPlaced
1	758	6
2	853	2
3	896	3
4	934	1
5	962	1
6	708	1
7	921	1
8	923	1
9	715	1
10	772	1

The Output Derived After Executing the Statement

Now, you want to view the output of both the preceding two queries in a single result set. To perform this task, SQL Server provides you, with the concept of derived tables. The derived tables is a virtual table, created by using a SELECT statement and is given an alias name using the AS clause. It is also referred to as the inline views. It can be referenced as a regular table or a view in the SELECT statement. The derived table has its scope restricted within the query, can be used only within the query, and ceases to exist once the query has finished executing. Unlike temporary tables or views, there is no overhead of creation, deletion, or data insertion.

For example, consider the following statements:

```
SELECT
Table1.Productid,TotalQuantityOrdered,
TotalSalesOrdersPlaced
 FROM
 (SELECT ProductID,SUM(OrderQty) AS
TotalQuantityOrdered FROM
Sales.SalesOrderDetail
 GROUP BY ProductId
 )AS Table1
 INNER JOIN
 (SELECT ProductID,SUM(OrderQty) AS
TotalSalesOrdersPlaced FROM
Sales.SalesOrderDetail
 GROUP BY ProductId ,SalesOrderId
 ) AS Table2
 ON Table1.ProductID=Table2.ProductID
```

The following steps describe the execution of the preceding statements:

1. The FROM clause of the preceding statement

creates two derived tables named, Table1 and Table2.

2. The derived table, Table1, selects the total number of orders placed for each product from the Sales.SalesOrderDetail table.
3. The derived table, Table2, selects the total sales orders placed for each product from the Sales.SalesOrderDetail table.
4. Finally, an inner join is used to obtain data from both the derived tables into a common result set.

The following figure displays the output of the preceding statements.

	Productid	TotalQuantityOrdered	TotalSalesOrdersPlaced
1	825	850	2
2	859	3464	1
3	708	6532	1
4	792	946	2
5	859	3464	1
6	707	6266	1
7	877	3319	1
8	777	678	5
9	759	618	5
10	860	1276	1

The Output Derived After Executing the Statement

Activity 3.3: Using CTE

Activity 3.4: Using Derived Tables

Summary

In this chapter, you learned that:

- ❑ Joins and subqueries are used to retrieve data from multiple tables.
- ❑ An inner join combines records from multiple tables by using a comparison operator on a common column.
- ❑ A left outer join returns all the rows from the left table and the matching rows from the right table.
- ❑ A right outer join returns all the rows from the right table and the matching rows from the left table.
- ❑ A full outer join returns all the matching and non-matching rows from both the tables on which the join is applied.
- ❑ A cross join returns each row from the first table joined with each row from the second table.
- ❑ In an equi join, only the equality operator is used to specify the join condition.
- ❑ A self join correlates one row in a table with other rows in the same table.
- ❑ The IN clause in a subquery returns zero or more values.
- ❑ The EXISTS clause in a subquery returns data in terms of a TRUE or FALSE value.
- ❑ The ALL and ANY keywords are used in a subquery to modify the existing comparison operator.
- ❑ Aggregate functions can also be used in subqueries to generate aggregated values from the inner query.
- ❑ Subqueries that contain one or more queries are specified as nested subqueries.
- ❑ A correlated subquery can be defined as a query that depends on the outer query for its evaluation.
- ❑ SQL Server provides the APPLY operator that allows you to combine the result sets retrieved from table expressions.
- ❑ There are two types of APPLY operators supported by SQL Server. These are:
  - CROSS APPLY
  - OUTER APPLY
- ❑ The CROSS APPLY operator returns only those rows from the outer result set that match with the inner result set.
- ❑ The OUTER APPLY operator returns all rows from the outer result set even if the corresponding row is not found in the inner result set.
- ❑ The UNION operator is used to combine the data of two or more queries into a single result set.
- ❑ The EXCEPT operator compares two result sets and returns the data from the first result set that is not found in the second result set.
- ❑ The INTERSECT operator returns the common rows after comparing two result sets. If the common rows are not found, then NULL value is returned.
- ❑ A CTE is used to create a temporary named result set.
- ❑ In a recursive CTE, result sets of more than one query are combined to populate the CTE.
- ❑ A recursive CTE consists of the following members to produce the required result:
  - Anchor query
  - Recursive query
- ❑ A derived table is a virtual table, which is created by using the SELECT statement with an alias in the AS clause.
- ❑ Derived table acts as a regular table or a view.

Reference Reading

Querying Data by Using Joins

Reference Reading: Books	*Reference Reading: URLs*
Sams Teach Yourself SQL in 24 Hours by Ryan Stephens and Ron Plew	*http://msdn.microsoft.com/en-us/library/ms191472.aspx*

Querying Data by Using Subqueries

Reference Reading: Books	*Reference Reading: URLs*
Microsoft SQL Server 2012 Bible by Adam Jorgensen, Jorge Segarra, and Patrick LeBlanc	*http://msdn.microsoft.com/en-us/library/ms189575%28v=SQL.105%29.aspx*

Managing Result Sets

Reference Reading: Books	*Reference Reading: URLs*
Beginning SQL Server 2012 for Developers by Robin Dewson	*http://msdn.microsoft.com/en-us/library/ms190766.aspx*

Chapter 4

Managing Databases and Tables

As a database developer, you are responsible for creating and managing databases and tables. While creating tables, it is important for you to maintain *data integrity*. This implies that data in tables is accurate, consistent, and reliable. SQL Server provides various checks that you can apply on tables to enforce data integrity.

This chapter introduces different types of system databases. It explains how to create and drop user-defined databases. Further, it explains how to create and manage user-defined tables by using DDL statements. In addition, the chapter focuses on various checks and rules that you can apply to tables to ensure data integrity.

Objectives

In this chapter, you will learn to:

- Manage databases
- Manage tables

Managing Databases

A database is a collection of tables and objects such as views, indexes, stored procedures, and triggers. The data stored in a database may be related to a process, such as an inventory or a payroll.

As a database developer, you might need to create databases to store information. At times, you might also need to delete a database, if it is not required. Therefore, it is essential to know how to create and delete a database.

SQL Server supports many databases and contains some standard system databases. Before creating a database, it is important to identify the system databases supported by SQL Server and their importance.

A view is a virtual table that provides access to a subset of columns from one or more tables.

An index is an internal table structure that SQL Server uses to provide quick access to the rows of a table based on the values of one or more columns.

A stored procedure is a collection or batch of T-SQL statements and control-of-flow language that is stored under one name and executed as a single unit.

A trigger is a block of code that constitutes a set of T-SQL statements. These statements are activated in response to certain actions.

Identifying System Databases in SQL Server

System databases are the standard databases that exist in every instance of SQL Server. These databases contain a specific set of tables that are used to store server-specific configurations and templates for other databases. In addition, these databases contain a temporary storage area required to query the database.

SQL Server contains the following system databases:

- master
- tempdb
- model
- msdb
- Resource

The master Database

The master database consists of system tables that keep track of the server installation as a whole and all the other databases. It records all the server-specific configuration information, including authorized users, databases, system configuration settings, and remote servers. In addition, it records the instance-wide metadata, such as logon accounts, endpoints, and system configuration settings.

The master database contains critical data that controls the SQL Server operations. It is advisable not to give any permission to users on the master database. It is also important to update the backups of the master database to reflect the changes that take place in the database as the master database records the existence of all other databases and the location of those database files.

The master database also stores the initialization information of SQL Server. Therefore, if the master database is unavailable, the SQL Server database engine will not be started.

The query window of SQL Server Management Studio defaults to the master database context. Any queries executed from the query window will execute in the master database unless you change the context.

The tempdb Database

The tempdb database is a temporary database that holds all temporary tables and stored procedures. It is automatically used by the server to resolve large or nested queries or to sort data before displaying results to the user.

All the temporary tables and results generated by the GROUP BY, ORDER BY, and DISTINCT clauses are stored in the tempdb database. The tempdb database is re-created every time SQL Server is started so that the system always starts with a clean copy of the database. Temporary tables and stored procedures are dropped automatically on disconnect or when the system is shut down. Backup and restore operations are not allowed on tempdb database. Therefore, you should not save any database object in the tempdb database because this database is recreated every

time SQL Server starts. This results in losing the data you saved earlier.

Stored procedures will be discussed later in Chapter 7. Endpoints will be discussed later in Chapter 10.

The model Database

The model database acts as a template or a prototype for new databases. Whenever a database is created, the contents of the model database are copied to the new database.

If you modify the model database, all databases created after the modification will inherit those changes. The changes include setting permissions or database options, or adding objects such as tables, functions, or stored procedures. For example, if you want every new database to contain a particular database object, you can add the object to the model database. After this, whenever you create a new database, the object will be added to the database.

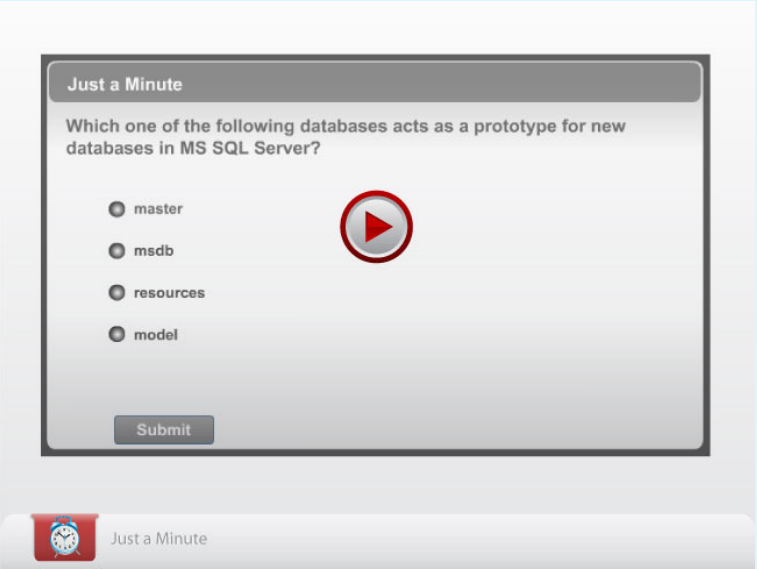
The msdb Database

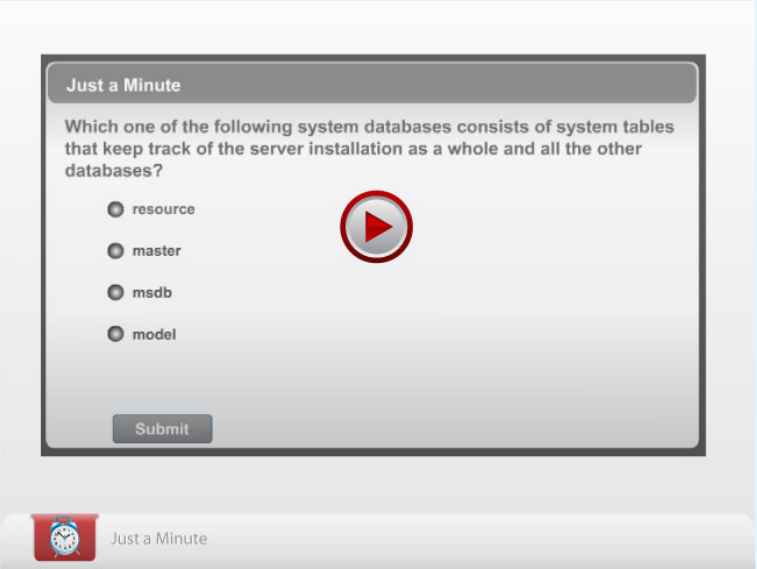
The msdb database supports SQL Server Agent, which is a tool to schedule periodic activities of SQL Server, such as backup and database mailing. SQL Server Agent is used to schedule a job to run in response to a specific event or on demand. For example, if you want to back up all the company servers every weekday, you can automate this task. Schedule the backup to run after 12.00 P.M., Monday through Friday. If the backup encounters a problem, SQL Server Agent can record the event and notify you. The msdb database contains task scheduling, exception handling, alert management, and system operator information needed for the SQL Executive Service. The msdb database contains a few system-defined tables that are specific to the database.

As a database developer, you can query this database to retrieve information on alerts, exceptions, and schedules. For example, you can query this database to know the schedule for the next backup and to know the history of previously scheduled backups. You can also query this database to know how many database e-mail messages have been sent to the administrator.

The Resource Database

The Resource database is a read-only, hidden system database that contains all the system objects, such as system-defined procedures and views, included with SQL Server. It does not contain user data or user metadata. It makes upgrading to a new version of SQL Server easier and faster. By design, it is not visible in the Object Explorer window of SQL Server Management Studio. It also does not have an entry in master.sys.databases.





Identifying the Database Files

SQL Server maps a database over a set of operating-system files. Each database is stored as a set of files on the hard disk of the computer. These files include:

- **Primary data file**: The primary data file contains database objects. It can be used for the system tables and objects. It is the starting point of the database and points to other files in the database. Every database has one primary data file. It has a .mdf extension.
- **Secondary data file**: The secondary data file is used to store user-defined database objects. Very large databases may need multiple secondary data files spread across multiple disks. Databases need not have secondary data files if the primary data file is large enough to hold all the data in the database. The secondary data file has a .ndf extension.
- **Transaction log file**: The transaction log file records all modifications that have occurred in the

database and the transactions that caused those modifications. The transaction log files hold all the transaction information and can be used to recover a database. At least one transaction log file must exist for a database. However, there can be more than one transaction log files. The minimum size of a transaction log file is 512KB. The size of the transaction log file should be 25 - 40 percent of the size of the database. The log files have a .ldf extension.

A database must consist of a primary data file and one transaction log file. In SQL Server, the locations of all the files in a database are recorded in the primary data file of the database and in the master database.

What are Filegroups?

Database files are grouped together in *filegroups* for allocation and administration purposes. A filegroup is a collection of files. A database comprises a primary filegroup and any user-defined filegroup. A primary filegroup contains the primary data file and any other files that are not put into any other filegroup. It also contains the system tables. When objects are created in a database without specifying the filegroup, they are assigned to the default filegroup. Only one filegroup in a database can be the default filegroup.

A user-defined filegroup is a filegroup that is created by users. You can create filegroups to distribute data amongst more than one filegroups to improve the performance of a database.

Creating a User-Defined Database

In addition to system databases, SQL Server also contains user-defined databases where users store and manage their information. When a user creates a database, it is stored as a set of files on the hard disk of the computer.

To create a user-defined database, you can use the CREATE DATABASE statement. The syntax of the CREATE DATABASE statement is:

```
CREATE DATABASE database_name
 [ ON [ PRIMARY ] [ < filespec >]]
 [ , <filegroup> [ ,...n ] ]
 [ LOG ON [ < filespec > ]]
 < filespec > ::=
 ( [ NAME = logical_file_name , ]
 FILENAME = 'os_file_name'
 [ , SIZE = size ]
 [ , MAXSIZE = { max_size |
UNLIMITED } ]
 [ , FILEGROWTH = growth_increment ] )
[ ,...n ]
 }
 <filegroup> ::=
 {
 FILEGROUP filegroup_name [ CONTAINS
FILESTREAM ] [ DEFAULT ]
 <filespec> [ ,...n ]
 }
```

where,

`database_name` is the name of the new database.

`ON` specifies the disk files used to store the data portion of the database (data files).

`PRIMARY` specifies the associated `<filespec>` list that defines files in the primary filegroup.

`LOG ON` specifies the disk files used to store the log files.

`NAME=logical_file_name` specifies the logical name for the file.

`FILENAME=os_file_name` specifies the operating-system file name for the file.

`SIZE=size` specifies the initial size of the file defined in the `<filespec>`list.

`MAXSIZE=max_size` specifies the maximum size to which the file defined in the `<filespec>` list can grow.

`FILEGROWTH=growth_increment` specifies the growth increment of the file defined in the `<filespec>` list. The `FILEGROWTH` setting for a file cannot exceed the MAXSIZE setting.

`filegroup_name` specifies the logical name of the filegroup. It must be unique in the database and cannot be the system-provided name.

`CONTAINS FILESTREAM` specifies that the filegroup stores the FILESTREAM data in the file system.

`DEFAULT` specifies that it is the default filegroup in the database.

To create a database, you must be a member of the dbcreator server role. In addition, you must have the CREATE DATABASE, CREATE ANY DATABASE, or ALTER ANY DATABASE permissions.

The following statement creates a database named Personnel to store the data related to all the employees:

```
CREATE DATABASE Personnel
```

The preceding statement creates a database named Personnel in the C:\Program Files\Microsoft SQL Server \MSSQL10.MSSQLSERVER\MSSQL\DATA folder. The data file name of the database is Personnel.mdf and the log file name is Personnel_log.ldf.

Specifying FileGroups While Creating Databases

When you specify only the name of the database while creating a database, it is created with the default filegroup. However, you can also specify the name of the filegroup to be used for a database while creating it. For example, consider the following statements:

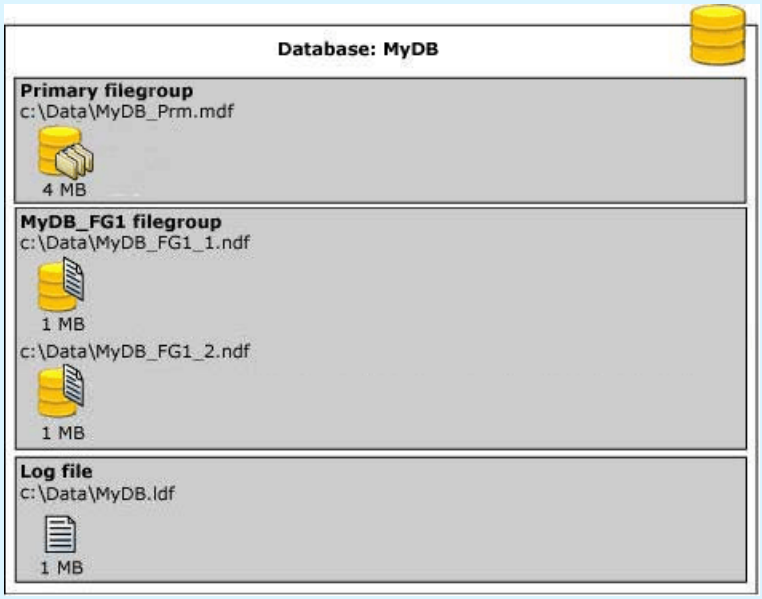
```
USE master
```

```
GO
CREATE DATABASE MyDB
ON PRIMARY
( NAME='MyDB_Primary',
FILENAME=
'c:\Data\MyDB_Prm.mdf',
SIZE=4MB,
MAXSIZE=10MB,
FILEGROWTH=1MB),
FILEGROUP MyDB_FG1
( NAME = 'MyDB_FG1_Dat1',
FILENAME =
'c:\Data\MyDB_FG1_1.ndf',
SIZE = 1MB,
MAXSIZE=10MB,
FILEGROWTH=1MB),
( NAME = 'MyDB_FG1_Dat2',
FILENAME =
'c:\Data\MyDB_FG1_2.ndf',
SIZE = 1MB,
MAXSIZE=10MB,
FILEGROWTH=1MB)
LOG ON
( NAME='MyDB_log',
FILENAME =
'c:\Data\MyDB.ldf',
SIZE=1MB,
MAXSIZE=10MB,
FILEGROWTH=1MB)
GO
```

The preceding statements create a database MyDB where the primary data file is stored in the primary filegroup, which is the default filegroup. The two secondary data files are stored in the user-defined filegroup named MyDB_FG1. It also creates a log file by the name MyDB.ldf.

NOTE *The Data folder must exist in the C drive for the preceding code to execute successfully.*

The following figure represents the creation of the primary, log, and secondary files of the MyDB database.



The MyDB Database

You can change the default filegroup by using the following statement:

```
ALTER DATABASE MyDB
 MODIFY FILEGROUP MyDB_FG1 DEFAULT
 GO
```

The preceding statement makes MyDB_FG1 as the default filegroup for the MyDB database.

Task 4.1: Creating a Database Using the Object Explorer Window

NOTE *After a database is created, you may also need to view the details of the database, such as database size, database owner, date of creation, and compatibility level. For this purpose, you can use the sp_helpdb statement. The syntax of the sp_helpdb statement is:*

```
sp_helpdb [database name]
```

Creating FILESTREAM-Enabled Databases

At times, you need to store unstructured data such as videos, graphic files, sound clips, Word documents, or Excel spreadsheets in a database. This unstructured data is called a Binary Large Object (BLOB). SQL Server provides the FILESTREAM feature that enables you to store and manage BLOBs effectively.

The FILESTREAM feature allows BLOBs to be stored directly in the Windows file system. This feature integrates the SQL Server Database Engine with New Technology File System (NTFS). The BLOBs are stored in the NTFS file system and the SQL Server database engine manages the link between the column declared with the FILESTREAM feature and the actual file located in the NTFS.

NTFS is the file system used by the Windows operating system to store and retrieve files from a hard disk.

To use the FILESTREAM feature, a database needs to contain a FILESTREAM filegroup. In addition, you need to create a table that contains a varbinary(max) column along with the FILESTREAM attribute. Setting a column with the FILESTREAM attribute causes the database engine to store all the data for that column in the Windows file system, and not in the database file.

For a database to contain the FILESTREAM filegroup, the FILESTREAM feature of the SQL Server instance, on which the database is created, must be enabled. You can enable the FILESTREAM feature of an SQL Server instance during its installation. In addition, you can enable the FILESTREAM feature of an SQL Server instance by using the SQL Server Configuration Manager. Once the FILESTREAM feature is enabled for an instance of SQL Server, you need to configure the FILESTREAM access level.

The FILESTREAM access level specifies the accessibility of the FILESTREAM data on an instance of SQL Server. The following table describes the possible values of access levels and their accessibility.

Access Levels	*Accessibility*
0	*FILESTREAM data access is disabled. This is the default value.*
1	*FILESTREAM data is enabled only for T-SQL access.*
2	*FILESTREAM data is enabled only for T-SQL and local file system access.*
3	*FILESTREAM data is enabled for T-SQL, local file system access, and remote file system access.*

The Access Levels and the Accessibility of FILESTREAM Data

You can execute the following statement in the Query Editor window to configure the FILESTREAM data access:

```
EXEC                    sp_configure
filestream_access_level, 2
```

Then, execute the following statement to apply the configuration changes:

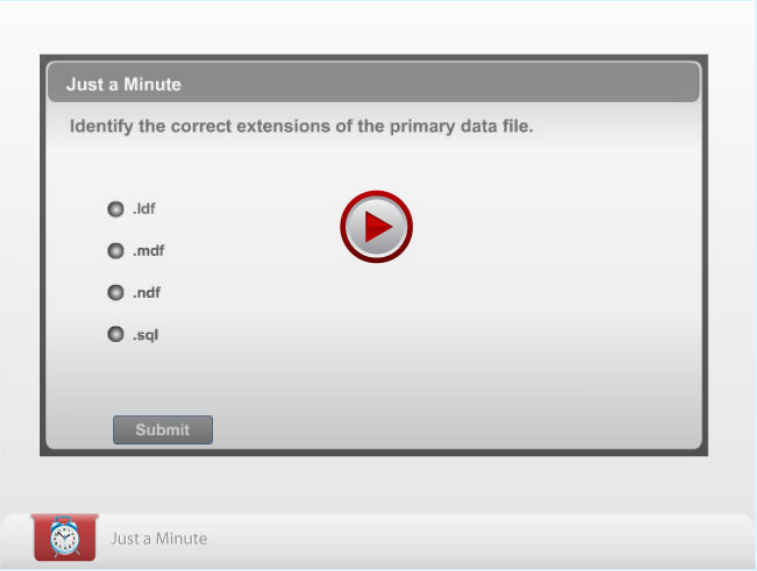
```
RECONFIGURE
```

Once the FILESTREAM data access has been configured successfully, you can create the FILESTREAM-enabled database to store the unstructured data. For example, TekSoft, Inc. is a leading software development company. The company wants to store unstructured data such as photos of the employees in the database. Therefore, being a database developer, you can use the following statement to create the FILESTREAM-enabled database named TekSoft:

```
CREATE DATABASE TekSoft
 ON
 PRIMARY ( NAME = TekSoft,
 FILENAME = 'C:\Data\TekSoft.mdf'),
 FILEGROUP   FileStreamGroup   CONTAINS
FILESTREAM(NAME = TekSoft_Data,
 FILENAME = 'C:\Data\TekSoft_Data')
 LOG ON ( NAME = Log1,
 FILENAME = 'C:\Data\TekSoft.ldf')
 GO
```

In the preceding statement, the CONTAINS FILESTREAM clause is used to specify that the database will store the FILESTREAM data. After the execution of the preceding statement, the TekSoft_Data folder and the TekSoft.ldf and TekSoft.mdf files are created in the Data folder of the C drive.

The TekSoft_Data folder contains the $FSLOG folder and the filestream.hdr file. The $FSLOG folder contains the log information of the FILESTREAM-enabled tables in the TekSoft database. However, the filestream.hdr is a system file that contains the FILESTREAM header information and must not be deleted.



Renaming a User-Defined Database

You can rename a database whenever required. Only a system administrator or the database owner can rename a database. The sp_renamedb stored procedure is used to rename a database. The syntax of the sp_renamedb statement is:

```
sp_renamedb          old_database_name,
new_database_name
```

where,
old_database_name is the current name of the database.
new_database_name is the new name of the database.
For example, the following statement renames the Personnel database to the Employee database:

```
sp_renamedb Personnel, Employee
```

Dropping a User-Defined Database

You can delete a database when it is no longer required. This causes all the database files and data to be deleted. Only the users with sysadmin role and the database owner have the permissions to delete a database. The DROP DATABASE statement is used to delete a database. The syntax of the DROP DATABASE statement is:

```
DROP DATABASE database_name
```

where,
database_name is the name of the database.
The following statement deletes the Employee database:

```
DROP DATABASE Employee
```

You cannot delete a system-defined database.

You can rename or delete a database by right-clicking the database to be renamed or deleted under the Databases folder in the Object Explorer window, and then selecting the Rename or Delete option from the pop-up menu.

Managing Tables

A table is a database object used to store data. Data in a table is organized in rows and columns. Each row in a table represents a unique record and each column represents an attribute of the record. The column names within a table must be unique, but the same column name can be used in different tables within a database.
As a database developer, you need to create tables to store data. While creating tables in a relational database, you must ensure that no one enters invalid data in it. For this, you need to apply certain rules and constraints on columns that specify the kind of data to be stored. In addition, you need to specify the relationships between various tables.
If you want to store a large volume of data in a table, you can create a partitioned table. This helps in improving query performance.
In addition to creating tables, you are responsible for managing tables. Management of tables involves modifying tables to add columns or to change the rules imposed on the table. It also involves deleting tables, when they are no longer required.

Creating a Table

In SQL Server, you can create a table by using the CREATE TABLE statement. The syntax of the CREATE TABLE statement is:

```
CREATE TABLE
 [ database_name. [ schema_name ] .]
table_name
 (     {     <column_definition>     |
<computed_column_definition> }
 [IDENTITY (SEED, INCREMENT)]
 [ <table_constraint> ] [ ,...n ] )
 [    ON    {    partition_scheme_name
( partition_column_name ) | filegroup
 | "default" } ]
 [   {   TEXTIMAGE_ON   {   filegroup   |
"default" } ]
 [ ; ]
```

where,
database_name specifies the name of the database where the table is created. If you do not specify a database name, the table is created in the current database.
schema_name specifies the schema name where the new table belongs. Schema is a logical group of database objects in a database. Schemas help in improving manageability of objects in a database.
table_name specifies the new table name. The table name can be a maximum of 128 characters.
column_name specifies the name of the column and must be unique in the table. It can be a maximum of 128 characters.
computed_column_definition specifies the expression that produces the value of a computed column. A computed column does not exist physically in the memory, but it is used to generate a computed value. The other columns in a table, which are not computed columns, are called persisted columns.
IDENTITY is used for those columns that need automatically generated, unique system values. This property can be used to generate sequential numbers.
SEED is the starting or the initial value for the IDENTITY column.
INCREMENT is the step value used to generate the next value for the column. This value can also be negative.
table_constraint is an optional keyword that specifies the PRIMARY KEY, NOT NULL, UNIQUE, FOREIGN KEY, or CHECK constraint.
partition_scheme_name specifies the partition scheme name that defines the filegroups on which the partition of a table is mapped. Ensure that the partition scheme exists within the database.
partition_column_name specifies the column name on which a partitioned table will be partitioned.

TEXTIMAGE_ON { filegroup | "default" } are keywords that specify that the text, ntext, image, xml, varchar(max), nvarchar(max), varbinary(max), and CLR user-defined type columns are stored on the specified filegroup. If there are no large value columns in the table, TEXTIMAGE_ON is not allowed.

You need to have the CREATE TABLE permissions to create a table in a database.

Consider the following example. The management of AdventureWorks, Inc. needs to maintain the leave details of the employees. For this, you need to create a table named EmployeeLeave in the HumanResources schema.

The following table shows the structure of the EmployeeLeave table.

Columns	***Data Type***	***Checks***
EmployeeID	*Int*	*NOT NULL*
LeaveStartDate	*Date*	*NOT NULL*
LeaveEndDate	*Date*	*NOT NULL*
LeaveReason	*varchar(100)*	*NOT NULL*
LeaveType	*char(2)*	*NOT NULL*

The Structure of the EmployeeLeave Table

You can use the following statement to create the table:

```
CREATE TABLE HumanResources.EmployeeLeave
 (
 EmployeeID int NOT NULL,
 LeaveStartDate datetime NOT NULL,
 LeaveEndDate datetime NOT NULL,
 LeaveReason varchar(100),
 LeaveType char(2)NOT NULL
 )
 GO
```

You can use the system stored procedure named sp_help to view a table structure. To view the stucture of HumanResources.EmployeeLeave table, you can use the following statement:

```
sp_help 'HumanResources.EmployeeLeave'
```

When you execute the preceding statement, the output contains the details of all the columns, keys, constraints, and indexes applied on the HumanResources.EmployeeLeave table.

In the EmployeeLeave table, you want that the values for all the columns should be inserted by a user. However, the values in the EmployeeID column should be generated automatically in sequential order. You can perform this task by assigning an IDENTITY property to the EmployeeID column.

If you assign an IDENTITY property to a column, SQL Server automatically generates sequential numbers for that column whenever new rows are inserted in the table. You can use the following statement to create an IDENTITY column in a table:

```
CREATE TABLE Emp
 (EmpCode int IDENTITY(100,1),
 EmpName char(25) NOT NULL,
 DeptNo char(4) NOT NULL)
 GO
```

The preceding statement creates a table Emp with the IDENTITY column. The EmpCode column of the Emp table is the IDENTITY column with the starting value (SEED) as 100 and the step value (INCREMENT) as 1. Therefore, the EmpCode column will store values as 100, 101, 102, and so on.

Consider another example where the management of AdventureWorks, Inc. needs to maintain the project details assigned to their employees. For this, you need to create a Project table in the HumanResources schema. The following table shows the structure of the Project table.

Columns	***Data Type***
ProjectCode	*Int*
ProjectManagerID	*Int*
Description	*varchar(50)*
StartDate	*Datetime*
EndDate	*Datetime*

The Structure of the Project Table

You can use the following statement to create the Project table:

```
CREATE TABLE HumanResources.Project
 (
 ProjectCode int,
 ProjectManagerID int,
 Description varchar(50),
 StartDate datetime,
 EndDate datetime
 )
```

The preceding statement creates a table named Project in the HumanResources schema where all the columns can store null values.

Sometimes, a column in a table contains a maximum number of NULL values, instead of actual data. For example, there is a column named OrderReturnDate in the Orders table. This column contains the date when the order was returned. There could be less data in this column as not many orders are returned. Therefore, most of the rows in this column would contain NULL values. To optimize the storage space for NULL values in the database engine, you can declare such columns as SPARSE. If a column contains zero or NULL values, the SPARSE column would require lesser space as compared to a regular column.

Consider the following statement:

```
CREATE TABLE Ord_Det
```

```
(
OrderID int,
OrderDate datetime,
OrderCost money,
OrderRetDate datetime SPARSE NULL
)
```

In the preceding staement, a table named Ord_Det is created that contains a SPARSE column named OrderRetDate. While defining a SPARSE column, you must consider the following guidelines:

- A sparse column must be nullable.
- A sparse column cannot have the ROWGUIDCOL property or the IDENTITY property.
- A sparse column can be defined on any, except the following data types:
  - Text
  - Ntext
  - Image
  - Timestamp
  - Geometry
  - geography
- A sparse column cannot be of the user-defined data type.
- A sparse column cannot have the FILESTREAM attribute.
- A sparse column cannot have a default value.
- A sparse column cannot be bound to a rule.
- A computed column cannot be marked as SPARSE although a computed column can be a part of the sparse column.
- A sparse column cannot be a part of a clustered index or a unique primary key index.

Consider another example, where you need to create the EmpDetails table to maintain the employee details. The following table shows the structure of the EmpDetails table.

Columns	***Data Type***	***Checks***
EmployeeID	*UNIQUEIDENTIFIER ROWGUIDCOL*	*NOT NULL*
EmployeeName	*varchar(30)*	*NOT NULL*
EmployeeAddress	*varchar(100)*	*NOT NULL*
EmployeeDept	*varchar(20)*	*NOT NULL*
EmployeePhoto	*varbinary(max)*	*NOT NULL*

The Structure of the EmpDetails Table

In the preceding table, the photo of the employee needs to be stored in the EmloyeePhoto column of the EmpDetails table. Photo is a graphic image and can be stored as a FILESTREAM data in the table. You can use the following statements to create the EmpDetails table that stores the FILESTREAM data:

```
USE TekSoft
 GO
 CREATE TABLE EmpDetails
 (
 EmployeeID            UNIQUEIDENTIFIER
ROWGUIDCOL NOT NULL UNIQUE,
 EmployeeName varchar(30),
 EmployeeAddress varchar(100),
 EmployeeDept varchar(20),
 EmployeePhoto          VARBINARY(MAX)
FILESTREAM
 )
 GO
```

In the preceding statements, the EmployeeID column is declared as UNIQUEIDENTIFIER ROWGUIDCOL, which uniquely identifies the records in the table across the databases. The values of the column set with the FILESTREAM attribute are stored in the Windows file system. Therefore, you must declare a column as UNIQUEIDENTIFIER ROWGUID while creating a table that stores the FILESTREAM data.

In addtion to storing the FILESTREAM data, you can store the spatial data in a table by using the geometry and geography data types. For example, you need to create the County_Location table to store the country ID and location of the country. The following table shows the structure of the Country_Location table.

Columns	***Data Type***	***Checks***
CountryID	*Int*	*NOT NULL*
CountryLocation	*geography*	*NOT NULL*

The Structure of the Country_Location Table

You can use the following statements to create the Country_Location table:

```
USE AdventureWorks
 GO
 CREATE TABLE Country_Location
 (
 CountryID int,
 CountryLocation geography
 )
 GO
```

In the preceding statements, the CountryLocation column is declared as the geography data type that can store the latitude and longitude cordinates, which indicate the location of the country.

Task 4.2: Creating a Table by Using SQL Server Management Studio

Task 4.3: Creating a Table by Using the Insert Snippet Menu

Implementing Data Integrity

If checks are not applied while defining and creating tables, the data stored in the tables can become redundant. For example, if you do not store the data about all the employees with complete address details, then the data would not be useful.

Similarly, if a database used by the Human Resource department stores employee contact details in two separate tables, the details of the employees might not match. This would result in inconsistency and confusion.

Therefore, it is important to ensure that the data stored in tables is complete and consistent. The concept of maintaining consistency and completeness of data is called *data integrity*. Data integrity is enforced to ensure that the data in a database is accurate, consistent, and reliable. It is broadly classified into the following categories:

- **Entity integrity**: Ensures that each row can be uniquely identified by an attribute called the primary key. The primary key column contains a unique value in all the rows. In addition, this column cannot be NULL. Consider a situation where there might be two candidates for an interview with the same name 'Jack'. By enforcing entity integrity, the two candidates can be identified by using the unique code assigned to them. For example, one candidate can have the code 001 and the other candidate can have the code 002.
- **Domain integrity**: Ensures that only a valid range of values is stored in a column. It can be enforced by restricting the type of data, the range of values, and the format of data. For example, you have a table called BranchOffice with a column called City that stores the names of the cities where the branch offices are located. The offices are located in 'Beijing', 'Nanjing', 'Hangzhou', 'Dalian', 'Suzhou', 'Chengdu', and 'Guangzhou'. By enforcing domain integrity, you can ensure that only valid values (as per the list specified) are entered in the City column of the BranchOffice table. Therefore, the user will not be allowed to store any other city names like 'New York' or 'London' in the City column of the BranchOffice table.
- **Referential integrity**: Ensures that the values of the foreign key match the value of the corresponding primary key. For example, if a bicycle has been ordered and an entry is to be made in the OrderDetail table, then that bicycle code should exist in the Product table. This ensures that an order is placed only for the bicycle that is available.
- **User-defined integrity**: Refers to a set of rules specified by a user, which do not belong to the entity,*domain*, and *referential integrity* categories.

When creating tables, SQL Server allows you to maintain integrity by:

- Applying constraints.
- Enabling and disabling constraints.
- Applying rules.
- Using user-defined data types.

Applying Constraints

Consider an example where a user entered a duplicate value in the EmployeeID column of the Employee table. This would mean that the two employees have same employee ID. This would further result in erroneous results when anybody queries the table. As a database developer, you can prevent this by enforcing data integrity on the table by using constraints.

Constraints define rules that must be followed to maintain consistency and correctness of data. A constraint can be either created while creating a table or added later. When a constraint is added after the table is created, it checks the existing data. If there is any violation, then the constraint is rejected.

A constraint can be created by using either of the following statements:

- CREATE TABLE statement
- ALTER TABLE statement

A constraint can be defined on a column while creating a table. It can be created with the CREATE TABLE statement. The syntax of adding a constraint at the time of table creation is:

```
CREATE TABLE table_name
 (
 column_name                    CONSTRAINT
constraint_name          constraint_type
[,CONSTRAINT             constraint_name
constraint_type]
```

where,

`column_name` is the name of the column on which the constraint is to be defined.

`constraint_name` is the name of the constraint to be created and must follow the rules for the identifier.

`constraint_type` is the type of the constraint to be added.

Constraints can be divided into the following types:

- Primary key constraint
- Unique constraint
- Foreign key constraint
- Check constraint

- Default constraint

Primary Key Constraint

A primary key constraint is defined on a column or a set of columns whose values uniquely identify all the rows in a table. These columns are referred to as the primary key columns. A primary key column cannot contain NULL values since it is used to uniquely identify rows in a table. The primary key constraint ensures entity integrity.

You can define a primary key constraint while creating the table or you can add it later by altering the table. However, if you define the primary key constraint after inserting rows, SQL Server will give an error if the rows contain duplicate values in the column. While defining a primary key constraint, you need to specify a name for the constraint. If a name is not specified, SQL Server automatically assigns a name to the constraint.

If a primary key constraint is defined on a column that already contains data, then the existing data in the column is screened. If any duplicate values are found, then the primary key constraint is rejected. The syntax of applying the primary key constraint while creating a table is:

```
CREATE TABLE table_name
 (
 col_name [CONSTRAINT constraint_name
PRIMARY KEY [CLUSTERED|NONCLUSTERED]
 col_name [, col_name [, col_name
[, ...]]]
 )
```

where,

`constraint_name` specifies the name of the constraint to be created.

`CLUSTERED|NONCLUSTERED` are the keywords that specify if a clustered or a nonclustered index is to be created for the primary key constraint.

`col_name` specifies the name of the column(s) on which the primary key constraint is to be defined.

You will learn more about indexes in Chapter 6.

In the Project table, you can add a primary key constraint while creating the table. You can use the following statement to apply the primary key constraint:

```
CREATE TABLE HumanResources.Project
 (
 ProjectCode int CONSTRAINT
pkProjectCode PRIMARY KEY,
 ...
 ...
 )
```

The preceding statement will create the Project table with a primary key column, ProjectCode.

You can create a primary key using more than one column. For example, you can set the EmployeeID and the LeaveStartDate columns of the EmployeeLeave table as a composite primary key. You can use the following statement to apply the composite primary key constraint:

```
CREATE TABLE
HumanResources.EmployeeLeave
 (
 EmployeeID int,
 LeaveStartDate datetime
 CONSTRAINT cpkLeaveStartDate PRIMARY
KEY(EmployeeID, LeaveStartDate),
 ...
 ...
 ...
 )
```

The preceding statement creates the EmployeeLeave table with a composite primary key constraint on EmployeeID and LeaveStartDate. The name of the constraint is cpkLeaveStartDate.

Unique Constraint

The unique constraint is used to enforce uniqueness on non-primary key columns. A primary key constraint column automatically includes a restriction for uniqueness. The unique constraint is similar to the primary key constraint except that it allows one NULL row. Multiple unique constraints can be created on a table. The syntax of applying the unique constraint when creating a table is:

```
CREATE TABLE table_name
 (
 col_name [CONSTRAINT constraint_name
UNIQUE [CLUSTERED | NONCLUSTERED]
 (col_name [, col_name [, col_name
[, ...]]])
 col_name [, col_name [, col_name
[, ...]]]
 )
```

where,

`constraint_name` specifies the name of the constraint to be created.

`CLUSTERED | NONCLUSTERED` are the keywords that specify if a clustered or a nonclustered index is to be created for the unique constraint.

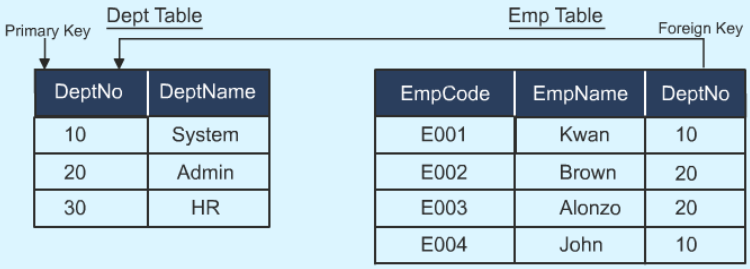
`col_name` specifies the name of the column(s) on which the unique constraint is to be defined.

You can use the following statement to enforce the unique constraint on the Description column of the Project table:

```
CREATE TABLE HumanResources.Project
 (
 ProjectCode int CONSTRAINT
pkProjectCode PRIMARY KEY,
 Description varchar(50) CONSTRAINT
unDesc UNIQUE,
 ...
 ...
 )
```

Foreign Key Constraint

You can use the foreign key constraint to remove the inconsistency in two tables when the data in one table depends on the data in the other table. A foreign key refers the primary key column of another table, as shown in the following figure.



DeptNo	DeptName
10	System
20	Admin
30	HR

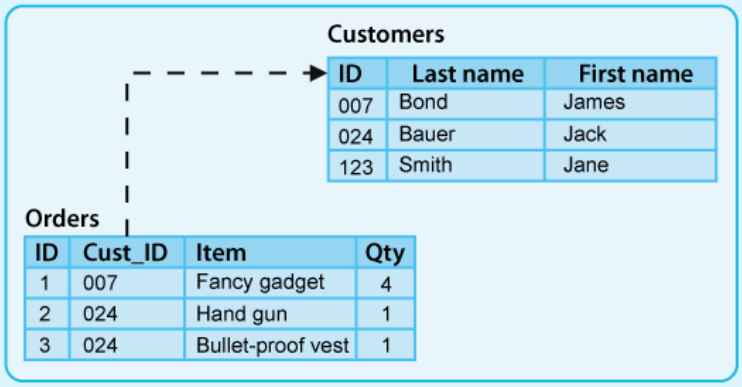
EmpCode	EmpName	DeptNo
E001	Kwan	10
E002	Brown	20
E003	Alonzo	20
E004	John	10

The Primary Key and Foreign Key Columns

The preceding figure contains two tables, Dept and Emp. The DeptNo column is the primary key in the Dept table and foreign key in the Emp table.

Consider the example of Customers and Orders table.

The following figure represents the primary key and foreign key relationship between the Customers and the Orders table.



The Primary Key and Foreign Key Relationship

In the preceding figure, ID is the primary key column of the Customers table and Cust_ID is the foreign key column in the Orders table.

A foreign key constraint associates one or more columns (the foreign key) of a table with an identical set of columns (a primary key column) in another table on which a primary key constraint has been defined. The syntax of applying the foreign key constraint when creating a table is:

```
CREATE TABLE table_name
 (
 col_name [CONSTRAINT constraint_name
FOREIGN KEY (col_name [, col_name
[, ...]])
 REFERENCES table_name (column_name [,
column_name [, ...]])]
 (col_name [, col_name [, col_name
[, ...]]])
 col_name [, col_name [, col_name
[, ...]]]
 )
```

where,

`constraint_name` is the name of the constraint on which the foreign key constraint is to be defined.

`col_name` is the name of the column on which the foreign key constraint is to be enforced.

`table_name` is the name of the related table in which the primary key constraint has been specified.

`column_name` is the name of the primary key column of the related table on which the primary key constraint has been defined.

For example, in the EmployeeLeave table of the HumanResources schema, you need to add the foreign key constraint to enforce referential integrity. The EmployeeID column is set as a primary key in the Employee table of the HumanResources schema. Therefore, you need to set EmployeeID in the EmployeeLeave table as a foreign key. You can use the following statement to apply the foreign key constraint in the EmployeeLeave table:

```
CREATE TABLE
HumanResources.EmployeeLeave
 (
 EmployeeID int CONSTRAINT
fkEmployeeID REFERENCES
HumanResources.Employee(EmployeeID),
 LeaveStartDate datetime CONSTRAINT
cpkLeaveStartDate PRIMARY KEY
(EmployeeID, LeaveStartDate),
 ...
 ...
 ...
 )
```

You can also apply the foreign key constraint in the EmployeeLeave table by using the following statement:

```
CREATE TABLE
HumanResources.EmployeeLeave
 (
 EmployeeID int,
 LeaveStartDate datetime CONSTRAINT
cpkLeaveStartDate PRIMARY KEY
(EmployeeID, LeaveStartDate),
 ...
 ...

 CONSTRAINT fkEmployeeID FOREIGN KEY
(EmployeeID) REFERENCES
HumanResources.Employee(EmployeeID)
 )
```

The preceding statement creates the EmployeeLeave table with a foreign key constraint on the EmployeeID column. The name of the constraint is fkEmployeeID.

A foreign key can also refer to the primary key column in the same table. Consider the following EmployeeDet table.

Emp_ID	*Mgr_ID*	*Emp_Name*
1	*NULL*	*Mark*
2	*1*	*Jack*
3	*1*	*Joe*
4	*2*	*Michel*
5	*3*	*Steve*

The EmployeeDet Table

The table contains the details of the employees. Emp_ID is the primary key in the table. Each employee record has a field named Mgr_ID that contains the ID of the reporting Manager of that employee. Each Manager is also an employee. Therefore, the details of the Managers are also stored in the same table. In this case, the Mgr_ID column can be a foreign key that refers to the Emp_ID column of the same table. Therefore, the records of the EmployeeDet table are related with other records of the same table by creating a parent-child relationship in the same table. Such a table is called a self-referencing table.

In this example, Joe is Steve's Manager. Joe, in turn, reports to Mark. The Manager of Michel is Jack. Jack also reports to Mark. Mark is the CEO of the organization and is at the top most level in the organizational hierarchy. He does not have a Manager. Therefore, the value of the Mgr_ID column is null for Mark.

You can create a self-referencing table by using the following statement:

```
CREATE TABLE EmployeeDet
 (
 Emp_id int PRIMARY KEY,
 Mgr_ID int,
 Emp_Name varchar(20),
 FOREIGN KEY (mgr_id)
 REFERENCES EmployeeDet(Emp_id)
 );
```

The following statements can be used to insert the records in the EmployeeDet table:

```
INSERT INTO EmployeeDet VALUES
(1,NULL,'Mark')
 INSERT INTO EmployeeDet VALUES
(2,1,'Jack')
 INSERT INTO EmployeeDet VALUES
(3,1,'Joe')
 INSERT INTO EmployeeDet VALUES
(4,2,'Michel')
 INSERT INTO EmployeeDet VALUES
(5,3,'Steve')
```

The preceding statements execute successfully. Now, consider the following `INSERT` statement:

```
INSERT INTO EmployeeDet VALUES
(6,10,'Harry')
```

The preceding statement gives an error because the foreign key constraint prevents the insertion of a record with Mgr_ID that does not exist in the table. However, you may assign a NULL value for Mgr_ID as done in the first `INSERT` statement.

In addition to the foreign key constraint, you can apply the cascading referential integrity constraint to the tables having foreign key relationships. The cascading referential integrity constraint defines the action that SQL Server performs when an attempt is made to update or delete a row in a table with a key referenced by a foreign key in another table. SQL Server supports the ON DELETE and ON UPDATE clauses to apply the cascading referential integrity constraint.

The ON DELETE and ON UPDATE clauses can be used with the following options:

- **ON DELETE | ON UPDATE NO ACTION**: Specifies that if an attempt is made to delete or update a primary key record, an error will be raised and the delete or update operation will be rolled backed.
- **ON DELETE | ON UPDATE CASCADE**: Specifies that if an attempt is made to update or delete a primary key record, the corresponding foreign key record is also updated or deleted, respectively.
- **ON DELETE | ON UPDATE SET NULL**: Specifies that if an attempt is made to delete or update a primary key record, the corresponding foreign key record is set to NULL.
- **ON DELETE | ON UPDATE SET DEFAULT**: Specifies that if an attempt is made to delete or update a primary key record, the corresponding foreign key record is set to its default values. However, to apply this constraint, all foreign key columns must have a default definition.

For example, the EmployeeLeave table is associated with the Employee table through the foreign key relationship. You can use the following statement to apply the cascading referential integrity constraint on the Employee table:

```
ALTER TABLE
HumanResources.EmployeeLeave
 ADD CONSTRAINT rfkcEmployeeID FOREIGN
KEY(EmployeeID)
 REFERENCES HumanResources.Employee
(EmployeeID)
 ON DELETE NO ACTION ON UPDATE NO
ACTION
```

In the preceding statement, the ON DELETE NO ACTION and ON UPDATE NO ACTION clauses ensure that any attempt to delete or update the EmployeeID in the Employee table will not be successful.

Check Constraint

A check constraint enforces domain integrity by restricting the values to be inserted in a column. It is possible to define multiple check constraints on a single column.

These are evaluated in the order in which they are defined.
The syntax of applying the check constraint is:

```
CREATE TABLE table_name
 (
 col_name [CONSTRAINT constraint_name]
CHECK (expression)
 (col_name [, col_name [, ...]])
 .
 )
```

where,

`constraint_name` specifies the name of the constraint to be created.

`expression` specifies the conditions that define the check to be made on the column. It can include elements, such as arithmetic operators and relational operators, or keywords, such as IN, LIKE, and BETWEEN.

A single check constraint can be applied to multiple columns when it is defined at the table level. For example, while entering project details, you want to ensure that the start date of the project must be less than or equal to the end date.

You can use the following statement to apply the check constraint on the Project table:

```
CREATE TABLE HumanResources.Project
 (
 ........
 ........
 StartDate datetime,
 EndDate datetime,
 Constraint chkDate CHECK (StartDate
<= EndDate)
 )
```

A check constraint can be specified by using the following keywords:

- **IN**: To ensure that the values entered are from a list of constant expressions. The following statement creates a check constraint, chkLeave on the LeaveType column of the HumanResources.EmployeeLeave table, thereby restricting the entries to valid leave types:

```
CREATE TABLE
HumanResources.EmployeeLeave
 (
 EmployeeID int,
 LeaveStartDate datetime
CONSTRAINT cpkLeaveStartDate
PRIMARY
 KEY(EmployeeID, LeaveStartDate),
 LeaveEndDate datetime NOT NULL,
 LeaveReason varchar(100),
 LeaveType char(2) CONSTRAINT
chkLeave CHECK(LeaveType
 IN('CL','SL','PL'))
 )
```

  The preceding statement ensures that the leave type can be any one of the three values: CL, PL, or SL. Here, CL stands for Casual Leave, SL stands for Sick Leave, and PL stands for Privileged Leave.

- **LIKE**: To ensure that the values entered in specific columns are of a certain pattern. This can be achieved by using wildcards. For example, the following statement creates a check constraint on DeptCode column of the Emp table:

```
CREATE TABLE Emp
 (
 ...
 ...
 DeptNo char(4) CHECK (DeptNo
LIKE '[0-9][0-9][0-9][0-9]')
 )
```

  In the preceding statement, the check constraint specifies that the DeptNo column can contain only a value that consists of characters from 0-9.

- **BETWEEN**: To specify a range of constant expressions by using the BETWEEN keyword. The upper and lower boundary values are included in the range. For example, the following statement creates a check constraint on the sal column of the EmpTable table:

```
CREATE TABLE EmpTable
 (
 ...
 ...
 sal money CHECK (sal BETWEEN
20000 AND 80000)
 )
```

  In the preceding statement, the check constraint specifies that the sal column can have a value only between 20000 and 80000.

The rules to be followed while creating the check constraint are:

- It can be created at the column level.
- It can contain user-specified search conditions.
- It cannot contain subqueries.
- It does not check the existing data in the table if created with the WITH NOCHECK option.
- It can reference other columns of the same table.

Default Constraint

A default constraint can be used to assign a constant value to a column, and the user need not insert values for such a column. Only one default constraint can be created for a column, but the column cannot be an IDENTITY column.

The syntax of applying the default constraint while creating a table is:

```
CREATE TABLE table_name
 (
 col_name [CONSTRAINT constraint_name]
DEFAULT (constant_expression | NULL)
 (col_name [, col_name [, ...]])
```

)

where,

`constraint_name` specifies the name of the constraint to be created.

`constant_expression` specifies an expression that contains only constant values. This can contain a NULL value.

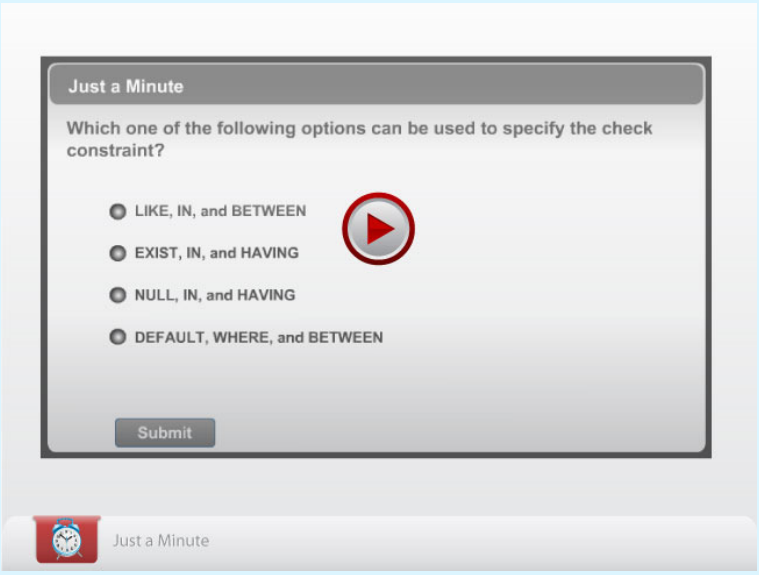
For example, while creating the EmployeeLeave table, you can insert a default constraint to add a default value for the LeaveType column. You can set the default leave type as PL.

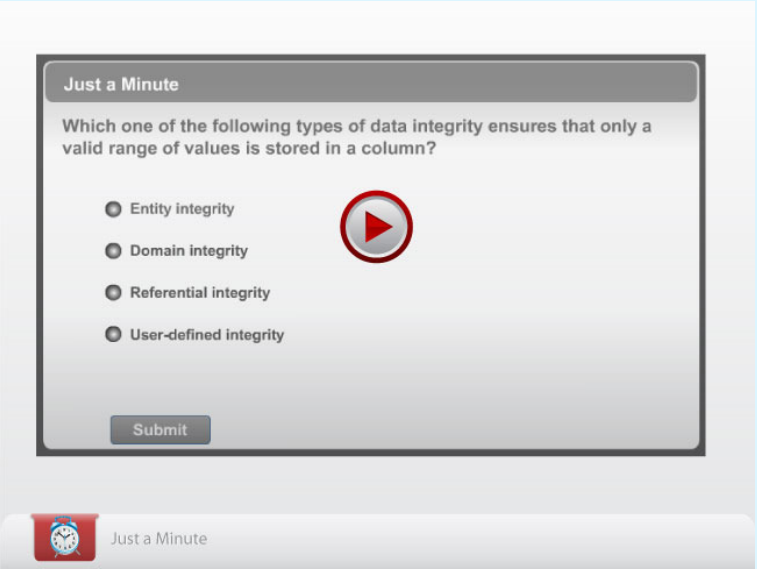
You can use the following statement to create the default constraint:

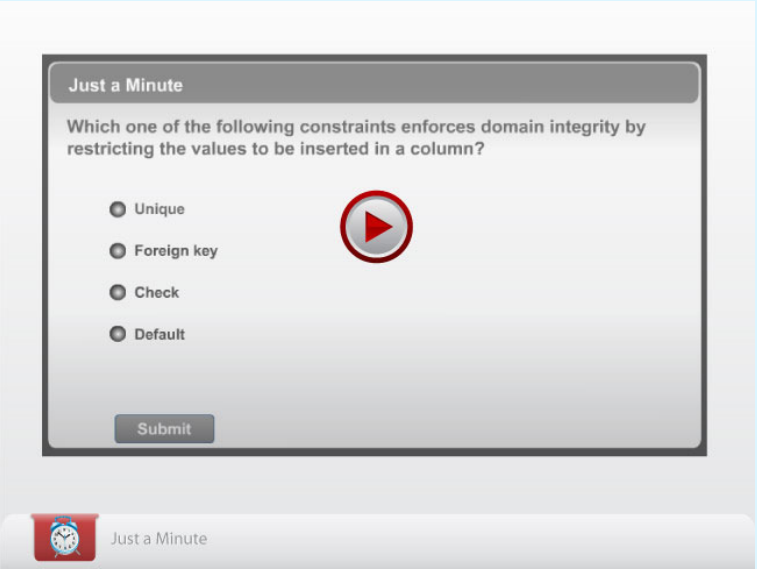
```
CREATE TABLE HumanResources.EmployeeLeave
(
......
 LeaveType char(2) CONSTRAINT chLeave
CHECK(LeaveType IN('CL','SL','PL'))
CONSTRAINT chkDefLeave DEFAULT 'PL'
)
```

The preceding statement creates the EmployeeLeave table with a default constraint on the LeaveType column, where the default value is specified as PL. The name of the constraint is chkDefLeave.

You can also use the DEFAULT database objects to create a default constraint that can be applied to columns across tables within the same database. For this, you can create database objects by using the CREATE DEFAULT statement.







Enabling and Disabling Constraints

Constraints are applied on a table to keep a check on the values inserted in the table. Sometimes, after applying a constraint, you need to insert a value that contradicts the definition of a constraint applied on that table. To accomplish this task, you first need to disable that constraint, and then insert the value in the table. After inserting the value, you can again enable the constraint. For example, you have applied a check constraint on the OrderDate column of the Orders table. This column accepts date between 15/01/2001 and 15/01/2010. Later, you realized that you need to insert details of an order with the order date 12/01/2000 in the Orders table. Therefore, you need to disable the check constraint applied on the Orders table. After inserting the required data, you can again enable the check constraint on the Orders table.

For example, the check constraint is applied on the LeaveType column of the EmployeeLeave table. This check constraint allows only the values CL, SL, and PL in the LeaveType column. However, you need to insert a record in the LeaveType column for maternity leave. For this, you have to specify ML in the LeaveType column. To

accomplish this task, you need to disable the check constraint applied on the EmployeeLeave table by using the following statement:

```
Use AdventureWorks
 ALTER                                TABLE
HumanResources.EmployeeLeave
 NOCHECK CONSTRAINT chkLeave
```

In the preceding statement the NOCHECK CONSTRAINT clause is used to disable the check constraint applied on the EmployeeLeave table. Then, you can successfully insert a record with the ML leave type value by using the following statement:

```
INSERT                                INTO
HumanResources.EmployeeLeave
 VALUES(101,  15/12/2009,  20/12/2009,
'Maternity Leave', 'ML')
```

After inserting the required record, you can enable the check constraint applied on the EmployeeLeave table by using the following statement:

```
ALTER                                 TABLE
HumanResources.EmployeeLeave
 CHECK CONSTRAINT chkLeave
```

The preceding statement enables the check constraint again. If you try to execute the following statement after enabling the constraint, an error is displayed:

```
INSERT                                INTO
HumanResources.EmployeeLeave
 VALUES(102,  20/12/2009,  20/02/2010,
'Maternity Leave', 'ML')
```

The execution of the preceding statement displays the following error:

```
Msg 547, Level 16, State 0, Line 1
 The INSERT statement conflicted with
the CHECK constraint "chkLeave".
 The conflict occurred in database
"AdventureWorks",                  table
"HumanResources.EmployeeLeave", column
'LeaveType'.
 The statement has been terminated.
```

Applying Rules

A rule enforces domain integrity for columns or user-defined data types. The rule is applied to a column or a user-defined data type before an INSERT or UPDATE statement is issued. In other words, a rule specifies a restriction on the values of a column or a user-defined data type. Rules are used to implement business-related restrictions or limitations. A rule can be created by using the CREATE RULE statement.

The syntax of the CREATE RULE statement is:

```
CREATE      RULE      rule_name      AS
conditional_expression
```

where,

`rule_name` specifies the name of the new rule that must conform to the rules for identifiers.

`conditional_expression` specifies the condition(s) that defines the rule. It can be any expression that is valid in a WHERE clause and can include elements such as arithmetic operators, relational operators, IN, LIKE, and BETWEEN.

The variable specified in the conditional expression must be prefixed with the '@' symbol. The expression refers to the value that is being specified with the INSERT or UPDATE statement.

In the preceding example of the EmployeeLeave table, you applied the check constraint on the LeaveType column to accept only three values: CL, SL, and PL. You can perform the same task by creating a rule, as shown in the following statement:

```
USE AdventureWorks
 GO
 CREATE RULE ruleType
 AS @LeaveType IN ('CL', 'SL', 'PL')
 GO
```

The following table lists the examples to create rule.

Example	***Description***
CREATE RULE dept_name_rule AS @deptname NOT IN ('accounts','stores')	*Specifies that if the value of deptname is accounts or stores, then the value is to be rejected from being inserted into the column to which the rule is bound.*
CREATE RULE min_price_rule AS @minprice >= $5000	*Allows a value of $5000 or more to be inserted in the column to which the rule is bound.*
CREATE RULE emp_code_rule AS @empcode LIKE '[F-M][A-Z][0-9][0-9][0-9]'	*Specifies that the value to be inserted in the column, to which the rule will be bound, must follow the pattern specified in the LIKE clause. The first character of the string can be any value between F to M; the second character can be any value ranging from A to Z; and from the third character onwards, the acceptable range is a numeric value from 0 to 9.*

The Examples of Creating Rules

After you create a rule, you need to activate the rule by using a stored procedure, sp_bindrule.

The syntax of sp_bindrule is:

```
sp_bindrule <'rule'>, <'object_name'>,
[<'futureonly_flag'>]
```

where,

rule specifies the name of the rule that you want to bind.
object_name specifies the object on which you want to bind the rule.
futureonly_flag applies only when you want to bind the rule to a user-defined data type.
Consider the following example where the rulType rule created for the LeaveType column of the EmployeeLeave table is bound by using the sp_bindrule stored procedure. You can use the following statement to bind the rule:

```
sp_bindrule
'ruleType','HumanResources.EmployeeLea
ve.LeaveType'
```

Similarly, when you want to remove a rule, the sp_unbindrule stored procedure is used. For example, to remove the rule from the EmployeeLeave table, you can use the following statement to unbind the rule:

```
sp_unbindrule
'HumanResources.EmployeeLeave.LeaveTyp
e'
```

A rule can be deleted by using the DROP RULE statement. The syntax for the DROP RULE statement is:

```
DROP RULE rule_name
```

where,
rule_name is the name of the rule to be dropped.
For example, you can use the following statement to delete the rule, ruleType:

```
DROP RULE ruleType
```

Using User-Defined Data Types

User-defined data types are custom data types defined by the users with a custom name. They are based on the system data types. A user-defined data type is basically a named object with the following additional features:

- Defined data type and length
- Defined nullability
- Predefined rule that may be bound to the user-defined data type
- Predefined default value that may be bound to the user-defined data type

You can create user-defined data types by using the CREATE TYPE statement. The syntax of the CREATE TYPE statement is:

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

where,
schema_name specifies the name of the schema to which the *alias* data type or the user defined data type belongs.
type_name specifies the name of the alias data type or the user-defined data type.
base_type specifies SQL Server supplied data type on which the alias data type is based.
precision specifies a non-negative integer that indicates the maximum number of decimal digits that can be stored both to the left and to the right of the decimal point.
scale specifies a non-negative integer that indicates the maximum number of decimal digits that can be stored to the right of the decimal point. It can be specified only if precision is specified and must be less than or equal to the precision.
NULL | NOT NULL specifies whether the data type can hold a null value. If not specified, NULL is the default.
The following statement creates a user-defined data type for descriptive columns:

```
USE AdventureWorks
 GO
 CREATE TYPE DSCRP
 FROM varchar(100) NOT NULL ;
 GO
```

In the preceding statement, a user-defined data type, DSCRP is created to store the varchar data type and the size limit is specified as 100. Further, it also specifies NOT NULL. Therefore, you can use this data for the columns that store description, address, and reason.
For example, you can use the DSCRP data type to store the data of the LeaveReason column of the EmployeeLeave table, as shown in the following statement:

```
CREATE                                TABLE
HumanResources.EmployeeLeave
 (
 .......
 LeaveReason DSCRP,
 LeaveType char(2) CONSTRAINT chkLeave
CHECK(LeaveType     IN('CL','SL','PL'))
CONSTRAINT chkDefLeave DEFAULT 'PL'
 )
```

Creating Synonyms

Consider a scenario. The Department table belongs to the HumanResources schema of the AdventureWorks database. Whenever you need to refer to the Department table in any SQL query, the table name might precede with the name of the schema and database. This is a tedious task. To retrieve all the records from the Department table, you can use the following query:

```
SELECT * FROM AdventureWorks.HumanResources.Department
```

However, if the location of the Department table is changed from AdventureWorks to NarrowFabrics, you need to reflect this change, as shown in the following query:

```
SELECT * FROM NarrowFabrics.HumanResources.Department
```

If there are many such queries used in your application, you need to make changes in each query manually. It is a time-consuming task and may lead to errors if you forget to make changes in few queries. To protect your application code from such changes, SQL Server provides the concept of synonym. SQL Server provides the concept of synonym that is a single-part alias name for a database object with a name, which has multiple parts.

Synonyms provides a layer of abstraction that protects SQL statements from changes made to the database object. You can create a synonym for a database object by using the CREATE SYNONYM statement.

The syntax of the CREATE SYNONYM statement is:

```
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` specifies the name of the schema in which the synonym is created.

`synonym_name` is the name of the synonym to be created.

`server_name` is the name of the server on which the base object is located.

`database_name` is the name of the database in which the base object is located.

`schema_name_2` is the name of the schema of the base object.

`object_name` is the name of the base object that the synonym refers to.

To create a synonym for the Department table in the AdventureWorks database, you can use the following query:

```
CREATE SYNONYM pro_dep
 for
Adventureworks.HumanResources.Department
```

You can retrieve the data from the table by using synonyms, as shown in the following query:

```
SELECT * from pro_dep
```

The following figure displays the output of the preceding query.

	DepartmentID	Name	GroupName	ModifiedDate
1	1	Engineering	Research and Development	1998-06-01 00:00:00.000
2	2	Tool Design	Research and Development	1998-06-01 00:00:00.000
3	3	Sales	Sales and Marketing	1998-06-01 00:00:00.000
4	4	Marketing	Sales and Marketing	1998-06-01 00:00:00.000
5	5	Purchasing	Inventory Management	1998-06-01 00:00:00.000
6	6	Research and Development	Research and Development	1998-06-01 00:00:00.000
7	7	Production	Manufacturing	1998-06-01 00:00:00.000
8	8	Production Control	Manufacturing	1998-06-01 00:00:00.000
9	9	Human Resources	Executive General and Administration	1998-06-01 00:00:00.000
10	10	Finance	Executive General and Administration	1998-06-01 00:00:00.000

The Output Derived After Using the Synonym Object

If you make modifications in the definition of the object referenced by a synonym, you need to drop that synonym and create a new one. For example, the database of the Department table is changed from AdventureWorks to NarrowFabrics. You first need to drop the synonym and then recreate the same. You can drop a synonym by using the DROP SYNONYM statement. The syntax of the DROP SYNONYM statement is:

```
DROP SYNONYM synonym_name
```

where, `synonym_name` is the name of the synonym.

You can drop the pro_dep synonym by using the following statement:

```
DROP SYNONYM pro_dep
```

Now, to create a synonym for the Department table in the Narrow Fabrics database, you can use the following query:

```
CREATE SYNONYM pro_dep
 for
NarrowFabrics.HumanResources.Departmen
t
```

You can retrieve the data from the table by using the following same query:

```
SELECT * from pro_dep
```

The following figure displays the output of the preceding query.

	DepartmentID	Name	GroupName	ModifiedDate
1	1	Engineering	Research and Development	1998-06-01 00:00:00.000
2	2	Tool Design	Research and Development	1998-06-01 00:00:00.000
3	3	Sales	Sales and Marketing	1998-06-01 00:00:00.000
4	4	Marketing	Sales and Marketing	1998-06-01 00:00:00.000
5	5	Purchasing	Inventory Management	1998-06-01 00:00:00.000
6	6	Research and Development	Research and Development	1998-06-01 00:00:00.000
7	7	Production	Manufacturing	1998-06-01 00:00:00.000
8	8	Production Control	Manufacturing	1998-06-01 00:00:00.000
9	9	Human Resources	Executive General and Administration	1998-06-01 00:00:00.000
10	10	Finance	Executive General and Administration	1998-06-01 00:00:00.000

The Output Derived After Using the Synonym Object

NOTE *The synonyms are stored in the sys.synonyms catalog view.*

NOTE *You can use a synonym with SELECT, INSERT, UPDATE, DELETE, and EXECUTE statements. A synonym cannot be altered.*

Creating a Partitioned Table

When the volume of data in a table increases, it takes time to query the data. You can partition such tables and store different parts of the tables in multiple physical locations based on a range of values for a specific column. This helps in managing the data and improving the query performance.

Consider the example of a manufacturing organization. The details of inventory movements are stored in the InventoryIssue table. The table contains a large volume of data. Therefore, the queries take a lot of time to execute, thereby slowing the report generation process.

To improve the query performance, you can partition the table to divide the data based on a condition and store different parts of the data in different locations. The condition can be based on the date of transaction and you can save the data pertaining to five years at a location. After partitioning the table, data can be retrieved directly from a particular partition by mentioning the partition number in the query.

In the preceding example, the partitioned tables were created after the database had been designed. You can also create partitioned tables while designing the database and creating tables. You can plan to create a partitioned table when you know that the data to be stored in the table will be large. For example, if you are creating a database for a banking application and you know that the transaction details will be huge, you can create the transaction tables with partitions.

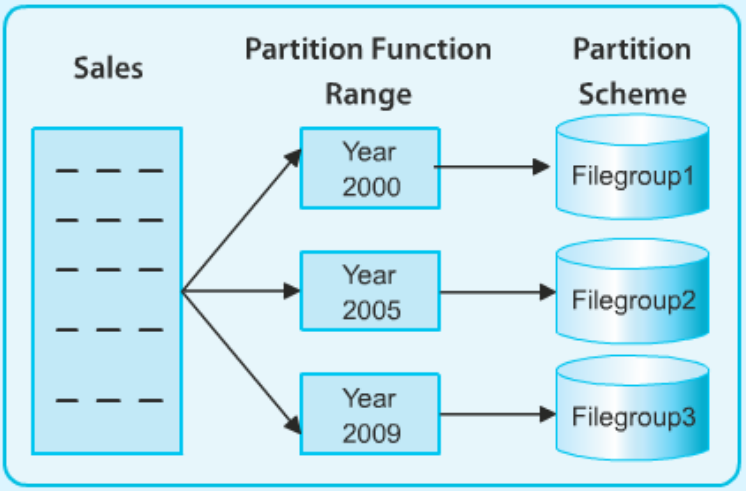
NOTE *Partitioning a table is only allowed in the Enterprise Edition of SQL Server.*

To create a partitioned table, you need to perform the following tasks:

1. Create a partition function.
2. Create a partition scheme.
3. Create a table by using the partition scheme.

For example, an organization stores the sales data of all the products for the last 11 years. However, this results in generation of a large volume of data, which adversely affects the query performance.

To improve the query performance, the database developer needs to partition the table based on a condition, as shown in the following figure.



Partitioning the Table

In the preceding figure, the sales data has been physically stored in different filegroups, on an yearly basis.

Consider a scenario of AdventureWorks database that stores the data of all the employees for the last 11 years. This data includes the personal details of the employees and their payment rates. Whenever there is a change in the payment rate of an employee, it is recorded in a separate record. However, this results in generation of a large volume of data and adversely affects the query performance. To improve the query performance, the database developer needs to partition the table storing the changes in payment rate.

Creating a Partition Function

A partition function specifies how a table should be partitioned. It specifies the range of values in a particular column, based on which the table is partitioned. The syntax of the CREATE PARTITION FUNCTION statement is:

```
CREATE PARTITION FUNCTION
partition_function_name
( input_parameter_type )
 AS RANGE [ LEFT | RIGHT ]
 FOR VALUES ( [ boundary_value
[ ,...n ] ] )
```

where,

`partition_function_name` specifies the name of the partition function.

`input_parameter_type` specifies the data type of the column used for partitioning.

`boundary_value` specifies the boundary values on which a table is to be partitioned.

`...n` specifies the number of boundary_value, which should not exceed 999. The number of partitions created is equal to n + 1.

`LEFT | RIGHT` specifies the side of each boundary value interval to which the boundary_value [,...n] belongs. Left is the default value.

For example, the following statement creates a partition function to partition a table or index into four partitions:

```
CREATE PARTITION FUNCTION pfrange(int)
 AS RANGE LEFT FOR VALUES (1, 100,
1000);
```

In the preceding statement, partition 1 will store the column value less than or equal to 1. The column value between 2 to 100 will be stored in partition 2. The column value between 101 to 1000 will be stored in partition 3. Partition 4 will store the column value which is greater than 1000.

The following table lists the values in each partition for RANGE LEFT.

Partition 1	Partition 2	Partition 3	Partition 4
<=1	>1 to <=100	>100 to <=1000	>1000

The Range LEFT Partition Values

In the preceding statement, if you specify RIGHT instead of LEFT, partition 1 will store the column value which is less than 1. The column value between 1 to 99 will be stored in partition 2. The column value between 100 to 999 will be stored in partition 3. Partition 4 will store the column value that is greater than or equal to1000. The following table lists the values in each partition for RANGE RIGHT.

Partition 1	Partition 2	Partition 3	Partition 4
<1	>=1 to <100	>=100 to <1000	>=1000

The Range RIGHT Partition Values

Consider the scenario of AdventureWorks, where you can partition the data based on years.

The following statement creates a partition function for the same:

```
CREATE PARTITION FUNCTION RateChngDate
( datetime )
 AS RANGE RIGHT FOR VALUES
('1996-01-01', '2000-01-01',
'2004-01-01', '2008-01-01')
```

The preceding statement creates a partition function named RateChngDate. It specifies that the data pertaining to the changes in the payment rate will be partitioned based on the years.

Creating a Partition Scheme

After setting the partition function, you need to create the partition scheme. A partition scheme associates a partition function with various filegroups resulting in the physical layout of the data. The syntax of the CREATE PARTITION SCHEME statement is:

```
CREATE PARTITION SCHEME
partition_scheme_name
 AS PARTITION partition_function_name
 [ ALL ] TO ( { file_group_name |
[ PRIMARY ] } [ ,...n ] )
```

where,

`partition_scheme_name` specifies the name of the partition scheme.

`partition_function_name` specifies the name of the partition function. Partition functions are linked to the filegroups specified in the partition scheme.

`ALL` specifies that all partitions map to a single filegroup provided in the file_group_name.

`file_group_name | [ PRIMARY ] [ ,...n]` specifies the list of filegroup names. If [PRIMARY] is specified, the partition is stored on the primary filegroup. Partitions are assigned to filegroups starting with partition 1, in the order in which the filegroups are listed in [,...n]. n is the number of partitions.

Therefore, before creating a partition scheme, you need to create filegroups.

Task 4.4: Creating Filegroups

After creating the filegroups, execute the following statement in the Query Editor window to create the partition scheme:

```
CREATE PARTITION SCHEME RateChngDate
 AS PARTITION RateChngDate
 TO (Old, First, Second, Third,
Fourth) Creating a Table by Using the
Partition Scheme
```

Creating a Table by Using the Partition Scheme

After you create a partition function and a partition

scheme, you need to create a table that will store the partition records. You can use the following statement to create the partitioned table:

```
Create Table EmpPayHistPart
 (
 EmployeeID int,
 RateChangeDate datetime,
 Rate money,
 PayFrequency tinyint,
 ModifiedDate datetime
 )ON RateChngDate(RateChangeDate)
```

In the preceding statement, the RateChngDate refers to the partition scheme that is applied to the RateChangeDate column. The records entered in the EmpPayHistPart table will be stored based on the condition specified in the partition function.

If you want to display records from a partition instead of displaying the entire table, you can use the following query:

```
SELECT * FROM EmpPayHistPart WHERE
$PARTITION.RateChngDate
(RateChangeDate) = 2
```

In the preceding query, $PARTITION.RateChngDate (RateChangeDate) function returns the partition number. Therefore, the preceding statement will retrieve records from partition 2.

After partitioning a table, your requirements may change and the volume of data in the partitioned table may shrink or expand. In such a case, you can manage the data in a partitioned table by modifying the current structure of the partitioned table. For example, you have partitioned a table into three partitions. As a result, the data in one of the partitions has increased many folds since the partitioning of the table. To handle such a huge data, you need to further divide that partition into two partitions. However, if data in any one of the partitions has reduced significantly, then it is not relevant to assign a separate partition for such low volume of data. Therefore, you can merge this partition with another existing partition and reduce the number of partitions in the table from three to two.

You can modify a partitioned table by:

- Modifying a partition scheme and a partition function.
- Assigning a table as a partition to an existing partitioned table.

Modifying a Partition Scheme and a Partition Function

To modify a partitioned table, you need to perform the following tasks:

- Modify a partition scheme.
- Modify a partition function.

Modifying a Partition Scheme

While creating a partition scheme, you need to create filegroups and files to be associated with the number of partitions created in the partition function. For each partition in the partition function, a filegroup must be defined in the partition scheme. Therefore, the partition scheme must be modified before modifying the partition function.

You can modify a partition scheme by using the ALTER PARTITION SCHEME statement. The syntax of the ALTER PARTITION SCHEME statement is:

```
ALTER PARTITION SCHEME
partition_scheme_name
 NEXT USED [ filegroup_name ] [ ; ]
```

where,

`partition_scheme_name` specifies the name of the partition scheme to be modified.

`filegroup_name` specifies the name of the new filegroup. This filegroup will accept a new partition that is created by modifying the partition function.

For example, the RateChngDate partition scheme created previously contains five filegroups. You want to create a new partition in the existing partitioned table. For this, you need to create a new filegroup and a file. After creating a new filegroup named Fifth and a file named File5 inside that filegroup, you can use the following statement to modify the partition scheme:

```
ALTER PARTITION SCHEME RateChngDate
 NEXT USED Fifth;
```

In the preceding statement, the Fifth filegroup will receive any additional partition of a partitioned table as a result of modifying the partition function.

Modifying a Partition Function

While creating a partition function, you declare the boundary values that are used to partition a table in logical units. You can modify a partition function by splitting or merging its boundary values. The ALTER PARTITION FUNCTION statement can be used to split one partition of a table into two partitions or merge two partitions of a table into one.

The syntax of the ALTER PARTITION FUNCTION statement is:

```
ALTER PARTITION FUNCTION
partition_function_name()
 {
 SPLIT RANGE(boundary_value)
 | MERGE RANGE(boundary_value)
 }
```

where,

`partition_function_name` specifies the name of the partition function to be modified.

`SPLIT RANGE(boundary_value)` adds a new partition to the partition function with the specified boundary value. The new boundary value must differ from the existing boundary ranges of the partition function. Based on the specified boundary value, one of the existing

ranges is split into two partitions. Of these two partitions, the one where the new boundary value resides is considered the new partition.

`MERGE RANGE(boundary_value)` drops a partition from the partition function and merges the specified boundary value with another existing boundary value of the partition function. The specified boundary value must be an existing boundary value. The filegroup held by the dropped partition is removed from the partition scheme if it is not used by the remaining partitions. The merged partition resides in the filegroup held by the boundary value to which it is merged.

For example, the following table lists the values in each partition for the RateChngDate partition function.

Partition 1	Partition 2	Partition 3	Partition 4	Partition 5
<1996-01-01	>=1996-01-01 to <2000-01-01	>=2000-01-01 to <2004-01-01	>=2004-01-01 to <2008-01-01	>=2008-01-01

The Values in Each Partition for the RateChngDate Partition Function

If you want to add one more partition to the RateChngDate partition function, you can use the following statement:

```
ALTER PARTITION FUNCTION RateChngDate
()
 SPLIT RANGE ('2002-01-01');
```

In the preceding statement, the boundary value, 2002-01-01 falls in partition 3. Therefore, this partition will be split into two partitions.

The following table lists the new values in each partition for the RateChngDate partition function.

Partition 1	Partition 2	Partition 3	Partition 4	Partition 5	Partition 6
< 1996-01-01	>=1996-01-01 to < 2000-01-01	>=2000-01-01 to < 2002-01-01	>=2002-01-01 to < 2004-01-01	>=2004-01-01 to < 2008-01-01	>=2008-01-01

The New Values in Each Partition for the RateChngDate Partition Function

Similarly, you can use the following statement to merge two partitions of the RateChngDate partition function:

```
ALTER PARTITION FUNCTION RateChngDate
()
 MERGE RANGE ('2008-01-01');
```

The preceding statement merges the partition with boundary value 2008-01-01 with the previous partition having boundary value 2004-01-01. Now, the records are stored in the partitions as shown in the following table.

Partition 1	Partition 2	Partition 3	Partition 4	Partition 5
<1996-01-01	>=1996-01-01 to <2000-01-01	>=2000-01-01 to <2002-01-01	>=2002-01-01 to <2004-01-01	>=2004-01-01

The Records Stored After Merging the Partitions

Assigning a Table as a Partition to an Existing Partitioned Table

In addition to modifying the partition scheme and partition function to make changes in the structure of a partitioned table, you can add a table as a partition to the existing partitioned table.

Table partitioning is performed to improve the performance of a data retrieval operation from large tables. However, insertion and updation of data in partitioned tables may be a time-consuming operation. Therefore, to insert or update data in a large partitioned table, you need to use another table that has the same structure as the partitioned table. The data is inserted or updated in that another table, which is then added as a partition to the partitioned table.

However, before adding the table as a partition to an existing partitioned table, you must ensure that:

- The source and target tables must have the same column structure and order.
- The nullability of the partitioning columns must match.
- The computed columns in both the tables must have the same syntax.
- The ROWGUID properties of the tables must be the same.
- The boundary values of the source partition must be within the boundary of the target partition.
- The source and target tables must have the same FOREIGN KEY constraints.

You can add a table as a partition to an existing partitioned table by using the ALTER TABLE statement. The syntax of the ALTER TABLE statement to add a table as a partition is:

```
ALTER TABLE Partitioned_Table_Name
SWITCH PARTITION
 Partition_Range TO New_Table_Name
```

where,

`Partitioned_Table_Name` specifies the name of the partitioned table to which the new table is to be added as a partition.

`Partition_Range` specifies the range of the partition to which the new table is to be added.

`New_Table_Name` specifies the name of the new table to be added as a partition.

For example, you need to add a new partition, 2012-01-01 to the RateChngDate partition function so that you can assign a new table to this partition. For this, you need to create a new filegroup and a file. In addition, you need to alter the partition scheme and partition function, and then create a table and add it as a partition.

To assign a table as a partition to an existing partitioned table, you need to perform the following tasks:

1. Create a new filegroup and file.
2. Alter the partition scheme.
3. Alter the partition function.

4. Create the table.
5. Add the table as a partition.

Creating a New Filegroup and File

Create a filegroup, Sixth and a file named File6.

Altering the Partition Scheme

Execute the following statement in the Query Editor window to alter the partition scheme:

```
ALTER PARTITION SCHEME RateChngDate
 NEXT USED Sixth;
```

Altering the Partition Function

Execute the following statement in the Query Editor window to alter the partition function:

```
ALTER PARTITION FUNCTION RateChngDate
()
 SPLIT RANGE ('2012-01-01');
```

Creating the Table

Execute the following statement in the Query Editor window to create a new table to be added to the new partition:

```
Create Table New_EmpPayHistPart
 (
 EmployeeID int,
 RateChangeDate datetime,
 Rate money,
 PayFrequency tinyint,
 ModifiedDate datetime
 )ON Sixth
```

Adding the Table as a Partition

You can use the following statement to add the New_EmpPayHistPart table to the new partition of the EmpPayHistPart partitioned table:

```
ALTER TABLE EmpPayHistPart SWITCH
PARTITION
 6 TO New_EmpPayHistPart
```

Modifying a Table

You need to modify tables when there is a requirement to add a new column, alter the data type of a column, or add or remove constraints on the existing columns. For example, AdventureWorks stores the leave details of all the employees in the EmployeeLeave table. According to the requirements, you need to add another column named ApprovedBy in the table to store the name of the supervisor who approved the leave of the employee. To implement this change, you can use the ALTER TABLE statement.

The syntax of the ALTER TABLE statement is:

```
ALTER TABLE [ database_name .
[ schema_name ] .] table_name
 {
 ALTER COLUMN column_name
 {
 [ NULL | NOT NULL ]
 }
 | [ WITH { CHECK | NOCHECK } ] [ADD
COLUMN <column_definition>][DROP
COLUMN <column_name>]
 {
 ADD CONSTRAINT constraint_name
constraint_type
```

where,

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

`schema_name` specifies the name of the schema to which the table belongs.

`table_name` is the name of the table that is to be altered. If the table is not in the current database, then the user needs to specify the database name and the schema name explicitly.

`ALTER COLUMN` specifies the name of the altered column.

`ADD COLUMN` specifies the name of the column to be added.

`column_definition` specifies the new column definition.

`CHECK | NOCHECK` specifies whether the existing data is to be checked for a newly added constraint or a re-enabled constraint.

`constraint_name` specifies the name of the constraint to be created and must follow the rules for the identifier.

`constraint_type` specifies the type of the constraint.

The following statement adds a column named ApprovedBy to the EmployeeLeave table:

```
USE AdventureWorks
 GO
 ALTER TABLE
HumanResources.EmployeeLeave
 ADD ApprovedBy VARCHAR(30) NOT NULL
 GO
```

In the preceding statement, the ApprovedBy column is

added that can store string values.

You can add a computed column to a table. A computed column contains values that are rather calculated than inserted. When you define a computed column, you need to include the expression that calculates the value for each row of the column. The values for the computed columns are not specified by using the INSERT statements. The expression for a computed column may refer to the other non-computed columns from the same table. For example, if the Orders table contains the UnitPrice column and the OrderQty column, the total cost of each order can be calculated as UnitPrice * OrderQty, as shown in the following code snippet:

```
ALTER TABLE Orders ADD TotalCost AS
UnitPrice * OrderQty
```

The values generated by the expression of the computed column are not stored within the database. Instead, the values are calculated every time they are required by a query. Therefore the computed column is virtual. However, you can specify that a computed column be persisted. The values of the persisted column are stored in the table. These values are recalculated whenever there is a change in any reference column. You can use the PERSISTED keyword to define the computed column as persisted, as shown in the following code snippet:

```
ALTER TABLE Orders ADD TotalCost AS
UnitPrice * OrderQty PERSISTED
```

A computed column must be persisted if you are adding a check constraint to it or if the column is marked as NOT NULL.

If you need to make the column persisted, it must be deterministic. This means that the database engine should be able to verify that the column will always produce the same result. The GetDate() function is not deterministic as its value changes every time the expression is evaluated.

The following statement modifies the Description column of the HumanResources.Project table:

```
USE AdventureWorks
 GO
 ALTER TABLE HumanResources.Project
 ALTER COLUMN Description varchar(100)
 GO
```

In the preceding statement, the size of the description column is increased to varchar(100).

The following statement drops the column named LeaveStatus from the EmployeeLeave table:

```
USE AdventureWorks
 GO
 ALTER                                TABLE
HumanResources.EmployeeLeave
 DROP COLUMN LeaveStatus
```

The following statement adds a constraint called chkRegion to the EmpTable table:

```
USE AdventureWorks
 GO
 ALTER TABLE EmpTable
 ADD CONSTRAINT chkRegion CHECK(Region
IN ('South America', 'North America',
'Middle East Asia'))
 GO
```

In the preceding statement, a CHECK constraint is added on the Region column.

While modifying a table, you can drop a constraint when it is not required. You can perform this task by altering the table by using the ALTER TABLE statement. The syntax to drop a constraint is:

```
ALTER TABLE [ database_name . [ schema_name ] . | schema_name . ]
table_name DROP CONSTRAINT
constraint_name
```

where,

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

`schema_name` specifies the name of the schema to which the table belongs.

`table_name` specifies the name of the table that contains the constraint to be dropped.

`constraint_name` specifies the name of the constraint to be dropped.

The following statement drops the default constraint, chkDefLeave of the EmployeeLeave table:

```
USE AdventureWorks
 GO
 ALTER                                TABLE
HumanResources.EmployeeLeave         DROP
CONSTRAINT chkDefLeave
 GO
```

In the preceding statement, the chkDefLeave constraint is dropped from the EmployeeLeave table.

All constraints defined on a table are dropped automatically when the table is dropped.

If the column, you are trying to add a constraint to, has some data in it, you can use CHECK | NOCHECK to specify whether the existing data needs to be checked with the added constraint.

Alter statement can be used to drop a column from a table definition. You cannot delete a row using the alter statement.

Renaming a Table

You can rename a table whenever required. The sp_rename stored procedure is used to rename the table. You can use sp_rename to rename any database object,

such as table, view, stored procedure, or function. The syntax of the sp_rename stored procedure is:

```
sp_rename old_name, new_name
```

where,

`old_name` is the current name of the object.

`new_name` is the new name of the object.

For example, the following statement renames the EmployeeLeave table:

```
USE AdventureWorks
 GO
 sp_rename
[HumanResources.EmployeeLeave],
[HumanResources.EmployeeVacation]
 GO
```

You can also rename a table by right-clicking the Tables folder under a specific database in the Object Explorer window, and then selecting the Rename option from the pop-up menu.

After a table is created, you may need to view the details of the table. Details of the table include the column names and the constraints. For this purpose, you can use the sp_help statement. The syntax of the sp_help statement is:

```
sp_help [table name]
```

Dropping a Table

At times, when a table is not required, you need to delete it. A table can be deleted along with all the associated database objects such as its index, triggers, constraints, and permissions. You can delete a table by using the DROP TABLE statement. The syntax of the DROP TABLE statement is:

```
DROP TABLE [ database_name . [ schema_name ] .] table_name
```

where,

`database_name` specifies the name of the database where the table is created.

`schema_name` specifies the name of the schema to which the table belongs.

`table_name` specifies the name of the table that needs to be dropped.

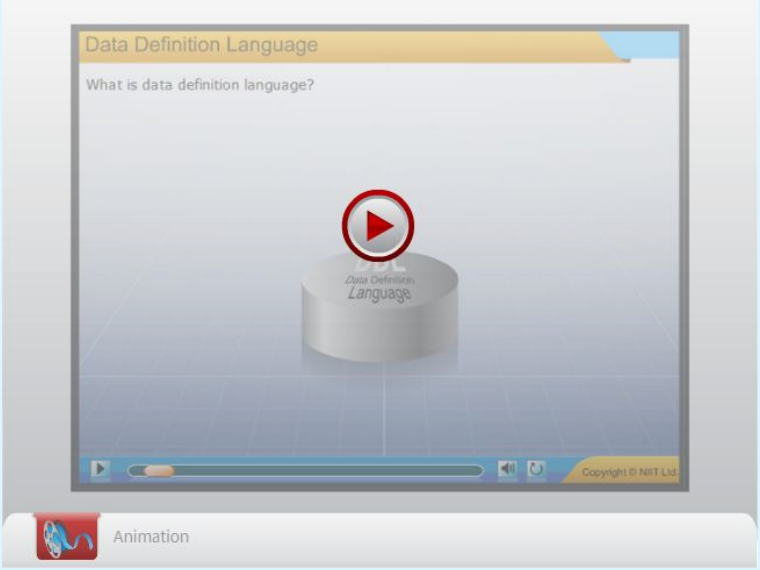
When a table is deleted, any other database object referenced by the table needs to be deleted explicitly. This should be done before deleting the table. This is because if violations occur in the rule of referential integrity while deleting a table, then an error occurs that restricts you from deleting the table. Therefore, if your table is referenced, you must delete the referenced table or the referenced constraint, and then delete the table.

For example, the Employee table in the HumanResource schema contains EmployeeID as its primary key. The EmployeeVacation table under the same schema contains EmployeeID as its foreign key and is referenced with the EmployeeID column of the Employee table. Therefore, when you want to delete the Employee table, you first need to delete the EmployeeVacation table.

The following statement deletes the EmployeeVacation table:

```
USE AdventureWorks
 DROP TABLE
HumanResources.EmployeeVacation
```

You can also delete a table by right-clicking the Tables folder under a specific database in the Object Explorer window, and then selecting the Delete option from the pop-up menu.



Activity 4.1: Managing Tables

Activity 4.2: Creating a Partitioned Table

Summary

In this chapter, you learned that:

- A database is a repository of information that contains data in an organized way.
- The master database records all the server-specific configuration information, including authorized users, databases, system configuration settings, and remote servers.
- The tempdb database is a temporary database that holds all the temporary tables and stored procedures.

- The model database acts as a template or a prototype for new databases.
- The msdb database supports the SQL Server Agent. SQL Server Agent includes features that schedule periodic activities of SQL Server.
- The Resource database is a read-only database that contains all the system objects that are included with SQL Server.
- The user-defined databases are created by the users to store data for client/server applications.
- A database consists of the following types of files:
  - Primary data file
  - Secondary data file
  - Transaction log file
- A database must consist of a primary data file and one transaction log file.
- The CREATE DATABASE statement is used to create a database, which also includes determining the name of the database, the size of the database, and the files used to store data in the database.
- The DROP DATABASE statement is used to delete a database.
- Tables are used to store data.
- The CREATE TABLE statement is used to create a table.
- Data integrity is enforced to keep the data in a database accurate, consistent, and reliable. It is broadly classified into the following categories:
  - **Entity integrity**: Ensures that each row can be uniquely identified by an attribute called the primary key.
  - **Domain integrity**: Ensures that only a valid range of values is allowed to be stored in a column.
  - **Referential integrity**: Ensures that the value of the foreign key match the value of the corresponding primary key.
  - **User-defined integrity**: Refers to a set of rules specified by a user, which do not belong to the entity, domain, and referential integrity categories.
- Constraints define rules that must be followed to maintain consistency and correctness of data.
- A primary key constraint is defined on a column or a set of columns whose values uniquely identify rows in a table.
- The unique constraint is used to enforce uniqueness on non-primary key columns.
- A foreign key constraint associates one or more columns (the foreign key) of a table with an identical set of columns (a primary key column in another table) on which a primary key constraint has been defined.
- A check constraint enforces domain integrity by restricting the values to be inserted in a column. The IN, LIKE, and BETWEEN keywords are used to define the check constraint.
- A default constraint can be used to assign a constant value to a column, and the user need not insert values for such a column.
- A rule provides a mechanism for enforcing domain integrity for columns or user defined data types.
- User-defined data types are custom data types defined by the users with a custom name.
- A partitioned table is created to manage the data and improve the query performance.
- You can modify a partition table by:
  - Modifying a partition scheme and partition function.
  - Assigning a table as a partition to an existing partitioned table.
- You can modify a partition function by splitting or merging its boundary values.
- Before adding the table as a partition to an existing partitioned table, you must ensure that:
  - The source and target tables must have the same column structure and order.
  - The nullability of the partitioning columns must match.
  - The computed columns in both the tables must have the same syntax.
  - The ROWGUID properties of the tables must be the same.
  - The boundary values of the source partition must be within the boundary of target partition.
  - The source and target tables must have the same FOREIGN KEY constraints.
- The ALTER TABLE statement is used to modify a table.
- The DROP TABLE statement is used to delete a table.

Reference Reading

Managing Databases

Reference Reading: Books	*Reference Reading: URLs*
Microsoft® SQL Server® 2012 Pocket Consultant by William R. Stanek *Learning SQL by Alan Beaulieu*	*http://msdn.microsoft.com/en-us/library/cc645585.aspx* *http://msdn.microsoft.com/en-us/library/ms179422.aspx*

Managing Tables

Reference Reading: Books	***Reference Reading: URLs***
SQL Server 2012 T-SQL Recipes: A Problem-Solution Approach by Jason Dewson	*http://msdn.microsoft.com/en-us/library/ms174979.aspx* *http://msdn.microsoft.com/en-us/library/ms190273.aspx*

Glossary

A

Alias

A temporary intra-query substitute for a table name or column name.

APPLY Operator

The operator used to combine the result set of two queries such that for each row of the first query, the second query is evaluated to determine if any rows are returned.

Arithmetic Operators

The operators used to perform mathematical operations, such as addition, subtraction, division, and multiplication, on numeric columns or on numeric constants.

Attribute

A column in a table.

B

Batch

A group of SQL statements submitted together to the SQL Server for execution.

Business Tier

The middle tier in a three-tier architecture where the business logic is implemented.

C

Cartesian Product

A binary operation resulting in the combination of all rows of one table with all rows of another table.

Client Tier

The tier that handles the User Interface (UI) in a client-server architecture.

Common Language Runtime (CLR)

The core runtime environment in the Microsoft .NET Framework on which the applications run.

Common Table Expression (CTE)

A temporary named result set defined within the execution scope of a single SQL statement.

Concatenation Operator

An operator that is used to combine string expressions.

Constant

An unvarying value used in a query.

Constraint

A restriction placed on a value in a database used to increase data integrity.

Cross Join

A join that combines each row from one table with each row of the other table.

D

Data

Recorded facts pertaining to entities.

Data Definition Language (DDL)

A language used to define the internal schema and conceptual schema in a database.

Data Integrity

The concept of maintaining consistency and completeness of data.

Data Manipulation Language (DML)

A language used to manipulate data using INSERT, UPDATE, and DELETE statements.

Data Warehouse

An application with a computer database that collects, integrates, and stores an organization's data with the aim of producing accurate and timely management of information and support for analysis techniques.

Database

A collection of logically associated or related data.

Default

A value assigned to data when no value is supplied.

Dialog

A bi-directional communication between two services.

Domain

The set of all possible values that a column value can have.

E

Equi Join

A query that combines tables with the help of a foreign key and displays all the columns from both the tables. Is similar to an inner join.

Extensible Markup Language (XML)

A universal language used to generically identify data that will be shared.

F

Filegroup

A collection of database files grouped together for allocation and administration purposes.

I

Inner Query

A subquery that is contained in another query. The result of the inner query is used an input for the condition specified in the outer query.

J

Join

An operation used to combine related rows from two tables into one table based on a logical comparison of column values.

L

Literals

The string values, enclosed in single quotes and added to the SELECT statement, are printed in a separate column as they are written in the SELECT list.

Local Variables

Variables declared in a batch that can be used in any statement inside the batch.

M

Managed Code

The code written in any of the .Net supported languages that runs within the scope of the SQL CLR.

Managed Database Object

A database object that can be created using CLR integration in any of the .NET-supported languages and then can be embedded in the database.

N

.NET Framework

A software development platform focused on rapid application development, platform independence, and network transparency.

NULL

A value given to a data item when the result is unknown.

O

Outer Join

An join operation that displays the result set containing all the rows from one table and the matching rows from another table.

Q

Query

An SQL instruction used to retrieve data from one or more tables or views. Queries begin with the SQL keyword SELECT.

Queue

A container used to stores messages.

R

Referential Integrity

The property that guarantees the values from one column depending on the values from another column are present in the other column.

Result Set

Output of an SQL statement.

Row

A horizontal slice of a table. A row is also known as a 'tuple' and at times is called a 'record'. However, a record usually refers to a physical representation of data and a row refers to a logical representation.

S

Schema

A design of the database typically using an entity relationship diagram.

Self Join

A join condition where a table is joined with itself.

Server Tier

The part that implements the application logic and manages the input data based on the business rules in a client server architecture.

Structured Query Language (SQL)

A language for defining the structure and processing of a relational database.

Subquery

The inner query within the outer (main) query; usually one SELECT query within another SELECT query.

U

Unmanaged Code

The code that is developed without considering the conventions and requirements of the common language runtime.

Certification Mapping

The content in this course partially covers the objectives of Microsoft Exam 70-461 (Microsoft SQL Server 2012, Database Development).
The following table lists the relevant exam objectives and the corresponding chapter numbers and section names.

Exam Objective	***Chapter***	***Section***
Create Database Objects (24%)		
Create and alter tables using T-SQL syntax(simple statements).	*4*	*Managing Tables*
Create and alter views(simple statements).	*6*	*Creating and Managing Views*
Create and modify constraints(simple statements).	*4*	*Managing Tables*
Create and alter DML triggers.	*8*	*Implementing Triggers*
Working with Data (27%)		
Query data by using SELECT statements.	*2*	*Retrieving Data*
Implement subqueries	*2, 3*	*Querying Data by Using Subqueries, Managing Result Sets, Summarizing and Grouping Data*
Implement data types.	*2, 4, 5*	*Retrieving Data, Managing Tables, Manipulating Data by Using DML Statements*
Implement aggregate queries.	*2, 4*	*Summarizing and Grouping Data, Using Functions to Customize the Result Set, Managing Tables*
Query and Manage XML data.	*5*	*Manipulating XML Data*
Modify Data (24%)		
Create and alter stored procedures (simple statements).	*7*	*Implementing Stored Procedures*
Modify data by using INSERT, UPDATE, and DELETE statements.	*5*	*Manipulating Data by Using DML Statements*
Combine datasets.	*5*	*Manipulating Data by Using DML Statements*
Work with functions.	*7*	*Implementing Functions*
Troubleshoot and Optimize (25%)		
Optimize Queries.	*9, 6, 7*	*Optimizing Performance, Creating and Managing Indexes, Implementing Stored Procedures*
Manage transactions.	*8*	*Implementing Transactions*
Implement error handling.	*7*	*Implementing Batches*
Evaluate the use of row-based operations vs set-based operations.	*7*	*Implementing Stored Procedures*

Certification Mapping

To prepare for the exam, you need to refer to the Student Guide as well as the Activity Book.