Microsoft

OFFICIAL MICROSOFT LEARNING PRODUCT

10985C

Introduction to SQL Databases

Product Number: 10985C

Part Number (if applicable): X21-64449

Released: 11/2017

Insert EULA pdf in final pdf here

Welcome!

Thank you for taking our training! We've worked together with our Microsoft Certified Partners for Learning Solutions and our Microsoft IT Academies to bring you a world-class learning experience—whether you're a professional looking to advance your skills or a student preparing for a career in IT.

- **Microsoft Certified Trainers and Instructors**—Your instructor is a technical and instructional expert who meets ongoing certification requirements. And, if instructors are delivering training at one of our Certified Partners for Learning Solutions, they are also evaluated throughout the year by students and by Microsoft.

- **Certification Exam Benefits**—After training, consider taking a Microsoft Certification exam. Microsoft Certifications validate your skills on Microsoft technologies and can help differentiate you when finding a job or boosting your career. In fact, independent research by IDC concluded that 75% of managers believe certifications are important to team performance[1]. Ask your instructor about Microsoft Certification exam promotions and discounts that may be available to you.

- **Customer Satisfaction Guarantee**—Our Certified Partners for Learning Solutions offer a satisfaction guarantee and we hold them accountable for it. At the end of class, please complete an evaluation of today's experience. We value your feedback!

We wish you a great learning experience and ongoing success in your career!

Sincerely,

Microsoft Learning
www.microsoft.com/learning

Microsoft | Learning

[1] *IDC,* Value of Certification: Team Certification and Organizational Performance, *November 2006*

Acknowledgements

Microsoft Learning would like to acknowledge and thank the following for their contribution towards developing this title. Their effort at various stages in the development has ensured that you have a good classroom experience.

Martin Ellis – Content Developer

Martin is a technical author, consultant, and SQL Server subject matter expert. As a Microsoft Certified Trainer (MCT), he has extensive experience of delivering Microsoft training courses on technologies including SQL Server and Windows Server. He has authored and co-authored numerous Microsoft courses for the SQL Server 2008 R2, SQL Server 2012, SQL Server 2014, and SQL Server 2016 curriculums, and is the author of several white papers and other technical documents. Martin also provides consultancy services for organizations in the planning, implementation, optimization, and maintenance of SQL Server solutions.

Simon Butler – Content Developer

Simon Butler FISTC is a highly-experienced Senior Technical Writer with nearly 30 years' experience in the profession. He has written training materials and other information products for several high-profile clients. He is a Fellow of the Institute of Scientific and Technical Communicators (ISTC), the UK professional body for Technical Writers/Authors. To gain this, his skills, experience and knowledge have been judged and assessed by the Membership Panel. He is also a Past President of the Institute and has been a tutor on the ISTC Open Learning course in Technical Communication techniques. His writing skills are augmented by extensive technical skills gained within the computing and electronics fields.

Geoff Allix – Technical Reviewer

Geoff Allix is a Microsoft SQL Server subject matter expert and professional content developer at Content Master—a division of CM Group Ltd. As a Microsoft Certified Trainer, Geoff has delivered training courses on SQL Server since version 6.5. Geoff is a Microsoft Certified IT Professional for SQL Server and has extensive experience in designing and implementing database and BI solutions on SQL Server technologies, and has provided consultancy services to organizations seeking to implement and optimize database solutions.

Lin Joyner – Technical Reviewer

Lin is an experienced Microsoft SQL Server developer and administrator. She has worked with SQL Server since version 6.0 and previously as a Microsoft Certified Trainer, delivered training courses across the UK. Lin has a wide breadth of knowledge across SQL Server technologies, including BI and Reporting Services. Lin also designs and authors SQL Server and .NET development training materials. She has been writing instructional content for Microsoft for over 15 years.

Contents

About This Course

This section provides a brief description of the course, audience, suggested prerequisites, and course objectives.

Course Description

This three-day instructor-led course is aimed at people looking to move into a database professional role or whose job role is expanding to encompass database elements. The course describes fundamental database concepts including database types, database languages, and database designs.

Audience

The primary audience for this course is people who are moving into a database role, or whose role has expanded to include database technologies.

Student Prerequisites

This is a foundation level course and therefore only requires general computer literacy.

Course Objectives

After completing this course, students will be able to:

- Describe key database concepts in the context of SQL Server 2016.
- Describe database languages used in SQL Server 2016.
- Describe data modelling techniques.
- Describe normalization and denormalization techniques.
- Describe relationship types and effects in database design.
- Describe the effects of database design on performance.
- Describe commonly used database objects.

Course Outline

The course outline is as follows:

- Module 1: 'Introduction to Databases' introduces key database concepts in the context of SQL Server 2016
- Module 2: 'Data Modelling' describes a number of data modelling techniques.
- Module 3: 'Normalization' introduces the concept of normalization and describes normalization and denormalization techniques.
- Module 4: 'Relationships' describes the different types of relationship and their effects in database design
- Module 5: 'Performance' describes the effects of database design on performance.
- Module 6: 'Database Objects' introduces commonly used database object.

Course Materials

The following materials are included with your kit:

- ***Course Handbook:*** a succinct classroom learning guide that provides the critical technical information in a crisp, tightly-focused format, which is essential for an effective in-class learning experience.
  - **Lessons**: guide you through the learning objectives and provide the key points that are critical to the success of the in-class learning experience.
  - **Labs**: provide a real-world, hands-on platform for you to apply the knowledge and skills learned in the module.
  - **Module Reviews and Takeaways**: provide on-the-job reference material to boost knowledge and skills retention.
  - **Lab Answer Keys**: provide step-by-step lab solution guidance.

 **Additional Reading: Course Companion Content on the http://www.microsoft.com/learning/en/us/companion-moc.aspx Site*:*** searchable, easy-to-browse digital content with integrated premium online resources that supplement the Course Handbook.

- **Modules**: include companion content, such as questions and answers, detailed demo steps and additional reading links, for each lesson. Additionally, they include Lab Review questions and answers and Module Reviews and Takeaways sections, which contain the review questions and answers, best practices, common issues and troubleshooting tips with answers, and real-world issues and scenarios with answers.
- **Resources**: include well-categorized additional resources that give you immediate access to the most current premium content on TechNet, MSDN®, or Microsoft® Press®.

 **Additional Reading: Student Course files on the http://www.microsoft.com/learning/en/us/companion-moc.aspx Site**: includes the Allfiles.exe, a self-extracting executable file that contains all required files for the labs and demonstrations.

- ***Course evaluation:*** at the end of the course, you will have the opportunity to complete an online evaluation to provide feedback on the course, training facility, and instructor.
- To provide additional comments or feedback on the course, send email to mcspprt@microsoft.com. To inquire about the Microsoft Certification Program, send an email to mcphelp@microsoft.com.

Virtual Machine Environment

This section provides the information for setting up the classroom environment to support the business scenario of the course.

Virtual Machine Configuration

In this course, you will use Microsoft® Hyper-V™ to perform the labs.

Note: At the end of each lab, you must revert the virtual machines to a snapshot. You can find the instructions for this procedure at the end of each lab

The following table shows the role of each virtual machine that is used in this course:

Virtual machine	Role
10985C-MIA-DC	MIA-DC1 is a domain controller.
10985C-MIA-SQL	MIA-SQL has SQL Server installed
MSL-TMG1	TMG1 is used to access the internet

Software Configuration

The following software is installed on the virtual machines:

- Windows Server 2016
- SQL2017
- Visual Studio 2017
- SharePoint 2017

Course Files

The files associated with the labs in this course are located in the D:\Labfiles folder on the 10985c-MIA-SQL virtual machine.

Classroom Setup

Each classroom computer will have the same virtual machine configured in the same way.

Course Hardware Level

To ensure a satisfactory student experience, Microsoft Learning requires a minimum equipment configuration for trainer and student computers in all Microsoft Learning Partner classrooms in which Official Microsoft Learning Product courseware is taught.

- Intel Virtualization Technology (Intel VT) or AMD Virtualization (AMD-V) processor
- Dual 120-gigabyte (GB) hard disks 7200 RM Serial ATA (SATA) or better
- 16 GB of random access memory (RAM)
- DVD drive
- Network adapter
- Super VGA (SVGA) 17-inch monitor

- Microsoft mouse or compatible pointing device
- Sound card with amplified speakers

Additionally, the instructor's computer must be connected to a projection display device that supports SVGA 1024×768 pixels, 16-bit colors.

Module 1

Introduction to Databases

Contents:

Module Overview

There are many different types of database, and many languages that you can use to create, manage, and query them. This module describes some of the most common types of database, outlines some key concepts for working with relational databases, and introduces the languages that you can use when working with databases in Microsoft® SQL Server®. The module also explains how you can use Transact-SQL to access and manipulate data.

Objectives

After completing this module, you will be able to:

- Describe the characteristics of relational databases.
- Describe several other common types of database and storage methods.
- Describe the process of data analysis.
- Describe the languages that you can use in SQL Server.

Lesson 1
Introduction to Relational Databases

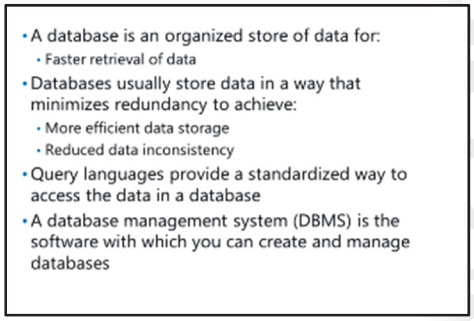
The term database covers a range of different types of storage structures, each of which has its own distinct set of characteristics. This lesson describes what a database is and introduces the relational database, which is the most widely implemented type of database.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain what a database is.
- Describe the key characteristics of relational databases.
- Describe table relationships in relational databases.
- Explain the concept of normalization in relational databases.

What Is a Database?



At the simplest level, a database is just an organized store of data. Organizing data in a logical and consistent way makes it faster and easier to access the data when you need it. Just as it is easier to locate a specific book on a set of shelves when the books are organized alphabetically, or by genre, or by a combination of both, the organizational principles that databases use make data retrieval much more efficient. However, there is no single "correct" way to organize data in a database—and there are several different types of database in use today.

For most people, probably the most familiar type of database is the relational database. A relational database consists of tables, each of which contains rows of data organized by columns, in an arrangement similar to a spreadsheet. Each table is related to one or more of the other tables in the database—for example, a table that contains customer information might relate to a table that contains information about orders. This relationship is based on the real-world fact that customers place orders. You will learn more about relational databases in subsequent topics. Other common types of database include hierarchical databases, such as the Windows® registry database, object-oriented databases, and column-oriented databases.

Redundancy

In addition to making it faster and easier to retrieve data, databases also make the storage of data more efficient by reducing data redundancy. The term data redundancy refers to the storage of the same piece of data multiple times. For example, imagine a small company that uses a spreadsheet to keep track of its orders, including details about the customers who placed the orders.

The spreadsheet contains one row for each order, and includes columns called Customer Name, Customer Email, and Customer Address. It is likely that some of the customers will have placed multiple orders, so the spreadsheet might contain the same customer information several times. Redundant data like this can significantly increase the size of a database, making storage more expensive, and the database potentially more difficult to manage.

Additionally, storing data repeatedly can lead to data inconsistency because a single data value might be stored differently in two or more locations. This might be because of spelling mistakes or because of the use of different terms to describe the same thing. For example, in the spreadsheet described above, a customer's address might include the state as "WA" in one row and as "Washington" in another.

It is important that data is consistent because this will affect the results of queries that you issue against the database. Inaccurate query results can have serious consequences because businesses rely on data to guide decision-making processes. Databases can reduce redundancy by storing different types of data in different tables, so that a single piece of information, such as a customer's address, can be stored just once instead of many times.

Accessing data in databases

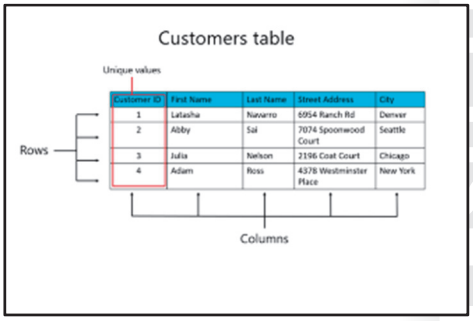
To access the data in databases, you have to be able to query the database, most commonly by using a client application. There are various languages that you can use to access and manipulate the data in a database, and the type of language will depend on the type of database you are querying. For relational databases, the best-known language is Structured Query Language (SQL). Other languages that support data access include XQuery and Multidimensional Expressions (MDX).

Database management systems

A database management system (DBMS) is the software that you install on a server to create and manage databases. A DBMS can typically support many databases. Well-known examples of a DBMS include Microsoft SQL Server, MySQL, and Oracle. Most DBMSs provide comprehensive administrative functionality, such as the ability to back up databases and to monitor database health and performance. A DBMS typically also supports security through features such as authentication (to establish the identity of the user attempting to connect to the database), authorization (to define what actions an authenticated user is permitted to perform), and data encryption.

Relational Database Fundamentals

Relational databases are based on a theoretical data storage model that dates back to the late 1960s, when Edgar F. Codd proposed it as a more efficient way to store data. This is probably the most familiar and widely implemented type of large database in the world today. Examples of relational DBMSs include SQL Server, Sybase, and Oracle.



Tables, rows, and columns

Relational databases store data in tables, and a single database usually contains multiple tables. Each table contains a subset of the data in the database. For example, in a database that stores transactional data—such as an order processing database—you might find tables called Customers, Orders, Products, Order Details, and Shippers. Each of these tables would contain only data about a specific entity—for example, the Customers table would contain data about customers; the Orders table would store information about orders.

Tables store data in horizontal rows, and each row represents a specific instance of the entity that the table represents. For example, in the Orders table, each row would represent a single order, and in the Products table, each row would represent a single product. Tables are divided vertically into columns, and each column contains a specific type of information. For example, in the Customers table, there might be columns called First Name**,** Last Name**,** Email Address, and Phone Number; each row would contain a single value for each of these columns.

In some books and other sources about relational databases, you might come across the words "relation", "tuple", and "attribute". These terms date back to the time when Codd first published his ideas about relational data models. They relate more to the conceptual description of a relational model rather than to an actual implementation of it.

You will learn more about this in Module 2 of this course. Most modern DBMSs implement relations as tables, tuples as rows, and attributes as columns. However, it is worth remembering these other terms because you are likely to some across them occasionally. For example, the data values in a column are sometimes referred to as attribute values. You may also come across the term "record", which usually means the same as "row", and "field", which usually means the same as "column".

Ensuring uniqueness by using a primary key

In most cases, a table will include a column that contains a unique value for each row—this column makes it possible to unambiguously identify each row in the table. The values in this column may be exist in the real world—such as a product number in the Products table—or it may be a column that you add to the table solely to provide an identifier for each row. For example, you might add a column called Customer ID to the Customers table. The value in this column for the first row might be 1, with subsequent rows containing the values 2, 3, 4, and so on. The following table illustrates this:

Customer ID	Name	Address	Phone Number
1	Latasha Navarro	6954 Ranch Rd	555-0123
2	Julia Nelson	2196 Coat Court	555-0126
3	Adam Ross	7378 Westminster Place	555-0195

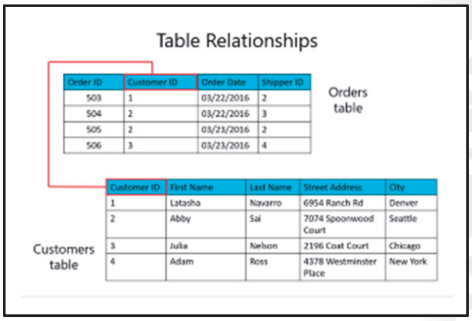
The numbers in the Customer ID column do not have any meaning in the real world; they simply serve as identifiers within the database. It is important to ensure that there is no duplication of the values in this column. For example, if the Products table contained two rows with the same product number, you would not be able to tell the two rows apart, and this could quickly cause problems with data consistency.

Imagine that a data analyst creates a report that sums the sales revenue for each product, based on the product number; the report would be inaccurate because the values for two distinct products would be summed together. To avoid this kind of problem, you add a primary key constraint to the column that contains the unique identifiers.

A constraint restricts the actions that you can perform when adding or modifying data in a table—there are several types of constraint that you can create to help to prevent data inconsistency. When you define a primary key constraint on a column, it prevents the addition of duplicate values to that column, so ensuring that every value is unique.

Tables in Relational Databases

Tables in a relational database are related to each other. You can describe this relationship in terms of the real-world process that the database is modelling. For example, customers place orders for products, and each order can contain multiple items. A transactional database might contain a table that represents each of the steps in this process—a Customers table to contain customer data; an Orders table to contain order data; a Products table to contain product data; and an Order Details table to contain data about the actual items in the order.



Additionally, if the company uses third-party shippers to deliver its orders to customers, there might be a Shippers table that contains information about the shippers. The Orders table would not contain extensive information about each customer, because it is the Customers table that stores this information. Instead, the Orders table might include a column called Customer ID that contains values that identify the customer who placed each order. For each row, this value must match one of the values in the unique Customer ID column in the Customers table. The following example tables illustrate this:

Customers Table

Customer ID	Name	Address	Phone Number
1	Latasha Navarro	6954 Ranch Rd	555-0123
2	Julia Nelson	2196 Coat Court	555-0126
3	Adam Ross	7378 Westminster Place	555-0195

Orders table

Order ID	Customer ID	Order Date	Shipper ID
503	1	03/22/2016	4
504	2	03/22/2016	2
505	2	03/23/2016	3

This arrangement helps to minimize data redundancy because, instead of storing a customer's name, address, and other details in the Orders table every time they place an order, the database stores a customer's details only once, as a row in the Customers table. When a customer places an order, this adds a new row to the Orders table, including a Customer ID that already exists in the Customers table.

Foreign keys

An order in the Orders table that did not have a corresponding Customer ID in the Customers table would be impossible to fulfill because there would be no delivery address shown, and no email address or phone number to contact the customer. To ensure that it is not possible to add Customer ID values to the Orders table that do not exist in the Customers table, you can use another type of constraint called a foreign key constraint.

When you add a foreign key constraint to a column, the constraint checks any new values that you add to the column to make sure that the values already exist in a second column—which is called the referenced column. In this example, you would create a foreign key on the Customer ID column in the Orders table that references the Customer ID column in the Customers table. The constraint would prevent the addition of a row to the Orders table if it contained a Customer ID value that was not present in the Customers table.

Unlike a primary key constraint, a foreign key constraint does not enforce uniqueness. It would make little sense to restrict a customer's Customer ID to only one appearance in the Orders table, because this would mean that they could only place a single order! The relationship between a primary key column and a foreign key column is called a one-to-many relationship. This means that the values in a foreign key column can each appear multiple times, but the column that the foreign key references, usually a primary key column, must contain unique values.

To complete the example used in this topic, you would also create a primary key constraint on the Order ID column of the Orders table, and on the unique identifier column of the other tables. You would then add any foreign key constraints to the tables as required.

Demonstration: Exploring a Relational Database

In this demonstration, you will see:

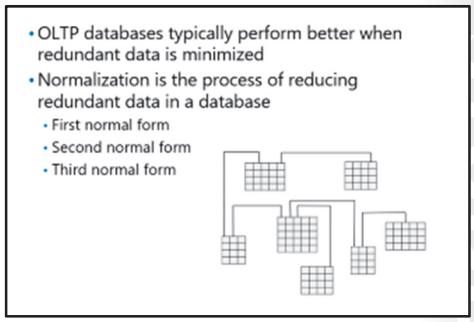
- How to create a database diagram.
- How to use a database diagram to view tables, primary key constraints, and foreign key constraints.

Demonstration Steps

1. Start the 10985C-MIA-DC and 10985C-MIA-SQL virtual machines, and then log on to 10985C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. In the **D:\Demofiles\Mod01** folder, right-click **Setup.cmd**, and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**, and wait for setup to complete.
4. Open Microsoft SQL Server Management Studio, and then connect to the **MIA-SQL** instance of the database engine by using Windows Authentication.
5. In Object Explorer, expand **Databases**, expand **AdventureWorks2016**, right-click **Database Diagrams**, and then click **New Database Diagram**.
6. If the **Microsoft SQL Server Management Studio** dialog box appears asking if you wish to create support objects for database diagramming, click **Yes**.
7. In the **Add Table** dialog box, press and hold down the CTRL key, click **Customer (Sales)**, click **SalesOrderDetail (Sales)**, click **SalesOrderHeader (Sales)**, click **ShipMethod (Purchasing)**, click **Add**, and then click **Close**.
8. Review the tables and note the following points:
    a. The **SalesOrderHeader (Sales)** table contains the **SalesOrderID**, which is the primary key column.
    b. The **SalesOrderDetail (Sales)** table also contains a **SalesOrderID** column.
9. In the **SalesOrderDetail (Sales)** table, right-click the **SalesOrderID** column, and then click **Properties**.
10. In the **Properties** window, click the **Description** field, and then click the ellipsis button (**...**).

11. In the **Description Property** dialog box, note that the column is a primary key column, and that there is a foreign key that references the **SalesOrderID** column in the **SalesOrderHeader** column, and then click **Cancel**.
12. Click the line between the **Customer (Sales)** table and the **SalesOrderHeader (Sales)** table. This line represents a foreign key relationship.
13. In the **Properties** window, click **Description**, and then click the ellipsis button (**…**).
14. In the **Description Property** dialog box, note that the foreign key references the **CustomerID** column in the **Customer (Sales)** table, and then click **Cancel**.
15. Close the database diagram window, and do not save any changes. Close SQL Server Management Studio.

Introduction to Normalization



The simple example order processing database described in the previous topics is an example of an operational or online transactional processing (OLTP) database. In an OLTP database like this, you expect there to be frequent inserts of new rows into the various tables, as customers place orders, and updates to rows as, for example, customers update their personal details.

By designing the database to minimize redundancy, as explained here, you help to ensure that each insert operation is as quick as possible. For example, when a customer places a new order, it updates only the tables that contain order data, leaving the other tables untouched. Furthermore, the insert operation does not have to add redundant data, such as the customer details, because they are already present in the database.

Normalization and normal form

The process of minimizing data redundancy in relation databases is called normalization. A database designer would create a plan or schema for the normalized database at the design stage, before the creation of the database. There is a formal process, which consists of multiple steps that you can follow to normalize a database.

These steps are called first normal form, second normal form, third normal form, and so on. Each step addresses specific types of data redundancy issues, and the steps are cumulative, meaning that each encompasses the previous one. For example, to implement second normal form in a database, you first have to implement first normal form.

- First normal form deals with separating related data into tables, and ensuring that each table has a unique identifier, as previously described. It additionally involves eliminating columns that contain repeating data. For example, a Customers table that contained columns called Telephone Number 1 and Telephone Number 2 would not be in first normal form because the two columns contain the same type of data.
- Second normal form involves complying with first normal form, in addition to separating data that might repeat across rows into different tables. You saw an example of this in the previous topic, which described storing customers' details in a separate table to avoid repeating them in the Orders table.

- Third normal form involves complying with second normal form, while also eliminating columns that contain values that do not fully depend on the primary key. For example, a column in an Order Details table called Total that contains the total order values as calculated by multiplying the unit price of items by the quantity ordered. Because the column contains values that are derived from the values in other columns, it is not fully dependent on the primary key and so the database is not in third normal form.

There are additional levels of normal form. You will learn more about these, and see more examples of normalization in Module 3 of this course.

Check Your Knowledge

Question	
Which of the following describes the purpose of a foreign key?	
Select the correct answer.	
	To ensure that values in a column are unique.
	To reduce data redundancy.
	To check that the values added to a column are valid by referencing a different column, usually a column in a different table.

Lesson 2
Other Types of Databases and Storage

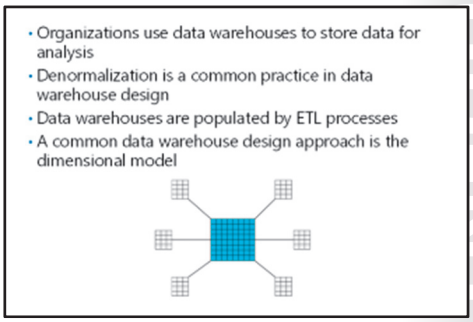
Although the OLTP relational database is a very common type of database, it is one of many that you might encounter. It is useful to be familiar with these types of database, even if you do not work with them on a day-to-day basis. This lesson describes several different types of database and outlines their key features.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the key features of data warehouses.
- Describe the key features of online analytical processing (OLAP) databases.
- Describe the key features of hierarchical databases.
- Describe the key features of column-oriented databases.
- Describe the key features of NoSQL databases and Hadoop.

Data Warehouses



The previous topics described relational databases in an OLTP context, but it is also possible to use relational databases to create data warehouses. A data warehouse is a very large database that typically contains historical data instead of current transactional data. For example, an organization might use an OLTP database to process and store transactions for the current month, and a separate data warehouse to store transactions from previous months and years.

Organizations can use this historical data to help them understand how the company is performing, or to plan future strategies. For example, you can use a data warehouse to analyze sales performance over previous periods, to assess the success of marketing campaigns, or to predict sales trends.

Redundant data and denormalization

A relational data warehouse contains related tables, in the same way as on OLTP relational database does. In an OLTP database, you create tables to minimize redundancy and optimize the performance of transactional data operations such as inserts, updates, and deletes. The down side of a design like this is that, when you read data back from the database, you have to locate the required data in multiple tables and correlate it so that it all matches up. Although the query engine component of the DBMS does this work for you, it is a relatively time-consuming process. However, because most of the workload for an OLTP database is transactional and not read-based, this is not usually a problem.

In a data warehouse, everyday transactional activity is usually rare to non-existent. Instead of inserting rows one at a time, as is the case in an OLTP database, you typically bulk-load data into a data warehouse on a scheduled basis—for example, once a week or once a month. In between these data loads, the primary activity is read-based, when data analysts query the data warehouse. Consequently, you create tables and relationships in a data warehouse with the aim of optimizing read performance instead of for transactional performance.

There are various recognized methods for designing data warehouses. Amongst the most commonly used are Ralph Kimball's dimensional model and Bill Inmon's normalized model. In the dimensional model, you design the data warehouse so that it will contain more redundant data than an OLTP database. The process of designing a database to include the storage of redundant data in this way is called denormalization.

Populating a data warehouse

The data in a data warehouse can come from many different sources, including spreadsheets, and proprietary systems such as accounting or customer relationship management (CRM) systems. One very common data source is a company's own OLTP databases. Organizations usually move data from OLTP databases or other source systems to the data warehouse, as part of the extract, transform, and load (ETL) process.

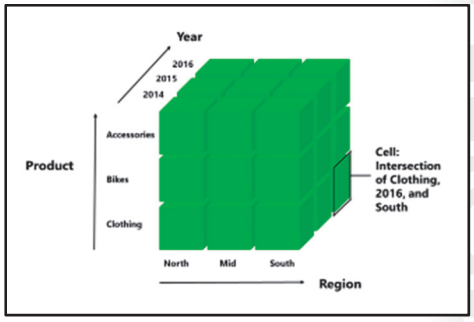
The ETL process implements the necessary changes to the data to get it ready for the denormalized data warehouse. These changes might include removing irrelevant data, concatenating data, and performing calculations on the data. For example, you might concatenate the separate Last Name and First Name columns in a Customer table to create a single Customer Name column, and add a calculation that multiplies the Unit Price column by the Quantity column for each order to obtain a total revenue value.

Star schemas

In a relational data warehouse built around the dimensional model, there is usually a large table, called a fact table. Fact tables store the important numerical values, such as sales data, costs, and so on. These are the data values, or measures, that organizations can analyze to gain business insights. Typically, business analysts want to examine this data in aggregated form. For example, they might look at total revenue aggregated by date, geographical region, or by product type.

The other tables—often called dimension tables—in the data warehouse, contain the information to be able to do this. For example, there might be a Product table that contains product information, tables that contain the required geographical data, and a table that includes the dates that are relevant to the orders or other data that the data warehouse contains. Conceptually, you can imagine the fact table as a large central table, surrounded by the dimension tables, with foreign key relationships between each dimension table and the fact table. The design of a data warehouse is sometimes called a star schema because it loosely resembles a star.

Online Analytical Processing Databases



A data warehouse often serves as a data source for an online analytical processing (OLAP) system. OLAP systems use pre-aggregated data and specialized storage for fast data analysis with very large data sets.

Multidimensional databases

The most widely used OLAP systems are based on the multidimensional model.

In a multidimensional database, you create a structure called a cube to store measures, which are discrete, aggregated, numerical values—such as sales revenue or manufacturing costs. The cube stores measures in individual cells, and has different dimensions by which you can analyze the data. The dimensions are analogous to the height, width, and depth dimensions of a real, physical cube, except that they represent values such as date, region, or product instead.

Each cell is a location in the cube that represents the intersection of the dimensions of the cube for a given set of dimension values. For example, you might select to analyze the sales data for the product category "Clothing" in the "South" region in the year 2016. The cell in the cube that represents the intersection of the dimensions for the values "Clothing", "South", and "2016" is easily located and it contains the required value, already aggregated.

Data is pre-aggregated in a cube, which means that values do not have to be calculated when a user issues a query against the cube. This makes it much less resource intensive which, in turn, makes it much faster at handling queries.

To query multidimensional cubes, you use the Multidimensional Expressions (MDX) language, though many analysts use tools such as Microsoft Excel® that issues MDX queries on their behalf. For example, when you connect to a multidimensional cube and add data to a pivot table, Excel issues the required MDX queries to the OLAP server in the background.

 **Note:** Cubes in multidimensional databases nearly always have many more dimensions than the three dimensions of a real, physical cube. Therefore, using the concept of a three-dimensional cube to understand multidimensional cubes is not always intuitive.

Multidimensional data storage

There are three principal storage mechanisms for multidimensional databases:

- **Multidimensional OLAP (MOLAP)**. In MOLAP databases, the cube is created, and then populated with the required measures and aggregations through processing. Processing a cube can be resource-intensive and take a long time, depending on the volume of data involved. After processing, the MOLAP database operates independently of the underlying data sources until you have to populate the MOLAP database with the latest data again. A MOLAP cube contains all of the data and aggregate data needed to service queries, such as the query for the dimension values "Clothing", "South", and "2016", described earlier—so there is no need to calculate these values at query time. Consequently, MOLAP storage provides the best performance for multidimensional databases. On the other hand, MOLAP databases are not always up to date, because changes occur to data in the source systems between processing operations. The only way to bring the database up to date is to process it again.
- **Relational OLAP (ROLAP)**. ROLAP storage uses the underlying relational database (often a data warehouse) to store the measures and aggregate values instead of processing them and storing them in the cube. Users still query the cube to access the data by using the MDX language, but the OLAP server must retrieve the necessary data from the relational database, which is less efficient and takes longer than querying the equivalent MOLAP database would. ROLAP has two key benefits:
    - It potentially provides users with access to data that is more up to date. This assumes that the ROLAP database is more up to date than an equivalent MOLAP database would be, which might not always be the case. If the ROLAP database is a data warehouse, the state of the data will depend on the frequency of data loads into the data warehouse.
    - The required storage is less than for the equivalent MOLAP database because the data is stored in only one location (the source relational database) rather than two (the source relational database and MOLAP storage).
- **Hybrid OLAP (HOLAP)**. As the name suggests, HOLAP uses a combination of ROLAP and MOLAP storage to achieve a trade-off between the two storage methods. With HOLAP, you can store some of the data in MOLAP storage and some in ROLAP. A common approach is to store aggregate data, which is usually the most commonly accessed data, in MOLAP storage, and detail-level data in ROLAP. This arrangement gives the performance advantage of MOLAP for queries for aggregate data, and the storage advantages of ROLAP for non-aggregate data.

Tabular Data Models

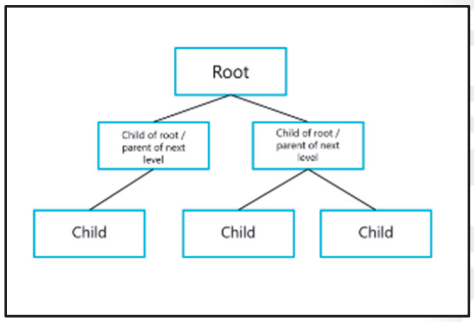
Microsoft SQL Server Analysis Services includes Tabular Data Models. The tabular data model is an alternative approach to creating an analytical solution, in which the data is presented in a relational format. This is much easier to understand and work with than a multidimensional data model, but it does not include all of the features that a multidimensional model offers.

For more information about the differences between multidimensional data models and tabular data models in SQL Server, see Microsoft Docs:

Comparing Tabular and Multidimensional Solutions (SSAS)

http://aka.ms/Jkvyd7

Hierarchical Databases



Although relational databases are the most common type of database, there are several other types that you might encounter, including hierarchical databases and NoSQL databases.

Structure of hierarchical databases

Hierarchical databases organize data in a different way to relational databases. Instead of arranging data as a set of related, non-hierarchical entities, a hierarchical database arranges data in a hierarchy. A hierarchy is a multilevel structure, in which each higher level is the parent of the level below—which is referred to as the child.

You can visualize the hierarchy as an inverted tree structure, with the root at the top, and the child levels branching out below. In a hierarchical database, each child can have only one parent, but each parent can have many children. In other words, there is a one-to-many relationship between the parent and child levels. When retrieving data from a hierarchical database, you start at the root, and follow the parent/child "path" until you locate the required data. The levels in the hierarchy are linked by pointers, which indicate the node to go to next.

Some data is naturally hierarchical and therefore suitable for storage in a hierarchical system. For example, the arrangement of folders, subfolders, and files in a file directory is a natural hierarchy. Each file (child) can only exist in a single folder (parent), but folders can contain many files. Each folder might itself be the child of another folder, and so on. The Windows file directory is an example of this kind of hierarchical database.

Limitations of hierarchical databases

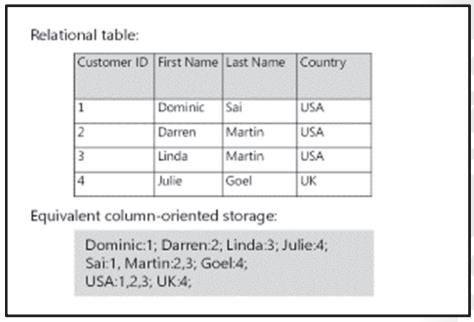
Hierarchical databases handle one-to-many relationships well, but they are not particularly suited to storing data that include many-to-many relationships. An example of a many-to-many relationship is the relationship between products and orders in a relational database. Each product can appear in many different orders, and any order can include many different products. Because this is not a simple one-to-many relationship, it is difficult to implement it in a hierarchical database. By contrast, it is much more straightforward to implement many-to-many relationships in relational databases.

Note: The hierarchical data model dates back to the 1960s, when it was developed by IBM. It predates the relational data model that you learnt about previously in this module. One of the key drivers for the development of the relational data model was to overcome the limitations imposed by the hierarchical model.

Some vendors include features with which you can store and manipulate hierarchical data in a relational database. For example, SQL Server includes the "hierarchyid" data type for this purpose.

Column-Oriented Databases

The term column-oriented refers to a method of storing data that can optimize query response times for some types of workloads. Instead of storing data in rows, as a standard relational database does, a column-oriented DBMS rearranges the data so that all of the data in a column is stored as a single record. For example, the following illustration shows a relational table that contains customer data:



Customer ID	First Name	Last Name	Country
1	Dominic	Sai	USA
2	Darren	Martin	USA
3	Linda	Martin	USA
4	Julie	Goel	UK

In column-oriented storage, this data might be represented as follows:

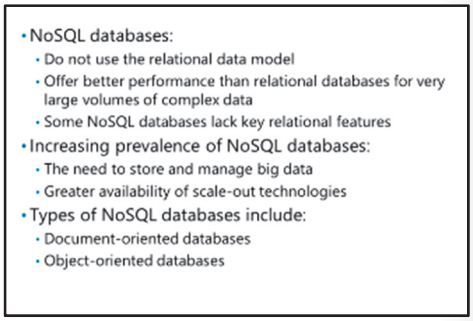
```
Dominic:1; Darren:2; Linda:3; Julie:4;
Sai:1, Martin:2,3; Goel:4;
USA:1,2,3;UK:4;
```

As you can see, each First Name value in the top row is stored along with the corresponding Customer ID value. In the second row, the Last Name value "Martin" appears with two Customer ID values because it pertains to both of these IDs. Similarly, in the third row, the Country value "USA" is associated with three values in the Customer ID column.

This way of arranging data makes it faster and more efficient for some types of queries because it can reduce the number of times the DBMS needs to access the storage disks to obtain the data. Disk access, which is also called disk input and output (IO), is one of the major bottlenecks in a database system, so reducing it can improve performance markedly. However, the query workloads for which column-oriented storage results in better performance are quite limited and specific. For example, queries that aggregate data from a relatively small subset of the total number of columns in a large relational table might perform better if column-oriented storage is used. However, there is no guarantee of this, and it requires careful planning and design.

SQL Server, although a relational DBMS, provides support for column-oriented storage through the Columnstore index feature, and in-memory technology that also uses compression to optimize performance for large database tables.

NoSQL Databases



NoSQL is used to refer to a set of different types of storage solutions that have one thing in common: they are not based on the relational data model. NoSQL databases have become increasingly prevalent in recent years in response to a number of factors, including:

- The rise of big data. Big data is a catch-all term that is used to describe a wide range of data, such as document files, video files, image files, audio files, and emails. These types of data are not structured, or at least not as structured as the data in a relational database. Structured data is organized in a way that preserves the semantic meaning of the data. For example, a column called Last Name in a relational database table called Employees provides the context that gives meaning to the data that it contains. Relational databases are said to be strongly typed, which means that when you create a table, you must define the type of data that each column is permitted to contain. You use data types, such as integer, datetime, and char to do this. By contrast, big data is either semi-structured, such as XML documents, or unstructured, such as the content inside word processing documents. This type of data either lacks the structure that would give context and meaning to the data, or provides only limited structure. NoSQL databases can usually handle this type of data better than relational databases.
- More powerful commodity hardware, virtualization technologies, and cloud services, which organizations can pay for as and when they use the resources, have made it easier and less expensive to create "scale-out" solutions. A scale-out solution involves distributing workloads across multiple nodes (for example, virtual or physical servers) to add computing power. This contrasts with the "scale-up" approach, which involves increasing computing power by adding components such as memory and processors to an individual server. Because of their almost limitless potential for adding processing nodes as required, scale-out solutions can deliver computing power on a massive scale.

In response to these factors, several technologies have emerged that can utilize the power of scale-out systems to store and manipulate semi-structured and unstructured data. Because these technologies do not use relational storage, they are often collectively referred to as NoSQL databases. Not all NoSQL concepts and technologies are new, however—some have existed since before the term NoSQL was coined.

Limitations of NoSQL databases

Although NoSQL databases have many advantages, particularly in terms of performance for larger data sets, they generally do not offer some of the important functionality that is standard in relational databases. For example, relational databases ensure that transactions occur according to a set of criteria called ACID properties. The acronym ACID stands for atomicity, consistency, isolation, and durability.

Atomicity means that each transaction can proceed on an all-or-nothing basis; either it completes in its entirety or not at all. Many transactions include multiple steps that should all complete as a unit.

Consistency means that a transaction will not leave the database in an inconsistent state after completion—for example, by the addition of invalid data.

Isolation means that transactions occur without interference from other transactions, which ensures that the data that users see is accurate and meaningful.

Durability means that, after completion, the changes that a transaction makes to a database are permanent.

Not all NoSQL systems support ACID properties to the same degree as relation databases. Furthermore, relational databases use SQL, a fully featured query language, to provide data access and manipulation. Although some NoSQL databases do include a query language that offers SQL-like functionality, these languages are not as sophisticated as SQL.

Note: Because some NoSQL databases include support for query languages similar to SQL, they are sometimes called "Not only SQL" databases as an alternative to the term NoSQL database.

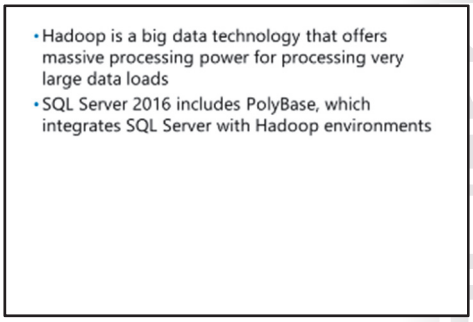
Types of NoSQL databases

There is no single type of NoSQL database. The following are examples of commonly used types of NoSQL databases:

- **Document-oriented databases**. As the name suggests, document-oriented databases store documents that are semi-structured. Examples include XML (extensible markup language) documents and JSON (JavaScript Object Notation) documents. These types of documents are semi-structured because they contain metadata (such as the tags in an XML document) that provides a semantic description of the data, giving it context and meaning. However, each document might contain completely different metadata, in contrast to a relational database table, in which each row must have the same structure. Documents in a document-oriented database are each paired with a key value that uniquely identifies them.
- **Object-oriented databases**. You create object-oriented databases by using an object-oriented programming language such as C♯, Perl, Java, or Visual Basic. This type of database can be useful when the data is complex, as might be the case with, for example, computer aided design (CAD) data. In an object-oriented database, you store data as programming objects, with data values stored as attributes of the object. Objects also include methods, which define the actions that you can perform on the object. For example, a database might store a customer as an object, with attributes including name, address, phone, and so on; methods might include update and delete. The structure of a customer object would be defined as a class, which is analogous to a blueprint. Whenever you create an object, its attributes and methods are determined by the object class.

Other types of NoSQL databases include key-value stores, which have document-oriented databases, and graph databases.

Hadoop



Like the NoSQL databases described in the previous topic, Hadoop is a scale-out platform for storing and working with semi-structured and unstructured data. Hadoop is actually a group of related technologies, including:

- The Hadoop Distributed File System (HDFS). Hadoop stores data in clusters of computers or nodes, and distributes large files across the nodes in a cluster by using HDFS.
- Hadoop MapReduce is a program you can use to coordinate the processing of data in parallel across multiple nodes. Parallel processing delivers much faster performance.

Although Hadoop and NoSQL databases both handle big data, they work best with different types of workloads. Hadoop is designed for massive-scale processing, and performs especially well with extremely large, read-only workloads. By contrast, NoSQL databases generally perform best with read/write workloads, such as those generated by web applications.

Hadoop and SQL Server

SQL Server includes a technology called PolyBase, which allows queries to be run on data stored in Hadoop. For example, with PolyBase you can:

- Access data stored in Hadoop and Azure Blob Storage by using Transact-SQL. You can create queries that return only data from Hadoop, or queries that return results which combine data from Hadoop and relational data stored in SQL Server.
- Export data from SQL Server to Hadoop and Azure Blob Storage.
- Use the suite of Microsoft business intelligence (BI) tools to analyze and report on Hadoop data.

For more information, see the PolyBase Guide in Microsoft Docs:

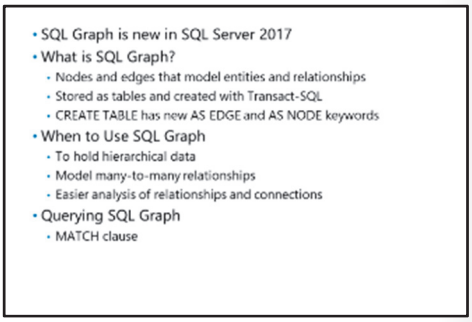
PolyBase Guide

http://aka.ms/Aqamxl

SQL Graph

SQL Graph is new in SQL Server 2017. A graph database allows complex relationships to be modelled, and simplifies the analysis of many-to-many relationships.

SQL Graph enables graph nodes and edges to be stored and queried in SQL Server. A SQL graph database doesn't provide any additional functionality compared to a relational structure, but it can simplify queries and make interconnected relationships easier to model.



What is SQL Graph?

A graph database is made up of a number of nodes and edges. Nodes and edges represent entities and relationships and are stored as tables. SQL Server 2017 includes the AS NODE and AS EDGE keywords to enable SQL graph tables to be created.

When to Use SQL Graph

Graph databases are well suited to modelling the following situations:

- You hold hierarchical data, and entities have one or more parents.
- Your data has complex many-to-many relationships.
- You want to analyze and query how your data is connected.

Querying Graph Data

The new MATCH clause allows matching and multi-hop navigation through the graph data. Use the MATCH clause in the WHERE clause of a SELECT statement.

Categorize Activity

Match the types of database to the correct description. Indicate your answer by writing the category number to the right of each item.

Items	
1	Highly structured databases that store data in related tables.
2	A broad category of different types of databases that are often used to store semi-structured and unstructured data, or highly complex data.
3	Analytical databases that contain aggregated data stored in cubes.

Category 1		Category 2		Category 3
Relational Databases		NoSQL Databases		OLAP Databases

Lesson 3
Data Analysis

The data that a database stores can have many uses, but without context, the data is meaningless. This lesson explains the difference between data and information, and how organizations can use data analysis to assist with their decision-making processes.

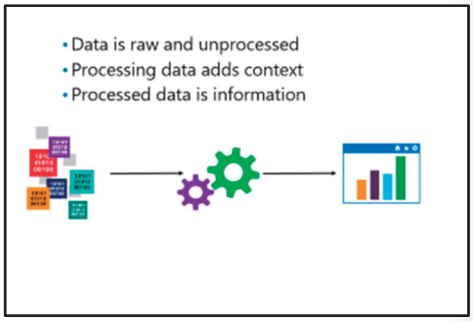
Lesson Objectives

After completing this lesson, you will be able to:

- Explain the difference between data and information.
- Explain the use of data analysis to obtain intelligence.
- Outline some key steps for preparing data for analysis.

Data and Information

To individuals who are not database professionals, the words "data" and "information" may mean approximately the same thing. However, to a data professional, these words are not synonymous and there is a clear difference between the two.



In a computing context, data is raw and unprocessed, and lacks any context to give it meaning. For example, a collection of numerical values could represent a list of sales receipts, a list of height measurements, or a list of distances. Without context, it is impossible to know exactly what the data is. Information is data that has been processed in some way that makes it meaningful and useful.

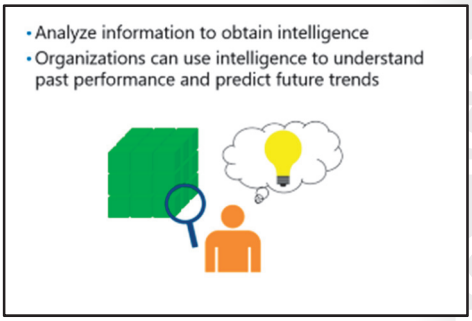
Processing data

In the previous lesson, you learned that databases can store data in many different ways. Each type of database stores data along with metadata, which is additional data that provides context for the data. For example, in a relational database, you take raw data and process it by storing it in tables and columns. The values in a column are the data and the column name is metadata. Processing data in this way gives meaning to the data. Other types of metadata can provide additional context.

Producing useful information

Metadata provides some meaning for data, but to transform data into truly meaningful and useful information often requires additional processing. For example, a list of values in a column called Sales Receipts is not particularly useful to a person who wants to know how much revenue the company generated last month. To obtain this information, you have to process the data, for example by aggregating the sales receipts values and filtering them by month. In some databases, such as data warehouses, you might store these aggregations in the database; in other types of database, such as an OLTP database, you might not store them, but instead calculate them as required when you query the database.

Information and Intelligence



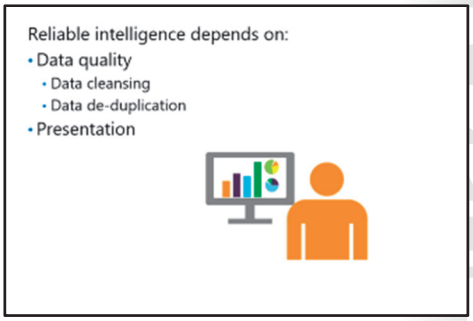
You can use data further by analyzing the information that it contains to reveal intelligence. This intelligence can help business decision makers to identify and understand past trends with a view to predicting possible future trends. For example, having processed raw sales data to generate information, you might notice that there was a significant increase in sales over a three-month period. A data analyst might investigate further to try to find the cause of this increase. The kinds of questions that they might ask include:

- Does the increase in sales correlate with any action that the company took; for example, what promotions, marketing campaigns, or advertising campaigns were going on at the time?
- Were any new products launched around that time?
- Were there any well-publicized events related to the company's area of business going on?
- Were there any other significant circumstances, such as unusual weather conditions?
- Had a competitor recently ceased trading?
- Are any of the above factors correlated?

Data analysts can also explore data by using sophisticated algorithms and statistical analyses to reveal patterns and trends that might not be immediately obvious, to reveal new lines of enquiry. In addition to finding answers to questions, analysts can use the information they have to predict future trends, and plan, for example, the best time to launch a promotion.

SQL Server Analysis Services enables you to create cubes and tabular data models that can be used with business intelligence tools, including Excel and PowerPivot, to perform data analysis. Analysis Services also includes data mining functionality.

Preparing Data for Analysis



To ensure data analysis yields reliable, practical, usable results, it is important that the data you use to perform the analysis is valid. You must also have ways of representing the findings that are clear and easily understood by decision makers.

Ensuring data quality

To be valid, data must be accurate so that you are confident that the data has the required meaning. Ensuring the quality of data includes:

- **Data cleansing**—the process of correcting or removing incorrect and incomplete data.
- **Data de-duplication**—the process of correcting data that is represented in different ways. For example, a database might contain the values "USA" and "United States", both of which refer to the same entity.

Many RDBMS systems include a tool for ensuring data quality, and there are also third-party tools. With SQL Server you can use Data Quality Services to profile data and correct data inconsistencies so that the data analyses you undertake are as accurate as possible. You would typically use Data Quality Services to ensure that the data you load into your data warehouse is valid.

Presenting data

It is important to communicate the insights that data analysis provide, and in a format that can be easily understood. Poorly presented intelligence is frustrating for those who have to use it, and can lead to misunderstandings about the findings. This might result in decision makers using incorrect interpretations to guide the decision-making processes. For the same reason, it is also important to ensure that analysis reports are up-to-date.

Reporting Services in SQL Server is a comprehensive tool for creating and managing reports—both reporting specialists and non-specialists can use it to generate reports quickly and easily. Tools such as Excel, PowerPivot, and Power View are also very popular ways of representing and communicating intelligence.

Verify the correctness of the statement by placing a mark in the column to the right.

Statement	Answer
Data Quality Services is a tool for analyzing data to gain insights.	

Lesson 4
Database Languages in SQL Server

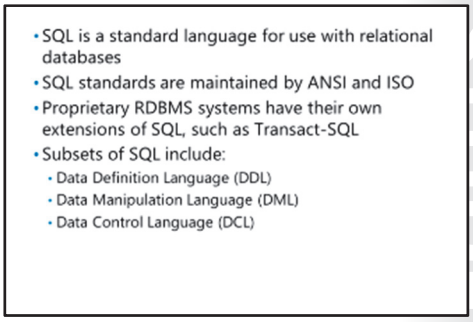
SQL Server includes support for several languages, so you can work with data in a wide variety of contexts to achieve a range of different aims. SQL is the principal language that most SQL Server database professionals use, but other languages exist that support other key functions, including data analysis and data mining.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the components of SQL.
- Write a SELECT query with a WHERE clause.
- Explain the purpose of the other languages that SQL Server supports.

Structured Query Language



Structured Query Language (SQL) is a standard language that is used to create, manage, and access relational data in an RDBMS. RDBMS systems that support SQL include Microsoft SQL Server, Oracle, IBM DB2, and Sybase.

SQL is governed by the standards set out by the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO)—the most recent set of standards for the language is SQL:2011. Proprietorial RDBMS systems generally adhere to these standards, but each RDBMS differs from the standards to some degree. The version of SQL that SQL Server uses is called Transact-SQL.

SQL includes three subsets, called Data Definition Language (DDL), Data Control Language (DCL), and Data Manipulation Language (DML).

DDL

DDL includes components with which you can define database objects, such as tables and indexes. An index is a data structure that improves performance by making it faster to find data, in a similar way to an index in a book. DDL uses CREATE, ALTER, and DROP statements for creating, modifying, and deleting objects.

The following code example is a Transact-SQL statement that creates a table called Sales.Customer, with the columns CustomerID, StoreID, TerritoryID, AccountNumber, and ModifiedDate:

DDL Example

```
CREATE TABLE [Sales.Customer](
             [CustomerID] [int] NOT NULL,
             [StoreID] [int] NULL,
             [TerritoryID] [int] NULL,
             [AccountNumber] [int] NOT NULL,
             [ModifiedDate] [datetime] NOT NULL);
GO
```

Each of the columns in the code example has a data type. The int data type defines that a column's contents must be integer values, and the datetime data type defines that a column must contain date and time data.

The definition for each column also states either NULL or NOT NULL. NULL defines whether the column is permitted to contain missing values. NULL means it can contain missing values, NOT NULL means it cannot.

 **Note:** You should use two-part names to refer to tables in SQL Server databases, such as Sales.Customer. The first part of the name refers to the schema, which you can think of as a namespace or logical container for the table.
You typically group tables into a schema based on a logical relationship between the tables, such as the fact that a group of tables all store data that is relevant to the sales process. The second part of the name refers to the table within the schema.

DML

DML includes components with which you can view and manipulate data in a database. You can write DML statements by using the SELECT, INSERT, UPDATE, and DELETE keywords.

You use SELECT statements to retrieve data from a database—for example, to view it in an application. With INSERT, UPDATE, and DELETE, you can add rows to tables, change values in rows in tables, and delete rows from tables. For example, when a customer signs up to a shopping website for the first time, a new row representing that customer will be added to the database.

When that customer changes their phone number, the same row is updated. The following code example is a Transact-SQL statement that retrieves the CustomerID and AccountNumber values for all rows in the Sales.Customer table:

DML Example

```
SELECT CustomerID, AccountNumber
FROM Sales.Customer;
GO
```

You will learn more about SELECT statements in the next topic.

DCL

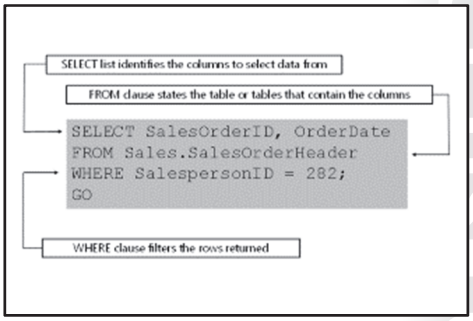
DCL uses GRANT, DENY, and REVOKE statements to control access to data by defining permissions. You can define permissions to individual users and to groups of users. GRANT statements permit users to perform specific actions, such as read data or update data. If you do not grant permissions to users, by default they will not be given any kind of access to the data. This is called implicitly denying permission. DENY statements over-ride GRANT statements and explicitly prevent users from performing specific actions. For example, a user might have access to data because permission has been granted to a group that they are a member of. You can deny this user access without denying the rest of the group access by using a DENY statement to override the GRANT permission for that user. REVOKE statements revoke the permissions defined by both GRANT and DENY statements.

The following Transact-SQL code example grants SELECT permission to the **Sales.Customer** table to a user called Sarah:

DCL Example

```
GRANT SELECT ON OBJECT::Sales.Customer
TO Lin;
GO
```

Transact-SQL Queries



The SELECT statement

You can query SQL databases by using SELECT statements, as you saw in the previous topic. A SELECT statement begins with the SELECT keyword, followed by a list of the columns, separated by commas, which contain the data that you want to view. You use the FROM keyword to specify the name of the table that contains the data.

Instead of listing specific columns to return, you can use the * symbol to indicate that the SELECT statement should return all of the columns. The end of a statement is marked by the semicolon symbol, although for many implementations of SQL—including Transact-SQL—this is not a requirement. It is however, recommended to finish with a semicolon.

Filtering SELECT statements by using WHERE clauses

The SELECT statement example in the previous topic returned all of the data in two columns for every row in the Sales.Customer table. However, you do not always want to return data for every row in a table. For example, if the reason for the query is to find the customers who placed orders in a particular month, or who live in a particular location, you would not be interested in any rows that contained customers that did not meet these criteria; in fact, having these irrelevant rows in the query result set would be an impediment to you finding an answer to your question.

An important principle when writing queries is to only return the data that you need. In addition to making results easier to understand and ensuring that they are accurate, limiting the rows that you return also results in faster query response times because the database engine has less data to locate and return.

To filter a query and limit the rows that a query returns, you can use a WHERE clause in a SELECT statement. A WHERE clause contains a predicate, which evaluates to true, false, or unknown.

In this example, the WHERE clause contains a predicate that evaluates whether orders were taken by the sales person with the SalespersonID 282. The SELECT statement returns only the rows where this evaluates to true (in other words, rows for which the SalespersonID value is 282).

SELECT … WHERE Example

```
SELECT SalesOrderID, OrderDate
FROM Sales.SalesOrderHeader
WHERE SalespersonID = 282;
GO
```

Operators

In the SELECT statement above, "SalespersonID = 282" is a predicate. The **=** (equals) symbol is an example of an operator. Operators define how predicates are evaluated. Other operators include:

- **>** (greater than)
- **<** (less than)
- **>=** (greater than or equal to)
- **<=** (less than or equal to)
- **<>** (not equal to)
- **!=** (not equal to)

You can also use operators to evaluate multiple conditions. For example the AND operator requires multiple conditions in the WHERE clause to be true, and the BETWEEN operator requires only one or the other of two conditions to be true. Other operators include LIKE, with which you can compare values by using wildcard characters, and IS NULL, which you can use to identify NULL values.

Demonstration: Querying a SQL Server Database

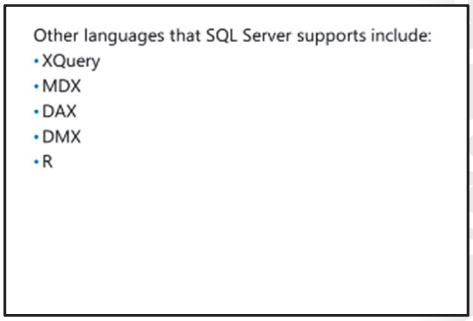
In this demonstration, you will see how to:

- Use a SELECT statement to return all rows and columns from a table.
- Use a SELECT statement to return all rows and specific columns from a table.
- Use a WHERE clause to filter the rows that a SELECT statement returns.
- Use different operators in a WHERE clause.

Demonstration Steps

1. Open Microsoft SQL Server Management Studio, and then connect to the **MIA-SQL** instance of the database engine by using Windows Authentication.
2. On the **File** menu, point to **Open**, click **File**, browse to **D:\Demofiles\Mod01**, click **TransactSQLQueries.sql**, and then click **Open**.
3. Select the code under the comment **Return all rows from all columns in the Sales.SalesOrderHeader table**, and then click **Execute**.
4. Review the result set, noting the columns that the query returns and, in the bottom right-hand corner of the results set, the number of rows the query returned.
5. Select the code under the comment **Return all rows from the SalesOrderID and OrderDate columns from the Sales.SalesOrderHeader table**, and then click **Execute**.
6. Review the result set, noting the columns and the number of rows the query returned.
7. Select the code under the comment **Return only rows from the SalesOrderID, OrderDate, and SalesPersonID columns for which the SalespersonID = 282**, and then click **Execute**.
8. Review the result set, noting the columns and the number of rows the query returned. Note that the number of rows returned is much lower than for the previous two queries.
9. Select the code under the comment **Return only rows from the SalesOrderID, OrderDate, and SalesPersonID columns for which the SalespersonID > 282**, and then click **Execute**.
10. Review the result set, noting the columns and the number of rows the query returned. Note that the number of rows returned has increased again.
11. Select the code under the comment **Return only rows from the SalesOrderID and OrderDate columns for which the SalespersonID = 282 and the orderdate is from the year 2013**, and then click **Execute**.
12. Review the result set, noting the columns and the number of rows the query returned. This query returned the lowest number of rows because it contained the most restrictive filters.
13. Close the TransactSQLQueries.sql query window and do not save any changes. Close SQL Server Management Studio.

Other Query Languages in SQL Server



Although Transact-SQL is the primary language that you use, SQL Server also supports other languages so that you can work with data in different formats. These languages include XQuery, MDX, Data Analysis Expressions (DAX), Data Mining Expressions (DMX), and R.

XQuery

XML is a markup language that looks very similar to Hyper Text Markup Language (HTML) because, like HTML, XML uses tags. In XML, these tags are called elements. Elements in a block of XML have angled bracket symbols (< and >) surrounding them. In XML, the tags surround items of data, giving it semantic meaning.

In this example, the XML includes data about a person's appearance, and the elements tell you what the data means:

In this example, the XML includes data about a person's appearance, and the elements tell you what the data means:

XML Example

```
<Appearance>
      <EyeColor>Green</EyeColor>
      <HairColor>Brown</HairColor>
      <HeightCentimeters>175</HeightCentimeters>
</Appearance>
```

XML documents usually include an XML schema, which defines the elements that the document can contain, the data type of those elements, and other things such as any child elements that are permitted or any default values that apply. With SQL Server, you can store XML data in columns in tables by using the xml data type, and you can query this data by using XQuery. For example, using the simple XML example above, you could use XQuery to return all people with green eyes, or blond hair.

MDX

MDX is a query language that is similar in some ways to SQL. Like SQL, MDX uses SELECT statements with FROM and WHERE clauses, but the syntax within these clauses is different to the syntax in the equivalent SQL clauses.

MDX is designed for querying multidimensional cubes, and the result set of an MDX query is also multidimensional; it is also a cube. In an MDX SELECT statement, you have to specify the values that you want to place on the axes in the result set. For example, you might choose to add measure values such as Sales Revenue to the row axis, and dimension values such as the years 2014 and 2015, from the Date dimension to the column axis. Because the cube is a multidimensional structure, you can add more than two axes if required. You specify the name of the cube that contains the data in the FROM clause, and in the WHERE clause, you specify the dimension that you want to use to filter the data. For example, you might specify the Product dimension to limit the query results to a specific product.

Many of the client applications that data professionals use, including Microsoft Excel, issue MDX queries to retrieve data in response to user actions. For example, a user might use Excel to connect to a cube and then add measures and dimensions from that cube to a pivot table or chart in a worksheet to create a visual representation of the data. The person using Excel does not have to write the MDX queries to retrieve the data themselves; Excel does this automatically in the background.

DAX

DAX is a formula-based language that you can use to implement business logic when working with tabular data models. DAX will be familiar to anyone who has worked with Excel formulas because it uses similar functions and syntax. You can use DAX functions to create formulas—for example to add measures to tabular data models—or to define data relationships, such as many-to-many relationships. You can also use DAX to write queries to return data.

Data Mining Extensions (DMX)

Data mining functionality is available as part of SQL Server Analysis Services. Data mining is the process of exploring large volumes of data to identify correlations and trends that might not be immediately obvious. You can use the DMX as a language to create, train, browse, and manage data mining models, and to generate predictions based on them.

R

R is a statistical programming language for performing advanced statistical analyses on complex data sets.

The statistical analysis functionality that R offers goes beyond the analytical capabilities of the other languages that SQL Server supports. With R, you can explore many different kinds of complex data, and use it to create powerful predictive data models.

R also includes graphics libraries that can present visualizations of data to users, making them easier to understand. Because R is built in to SQL Server, you can perform statistical analysis directly, without having to extract the data before analyzing it.

Check Your Knowledge

Question	
Which one of the following correctly represents the structure of a SQL Server Transact-SQL SELECT statement?	
Select the correct answer.	
	SELECT <table> FROM <column list> WHERE <filter that limits the rows returned>
	SELECT <column list> FROM <table> WHERE <filter that limits the rows returned>
	SELECT <rows> FROM <table> WHERE <filter that limits the rows returned>

Lab: Exploring and Querying SQL Server Databases

Scenario

You have recently started working as a database developer at Adventure Works, manufacturers and retailers of bicycles and associates products. Your line manager has asked you to familiarize yourself with the databases that you will be working with, and has given your user account the required permissions to do so. You intend to create database diagrams of parts of the OLTP database and data warehouse database, and examine the tables they contain. You will then write some simple SELECT statements and review the data.

Objectives

After completing this lab, you will be able to:

- Create database diagrams to explore databases.
- Write Transact-SQL SELECT statements that include a WHERE clause.

Estimated Time: 45 minutes

Ensure that the 10985C-MIA-DC and 10985C-MIA-SQL virtual machines are both running, and then log on to 10985C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.

Exercise 1: Exploring an OLTP Schema and a Data Warehouse Schema

Scenario

You will begin your exploration of the databases by creating two database diagrams, one for the Adventure Works OLTP database, and one for the Adventure Works data warehouse.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Explore an OLTP Database Schema
3. Explore a Data Warehouse Schema

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 10985C-MIA-DC and 10985C-MIA-SQL virtual machines are both running, and then log on to 10985C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. Browse to the **D:\Labfiles\Lab01\Starter** folder, and then run **Setup.cmd** as Administrator.

▶ Task 2: Explore an OLTP Database Schema

1. Open SQL Server Management Studio and connect to the MIA-SQL Database Engine instance by using Windows Authentication.
2. In the **AdventureWorks2016** database, create a database diagram and then add the following tables to it:
    - **Address (Person)**
    - **Customer (Sales)**
    - **SalesOrderDetail (Sales)**
    - **SalesOrderHeader (Sales)**
    - **SalesPerson (Sales)**
    - **SalesTerritory (Sales)**
    - **ShipMethod (Purchasing)**

3. Examine the primary key columns and the foreign key relationships between the tables in the diagram, and note which columns are involved in the relationships.
4. Save the database diagram as **Adventure Works Diagram**, and then close the diagram window.

▶ Task 3: Explore a Data Warehouse Schema

1. In the **AdventureWorksDW2016** database, create a database diagram and add then the following tables to it:
   - **DimCustomer**
   - **DimDate**
   - **DimProduct**
   - **DimProductCategory**
   - **DimProductSubcategory**
   - **FactInternetSales**
2. Examine the primary key columns and the foreign key relationships between the tables in the diagram, and note which columns are involved in the relationships.
3. Save the database diagram as **Adventure Works Data Warehouse Diagram**, and then close the diagram window.

Results: After completing this exercise, you will have created a database diagram in the AdventureWorks2016 database and a database diagram in the AdventureWorksDW2016 database.

Exercise 2: Querying a Database by Using Transact-SQL

Scenario

You will now continue your exploration of the Adventure Works OLTP database by writing and executing SELECT statements and examining the result sets.

The main tasks for this exercise are as follows:

1. Write Transact-SQL SELECT Statements
2. Write Transact-SQL SELECT Statements with a WHERE Clause

▶ Task 1: Write Transact-SQL SELECT Statements

1. In SQL Server Management Studio, open a new query window connected to the **AdventureWorks2016** database.
2. Write and execute a Transact-SQL query that returns all columns and all rows from the **Sales.SalesOrderHeader** table.
3. In the same query window, under the existing query, write and execute a Transact-SQL statement that returns the **SalesOrderID**, **OrderDate**, **SalesPersonID** columns and all rows from the **Sales.SalesOrderHeader** table.

▶ Task 2: Write Transact-SQL SELECT Statements with a WHERE Clause

1. Under the existing query, write and execute a Transact-SQL statement that returns the **SalesOrderID**, **OrderDate**, **SalesPersonID** columns from the **Sales.SalesOrderHeader** table, and use a WHERE clause to return only the rows with a **SalesPersonID** value of **279**.
2. Review the results and note the number of rows that the query returned.

3. Under the existing query, write and execute a Transact-SQL statement that returns the **SalesOrderID**, **OrderDate**, **SalesPersonID** columns from the **Sales.SalesOrderHeader** table, and use a WHERE clause to return only the rows with a **SalesPersonID** value of **279** or **282**.
4. Review the results and note the number of rows that the query returned.
5. Under the existing query, write and execute a Transact-SQL statement that returns the **SalesOrderID**, **OrderDate** columns from the **Sales.SalesOrderHeader** table, and use a WHERE clause to return only the rows with a **SalesOrderID** value that is between **57000** and **58000**.
6. Review the results and note the number of rows that the query returned.
7. Under the existing query, write and execute a Transact-SQL statement that returns the **SalesOrderID**, **OrderDate** columns from the **Sales.SalesOrderHeader** table, and use a WHERE clause to return only the rows with a **SalesPersonID** value of **279** and an **OrderDate** value that includes the year **2014**.
8. Review the results and note the number of rows that the query returned.
9. Close SQL Server Management Studio without saving changes.

Results:

After completing this exercise, you will have:

Written and executed SELECT statements to retrieve all columns and to retrieve specific columns from a table in the Adventure Works OLTP database.

Written and executed SELECT statements that include a WHERE clause to filter the rows that are returned from a table in the Adventure Works OLTP database.

Question: Why did the number of rows returned by the queries that you wrote in the lab vary?

Module Review and Takeaways

In this module, you have seen how to:

- Describe the characteristics of relational databases.
- Describe several other common types of database and storage methods.
- Describe the process of data analysis.
- Describe the languages that you can use in SQL Server.

Review Question(s)

Question: What types of databases and RDBMSs are in use in your work place? Do you have any experience of working directly with them?

Module 2

Data Modeling

Contents:

Module Overview

This module provides an introduction to data modeling techniques that you can employ when designing databases. The techniques can be applied to any relational database system; however, in this module (where appropriate) there will be specific examples using Microsoft® SQL Server®.

Objectives

After completing this module, you will be able to:

- Describe data modeling techniques.
- Describe the three levels of the ANSI-SPARC Database Model.
- Understand the components of an entity relationship model.

Lesson 1
Data Modeling

This lesson provides an overview of common data modeling techniques, so that you can understand them and how they are used.

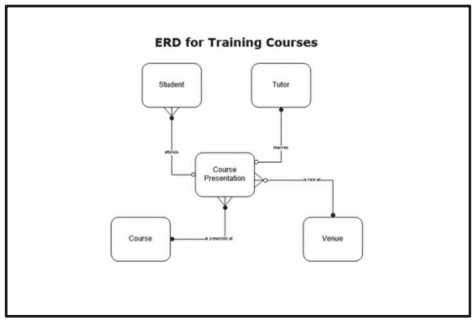
Lesson Objectives

After completing this lesson, you will be able to:

- Describe entity relationship models.
- Describe database normalization.
- Describe transaction path analysis.

Entity Relationship Modeling

Entity relationship modeling is a method for diagrammatically showing the entities required for a particular application and the relationships that exist between them. As an example, consider an application that manages training courses. There is likely to be a need to model the following items:



- Students
- Tutors
- Courses
- Venues
- Course presentations

These items would be modeled as entities, with each entity containing the relevant details, or attributes, for that item.

Additionally, there are relationships that exist between these entities:

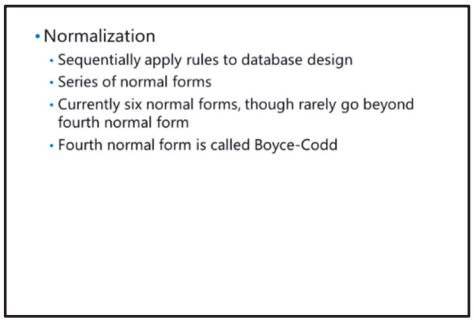
- STUDENTS attend COURSES.
- TUTORS present COURSES.
- COURSES are run at VENUES.
- COURSES are run at different PRESENTATIONS.

Entities and relationships are discussed in detail later in this module, but for now, it is sufficient that you understand the basic purpose of entity/relationship modeling. That is, to show what information is required to be stored and the relationships that need to exist between the information types.

Entities and relationships are usually shown in an entity-relationship diagram (ERD). In this diagram, entities are shown as blocks, with the lines between them representing the relationships. A filled circle joining a relationship to an entity signifies that the entity is required; an open circle signifies that the entity is optional. The crow's foot shows that more than one of that entity can be related to one instance of the related entity.

Normalization

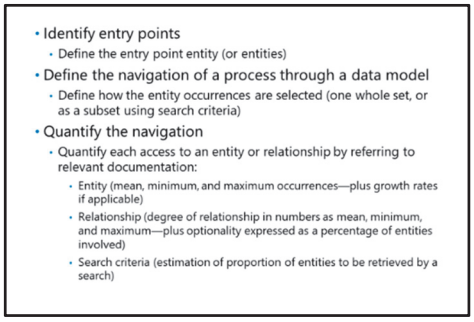
Normalization is a method of sequentially applying rules to a database design to remove data redundancies, improve stability, eliminate update anomalies, and improve data integrity. These rules are divided into a series of normal forms as introduced in Module 1 of this course. The normal forms are applied to a table sequentially; that is, a table must be in first normal form before you can apply second normal form, and so on.



Currently, there are six normal forms, with most systems adhering to the third or fourth normal form. The fourth normal form in the list is called Boyce-Codd normal form (BCNF) after its co-developers; the other normal forms have no individual names.

Transaction Path Analysis

You can use transaction path analysis (TPA) to determine entity/attribute usage and routes through the data model. TPA information is obtained by using techniques that examine the processes that access the data in a database. The objectives of TPA are to:



- Validate the data model. Does it support business processing needs?

  During analysis, the data model and the process model are likely to have been treated separately. It is vital to ensure that the developed data model sufficiently supports the needs of the business processes. Specifically, have all necessary entity types, relationships, and attribute types been identified?

- Validate the process design. Is the process sufficiently understood and mapped correctly?

  It may appear as though the processes have been defined to a sufficient level of detail. However, by considering how the processes need to access the data stored in the database, TPA can help you ensure that simple data access paths exist, and that you fully understand the process logic.

- Form the basis of security/privacy design. Which processes need to do what to which data?

  TPA is not only concerned with entity types and relationships. For each process, you should determine how well the data model supports it. For this, you need to define which attributes should be selected and how they are to be used. This can help you define the basis of the security and privacy framework.

- Form the basis of physical database design. With TPA, you can identify potential navigation problems early, helping you to:
- Identify where keys should be added to entities to avoid having to access them via a relationship from another entity.
- Recognize when derivable data should be held in an entity (rather than derived on the fly).
- Identify additional relationships that may be required.
- Assist with performance assessment.

  TPA identifies the number of logical data accesses required against a data model for each process. This means you can estimate the performance of individual processes and the overall processing requirements of the entire system.

In summary, TPA tells us:

- Whether the data model will support the required business processing.
- The entry points (entities) required.
- Who needs to use which data and how it will be used.

Verify the correctness of the statement by placing a mark in the column to the right.

Statement	Answer
True or false? You can use entity-relationship modeling to show the way the elements in the database are interconnected.	

Lesson 2
ANSI-SPARC Database Model

The ANSI-SPARC database model is an abstract model for a DBMS. It stands for American National Standards Institute, Standards Planning and Requirements Committee—but is not a formal standard. It consists of three levels, each of which is associated with a database schema.

This lesson describes the three levels of the ANSI-SPARC database model and their usage. It also introduces database schemas.

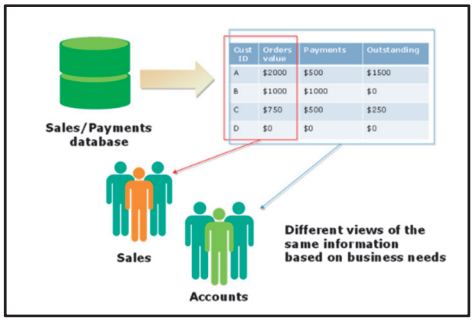
Lesson Objectives

After completing this lesson, you will be able to:

- Describe the external level of the ANSI-SPARC model.
- Describe the conceptual level of the ANSI-SPARC model.
- Describe the internal level of the ANSI-SPARC model.
- Understand and explain database schemas.

External Level

The external level of the ANSI-SPARC model defines how the data in a database will be viewed by different users. Each user might need to see different parts of the data, or to see the data arranged in different ways. These requirements are handled by the use of views, each extracting and organizing the information in the required ways.



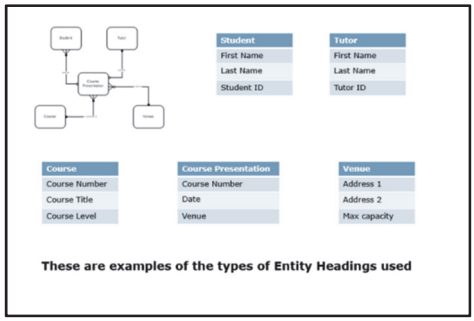
For example, an employee in the Sales department may want to see information about customers and their orders, whilst someone in the Accounts department may need information about those customers and their payment history. All this information is likely to be contained in a database holding the sales and payments information, but separate views will extract the necessary information to satisfy the two groups of employees.

You can also use the external view to restrict access to data to authorized users.

Conceptual Level

The conceptual level of an ANSI-SPARC database model represents the data requirements for an organization. It is important to recognize that:

- The Database Administrator works at and defines this level.
- It describes the user structure and gives a global view of the database.
- It is independent of hardware and software.



The conceptual level shows the three fundamental building blocks of:

- Entities
- Attributes
- Relationships

It usually comprises an ERD, entity headings, and any constraints and/or assumptions.

ERD and Entity Headings

The slide shows the ERD and tables with typical entity headings. These are just sample headings; many more can be added according to the business requirements.

Constraints

The following constraints exist:

- Each COURSE PRESENTATION *must* have at least one STUDENT attending.
- Each COURSE PRESENTATION *must* be taught by one TUTOR.
- Each COURSE PRESENTATION *must* be held at one VENUE.
- Each COURSE PRESENTATION *must* be for one COURSE.
- Each COURSE *must* have at least one COURSE PRESENTATION.

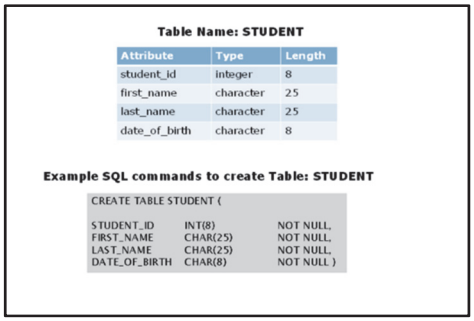
Assumptions

There are no assumptions made in this example.

Internal Level

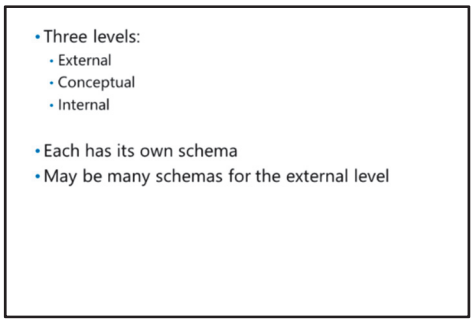
The internal level of an ANSI-SPARC database model is used to show how the database is physically represented in the computer system. It describes how the data is actually stored in the database and on the computer hardware.

The internal view defines the actual record layouts of each table in a database. Information from the conceptual view is used to create the physical files, using SQL commands. These physical files will store the data described by the conceptual views.



Database Schemas

A database schema can graphically show the way a particular database is organized. There are three different types of schema corresponding to the three levels in the ANSI-SPARC architecture previously described:



- The external schema describes the different external views of the data; there may be many external schemas for a given database.
- The conceptual schema describes all the data items and relationships between them, together with integrity constraints. There is only one conceptual schema per database. The components in the conceptual schema are described in the next lesson.
- The internal schema, the lowest level, contains definitions of the stored records, the methods of representation, the data fields, and indexes. There is only one internal schema per database.

The overall description of a database is called the database schema.

Check Your Knowledge

Question	
Where is the physical representation of the data modeled?	
Select the correct answer.	
	External level
	Conceptual level
	Internal level

Lesson 3
Entity Relationship Modeling

This lesson expands on the technique of Entity Relationship Modeling introduced in a previous lesson.

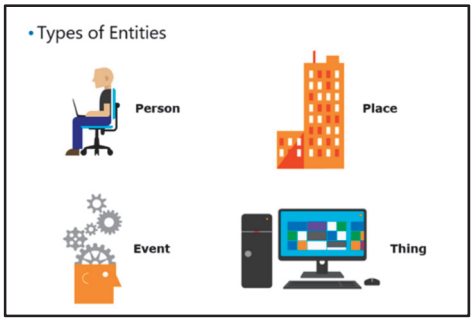
Lesson Objectives

After completing this lesson, you will be able to:

- Explain what is meant by an entity.
- Describe the types of attribute that an entity can have and their use as keys.
- Explain what is meant by a relationship.
- Identify the different types of relationships and how they are implemented.
- Read and understand an ERD and appreciate the different notations used.

Entities

An entity is a data modeling concept that represents a real-world object about which information is required to be stored. An entity can represent a physical object, such as a person, place, or a thing; or it could represent a virtual object, such as an event.



Entities can have the following associated with them:

- **Name**: the name of the entity, used to uniquely identify the entity.
- **Description**: more detailed information about the entity.
- **Range of values**: the valid range of values that an entity can hold.
- **Quantity**: the number of allowable instances of a particular entity (optional).
- **Growth**: whether the number of instances of a particular entity is fixed, or can increase; also specifies the rate at which an entity can grow.

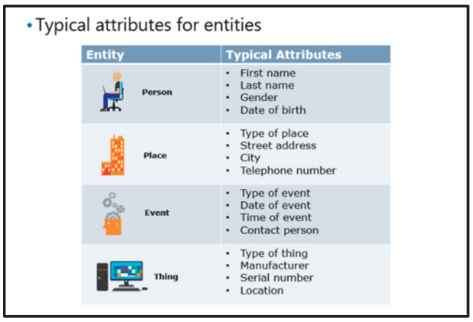
In a database, an entity is represented by a table. Within each table, the individual rows represent one instance of that entity. These instances are also referred to as "tuples". Each tuple is, by definition, distinct from all other tuples within the same table. This is because a primary key always exists.

An informal definition of a primary key is an attribute, or combination of attributes, where no two rows (tuples) in a table (entity) can have the same value(s) for the attribute(s) comprising the primary key.

Attributes and Keys

Attributes are pieces of data that we wish to record for an entity. For example, for an entity representing a Person, we may wish to record:

- First name
- Last name
- Date of birth
- Gender
- … and so on …



An attribute is a noun or noun phrase, specified in the singular. Attributes representing the same type of thing (such as names, dates, and so on) are declared on domains. Domains are the means by which the values of attributes can be formalized. This means that, if we have attributes such as StartDate and EndDate, these are common attributes whose values are comparable. They can therefore be declared on the same domain.

The following is an example of a domain declaration:

domains

IdentifiersOfStudents = s01 to s99

PersonNames = string

Years = 2000 to 2016

IdentifiersOfStaff = t01 to t99

TitlesOfCourses = string

An example of a full domain declaration is shown in the model answers to the labs at the end of this module.

Purpose of Attributes

Attributes, or combination of attributes, make each row (tuple) within a table (entity) unique. In the Person example above, there may be several instances with the same First Name. We therefore need to include the Last Name attribute in combination, to try to achieve this uniqueness. It may be necessary to include other attributes to further ensure uniqueness.

The attributes used in this way are referred to as candidate keys.

Candidate Keys

By definition, tuples are distinct. This implies that, for some attribute, or combination of attributes, within an entity, no two tuples will have the same value for that attribute, or combination of attributes. We call such an attribute (or combination of attributes) K—a candidate key if, and only if, it has the properties of uniqueness and minimality.

By uniqueness, we mean that, at any given time, no two tuples of the entity have the same value for the attribute K. By minimality, we mean that, if K is a combination of attributes, no attribute can be discarded from the combination without destroying its uniqueness.

Every entity has at least one candidate key, by virtue of the distinctness of entities. Note that a candidate key is a constraint that determines what is and what is not allowed in a particular row.

Primary Key

The primary key of an entity is an attribute (or combination of attributes) that has (have) the properties of uniqueness and minimality.

Every primary key is a candidate key. An entity may have more than one candidate key; in this case, just one of these candidate keys is designated as the primary key. The remaining candidate keys are termed alternate keys. When there is only one candidate key, that is the primary key and there are no alternate keys. Where there is a choice between alternate keys for the primary key, the choice is (to a large extent) arbitrary.

Primary and alternate keys are identified in a database schema as part of an entity definition. It is important to include this in a schema, because an alternate key (like a primary key) is a constraint.

Foreign Key

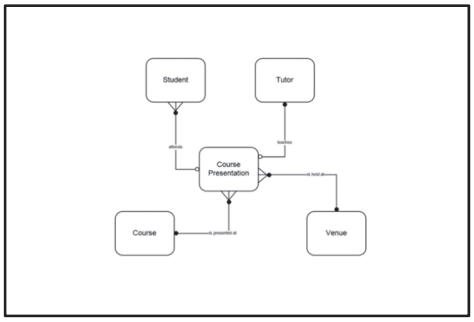
The foreign key is an attribute (or combination of attributes) in one entity whose values are the same as values of the primary key in another.

The major purpose of a primary key is to simplify the way in which the schema represents relationships between entities. This is done by making a primary key in one entity, and a foreign key in another. Note that the foreign key has no uniqueness property for the entity where it has been placed.

The use of foreign keys is the mechanism for representing a one-to-many (1:n) relationship. The entity on the :n side includes, as a foreign key, the primary key of the entity on the 1: side of the relationship.

Relationships

In a conceptual model, a relationship can exist between two entities. It is a named, meaningful link between entities. It represents how the attributes in one entity are related to those in the other and can be thought of as a verb phrase that links the two entities to form a sentence—for example:



- STUDENT attends COURSE PRESENTATION.
- COMPANY sponsors STUDENT.
- COURSE PRESENTATION is one presentation of COURSE TYPE.

We will discuss the types of relationships that can exist (and what they represent) in the next topic. For now, it is sufficient to appreciate that, between two entities, a relationship can define that: for one instance in one entity, there may be none, one, or many related instances in the other. Relationships can be read in both directions. Using the examples above:

- STUDENT attends COURSE PRESENTATION.

  A STUDENT attends one COURSE PRESENTATION, but there may be several STUDENTS on a particular COURSE PRESENTATION.

- COMPANY sponsors STUDENT.

  A COMPANY may sponsor many STUDENTs, but each STUDENT will be sponsored by one COMPANY.

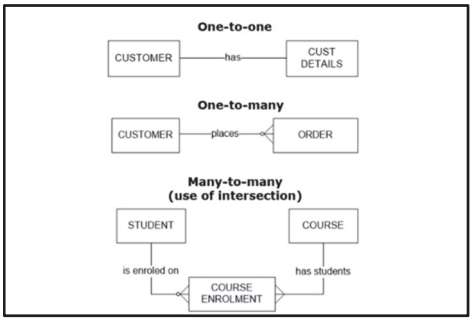
- COURSE PRESENTATION is one presentation of COURSE TYPE.

A COURSE PRESENTATION will be a presentation of one COURSE TYPE on a specific date and at a specific place; COURSE TYPEs may be presented on many different dates and/or at many different places.

When modeling relationships, it is important to not only document the fact that a relationship exists, but also what it means. This removes ambiguity and the potential for misunderstanding, as different people may interpret the relationship differently.

Types of Relationships

Relationships can be of many different degrees. As explained previously, they are implemented by using the primary key from one entity as a foreign key in the other.



For each relationship, we need to specify:

- Name
- Degree
  - One-to-one (1:1)
  - One-to-many (1:n)
    Need to quantify *n*
  - Many-to-many (m:n)
    Need to quantify *m* and *n*
- Optionality
  - How often does each entity participate in the relationship?
    Need to quantify how many times is 'often'.

One-to-One

In a one-to-one relationship, an attribute in one entity is related to one (and only one) attribute in the other. For example, consider two entities, one containing Customer Names, and the other holding Customer Details (address, telephone number, and so on). In this scenario, each Customer will have only one set of details.

One-to-Many

In a one-to-many relationship, an attribute in one entity is related to many attributes in the other. For example, consider two entities, one containing information about Customers, and the other holding details of Orders. It is likely that each Customer might have placed one or more orders (many), but each order has only been placed by one Customer.

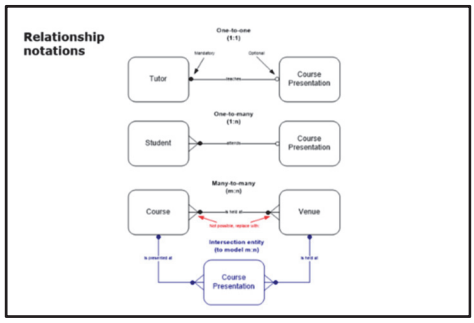
Many-to-Many

In a many-to-many relationship, many attributes in one entity are related to many attributes in the other. For example, consider two entities, one containing information about Students (STUDENT), and the other containing information about Courses (COURSES). It is likely that Students will be enrolled on many courses and each Course will have many Students enrolled on it.

Many-to-many relationships cannot be directly implemented in a relational database because they would violate table rules—in that it would require multiple values in a particular column for a row. By the rules, each row in a table must be atomic; that is to say, for each row, the value within a column is always one value and never a group of values. In a many-to-many relationship, it would be required that a set of values for the primary key in one entity is included as a single value of a foreign key in the other. Such sets of values are not permitted by the logic in the relational model.

Many-to-many relationships are modeled using an intersection entity. In the above example, we could use an entity named COURSE ENROLMENT. There would be a one-to-many relationship between the entities STUDENT and COURSE ENROLMENT, indicating that one Student may be enrolled on many Courses; and a one-to-many relationship between the entities COURSE and COURSE ENROLMENT, indicating that one Course may have several Students enrolled on it.

Notations



There are several different methods for identifying and describing relationships between entities. So far in this module, we have used the symbols shown in the slide. They have the following meanings:

- A single line between two entities indicates a one-to-one relationship.
- The symbol that looks like a crow's foot symbolizes that this end of the relationship is a "many" end as opposed to a "one" end. This symbol is used to define a many-to-one relationship.
- When crows' feet appear at both ends of the relationship line, this identifies a many-to-many relationship. Recall that this type of relationship cannot be implemented, and so an intersection entity is inserted to make two one-to-many relationships.
- An open circle indicates an optional relationship at that end.
- A closed circle indicates a mandatory relationship at that end.

There are other methods for illustrating relationships. For example, the Oracle Case Method uses many of the symbols previously described. The following different symbols are used:

- A straight line indicates a mandatory relationship, while a dashed line indicates an optional relationship.
- A small straight vertical bar appearing perpendicular to the horizontal relationship line at either end indicates that the entity contains a unique identifier as an attribute.
- Non-transferable relationships (small diamond shape): this symbol is used to define a relationship that cannot participate in an either/or relationship.

For entities:

- Mandatory attribute (asterisk symbol *): this symbol identifies attributes that must always contain a value and cannot be NULL.
- Key attributes (the hash symbol #): this symbol is used to identify an attribute of an entity that is a unique identifier.
- Optional attributes (small circle symbol or the letter "o"): this symbol identifies attributes that do not require a value; the "o" is often omitted.

 **Additional Reading:** *Case*Method TM, Entity-Relationship Modelling* by Richard Barker (Addison Wesley).

Verify the correctness of the statement by placing a mark in the column to the right.

Statement	Answer
True or false? You can remove one or more attributes from a multi-attribute candidate key without detriment.	

Lab: Identify Components in Entity Relationship Modeling

Scenario

You are a Business Analyst with a distance learning company. You have been tasked with developing a database to record the information necessary for administering the courses they offer.

You have been given the following statement of data requirements:

> The company needs to keep details of staff and students, the courses that are offered, and the performance of students on these courses. Students are registered with the company before commencing their studies, and are issued with a student identification number. Students are not required to enroll on any course when they register. Students' full names and the date of their registration are recorded. Staff are also issued with a staff number, and their full names are recorded. Each staff member may tutor one or more students on courses. Each student has a tutor for each course on which they are enrolled. Students are allocated a tutor for the course on which they are enrolled at any time after enrollment.
>
> Each course that is available for study is given a course code, a title, and a value for credit (either 0.5 or 1.0). Students are not permitted to enroll on more than three credits worth of courses in any one year. There is a need to record current enrollments only.
>
> Courses may have a quota—the maximum number of students that can be enrolled on the course in any one year. A course may not (yet) have any students enrolled on it. Each course may have up to five assignments that the students are required to complete. These are graded by the tutor assigned to the individual student. The grade for each of these assignments must be recorded as a mark out of 100.

As a Business Analyst, your task is to model these data requirements and develop a conceptual model comprising:

- An ERD, showing entities and relationships.
- A set of entity descriptions, with suitable attributes.
- A statement of the constraints and assumptions made.

This lab can be completed as a paper exercise, but you may want to compare your solutions with the files in the **D:\Labfiles\Lab02\Solution** folder.

Objectives

After completing this lab, you will be able to:

- Identify suitable entities.
- Identify the relationships that need to exist between these entities.
- Identify constraints and assumptions.

Estimated Time: 50 minutes

Ensure that the **MT17B-WS2016-NAT**, **10985C-MIA-DC**, **10985C-MIA-SQL**, and **10985C-MIA-CLI** virtual machines are all running, and then log on to **10985C-MIA-CLI** as **Student** with the password **Pa55w.rd**.

Exercise 1: Identify Entities

Scenario

Using the data requirement statement in the Lab Scenario, you have been tasked to identify and list suitable entities and their attributes.

The main tasks for this exercise are as follows:

1. Create a List of Suitable Entities

2. Add Suitable Attributes

▶ Task 1: Create a List of Suitable Entities

1. Examine the text of the brief.
2. Use the real nouns from it to develop an initial list of entities.
3. Keep these to use for the subsequent tasks and exercises.

▶ Task 2: Add Suitable Attributes

1. Take each entity from the initial list in turn.
2. For each entity, identify suitable attributes for it.
3. For each attribute, develop a suitable domain for its declaration.
4. Add these attributes to the various entities in the list you developed in the previous task.
5. From these attributes, identify the candidate keys and primary keys.
6. Compare your list with **initial_entities.docx** in the **D:\Labfiles\Lab02\Solution** folder.

Results: After completing this exercise, you will have an initial list of entities and attributes that model the data requirements for the brief provided. The entity definitions will include appropriate domains.

Exercise 2: Identify Relationships

Scenario

Using the data requirement statement in the previous Lab Scenario, and the list of entities you created in the previous exercise, you now need to identify and list suitable relationships between these entities. You must develop the relationships, draw an ERD that models them, and resolve any m:n relationships.

The main tasks for this exercise are as follows:

1. Create a List of Named Relationships

2. Draw an ERD Modeling the Entities and Relationships

3. Resolve Any m:n Relationships

▶ Task 1: Create a List of Named Relationships

1. Take each entity in the model in turn.
2. Identify the relationships this has with other entities.
3. List and name these relationships.

▶ Task 2: Draw an ERD Modeling the Entities and Relationships

1. Using the list of entities and relationships, draw an ERD showing each entity and the relationships between them.
2. Remember to name each relationship.
3. Do not forget about optionality and degree for each relationship.
4. Model these relationships by sharing keys.
5. Compare your model with the **initial_ER.docx** in the **D:\Labfiles\Lab02\Solution** folder.
6. What problems do you see for modeling relationships with the model in its current form?

▶ Task 3: Resolve Any m:n Relationships

1. Identify any m:n relationships between entities in your model.
2. Resolve these relationships by a suitable method.
3. Compare your model with the **updated_diagram.xps** in the **D:\Labfiles\Lab02\Solution** folder.

Results: After completing this exercise, you will have an initial list of relationships between the entities that model the data requirements for the brief provided. You will have an initial ERD and have resolved any relationships that cannot be modeled (many-to-many relationships).

Exercise 3: Finalize Your Model

Scenario

Using the data requirement statement in the Lab Scenario and the data model you have developed in the previous exercises, you now need to finalize the model, so that it can be used as part of the development process. You must develop a list of constraints and assumptions, and update the data model accordingly.

The main tasks for this exercise are as follows:

1. Develop a List of Constraints and Assumptions

2. Finalize the ERD

▶ Task 1: Develop a List of Constraints and Assumptions

1. From the data brief, identify constraints for the items being recorded.
2. List any assumptions.

▶ Task 2: Finalize the ERD

1. Update your model to include the constraints and assumptions that you identified in the previous task.
2. Compare your model with the **final_ER.docx** in the **D:\Labfiles\Lab02\Solution** folder.

Results: After completing this exercise, you will have a final data model meeting the original specification.

Question: How do you identify entities?

Question: What is a relationship?

Module Review and Takeaways

 **Best Practice:** Always listen to your customers. You will need to extract their requirements because they might not be able to fully describe what they need. Approach data modeling in a logical manner. Do not try to add detail too early in the process, before you fully understand what is required in general.

Common Issues and Troubleshooting Tips

Common Issue	Troubleshooting Tip
Changes to requirements.	

Review Question(s)

Question: How would you approach data modeling?

Real-world Issues and Scenarios

One of the issues you may encounter might involve changes made to the requirements. These are usually driven by business needs or by an incomplete understanding of the requirements at the start of data modeling. By their nature, changes are frustrating and challenging, but they are a part of the process and, as such, need to be handled and planned for.

Tools

There are several ER modeling tools that can be used for the type of modeling discussed in this module. However, for simple models, Microsoft Visio® is sufficiently powerful.

Module 3

Normalization

Contents:

Module Overview

Normalization is a commonly implemented, powerful technique for creating efficient databases that are best suited to the tasks they will perform. The concepts used in normalization come from mathematical relational theory, and sometimes definitions of normal form use specialist terminology. This module describes normalization and normal form by using examples to clarify the points being made.

Objectives

After completing this module, you will be able to:

- Outline the benefits of normalization and some of the concepts that underpin it.
- Describe the various levels of normal form.
- Explain the benefits of denormalization and outline some techniques for denormalization.

Lesson 1
Fundamentals of Normalization

It is essential to understand the various benefits of normalization so that you know what it is that you are trying to achieve when applying the principles that this module explains. Additionally, to better understand the process of normalization, it is important to ensure that you are familiar with the relevant terminology. This lesson explains the benefits that normalizing a database can bring, and introduces some key terms and concepts.

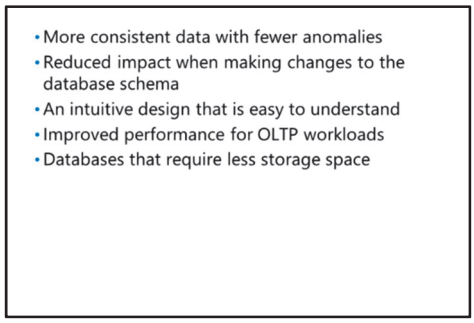
Lesson Objectives

After completing this lesson, you will be able to:

- Describe the benefits of normalization.
- Explain the terms primary key, candidate key, and surrogate key.
- Identify candidate keys.
- Describe the types of dependencies that you might encounter in relational tables.

Benefits of Normalization

Broadly speaking, the process of normalization helps to deliver a database schema that is standardized, and which ensures that data is consistent, with minimal redundancy. When you apply normalization guidelines to a database design, you can achieve many benefits, including:

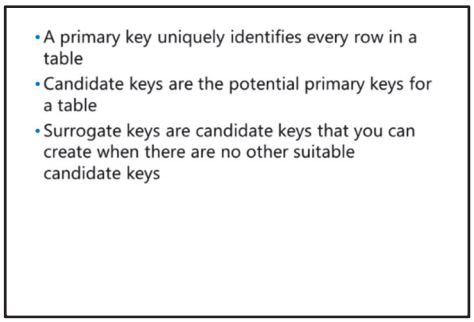


- **A database that is more consistent**. Actions such as modifying or deleting data can result in data anomalies. For example, if you store the contact details for a customer in a Customers table and also store the same contact details in every order for that customer in an Orders table, when you update the address, you must update it in all of these locations, to avoid anomalies. In a normalized design, you would store the contact details only once, so that when you update the contact details, you do not create anomalies.
- **A database that you can extend without having to redesign it**. In a normalized database design, each table represents one entity, and the columns in a table are properties that describe that entity. This makes for a very simple design that should not need to be changed much (if at all) when you make subsequent changes to the database schema, such as adding new entities.
- **An intuitive, easy-to-use database**. A normalized database typically stores data in a way that mirrors real-world processes, which makes it easy for users. For example, a database that models the process of customers placing orders might have a Customers table that contains customer information, a Products table that contains product information, and an Orders table that contains order information by reference to the Customers and Products tables.

- **A database that is optimized to support general online transaction processing (OLTP) workloads**. A normalized database will generally deliver good performance for OLTP workloads, and should be capable of handling many different transactions, including impromptu queries and transactions that might not have been anticipated at design time. However, for some workloads, a fully normalized database might not be the optimal solution. To ensure that designs meet performance requirements, database designers test their designs against sample workloads, and make adjustments accordingly.
- **More efficient use of storage media**. Including less redundant data in a database results in a smaller database that requires less storage space.

Keys

One of the most important principles in relational database design is that every row in a table must be unique so that it can be unambiguously identified. Database designers use keys to achieve this.



Primary keys

A primary key is an attribute value, or a combination of attribute values, that is unique for every row in a table. To ensure that the values in a primary key column are never duplicated, you create a primary key constraint on the primary key column. A primary key constraint automatically prevents the addition of any data value to the column if that value already exists in the column. Primary keys that include more than one column are sometimes referred to as compound keys.

Candidate keys

To select the primary key for a table, a database designer first identifies every column that contains unique data values. Each of these columns is a candidate key for the table, and any candidate key can potentially serve as the table's primary key. Candidate keys can also be combinations of columns in which the combined values are always unique, even if the individual values in each column are not. When identifying candidate keys, you should avoid columns that are configured to allow NULL values, because NULLs are not valid in a primary key. This includes columns that might not currently contain NULLs but whose definition allows them, because this means that the column might contain NULLS at some future point.

Surrogate keys

If it is not possible to identify a suitable candidate key, you can use a surrogate key. A surrogate key is a value that you create to serve as a candidate key, and which has no meaning outside the context of the database. For example, imagine a Products table that includes a Product Number column that you identify as a candidate key. Product number values are used in other applications to identify products, so they have meaning beyond the database context.

On investigation, you discover that product numbers are occasionally reused to represent different products, meaning that the Product Number column is not a good candidate key because values are not guaranteed to be unique. Instead, you could create a column called Product ID in the Products table to act as a surrogate key. Product ID would contain a unique value for each row, helping you to differentiate between all products, even those that have the same product number. Surrogate key columns often simply contain integer values that the database generates automatically.

 **Note:** In contrast to a surrogate key, a candidate key that represents a value that exists in the real world, such as the product number in the earlier example, is sometimes called a natural key. When identifying candidate keys, you should be cautious about assuming that natural key values are unique. For example, you might reasonably assume that values such as social security numbers and passport IDs are unique within a country—but they may not be unique in a worldwide context.

Demonstration: Identifying Candidate Keys

In this demonstration, you will see how to identify candidate keys in a table.

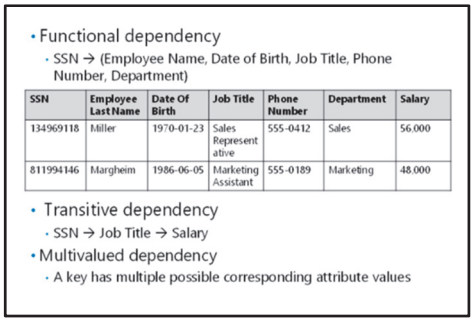
Demonstration Steps

1. Start the **10985C-MIA-DC** and **10985C-MIA-SQL** virtual machines, and then log on to **10985C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. In the **D:\Demofiles\Mod03** folder, right-click **Setup.cmd**, and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**, and wait for setup to complete.
4. Open **Microsoft SQL Server Management Studio**, and then connect to the **MIA-SQL** instance of the database engine by using Windows Authentication.
5. On the **File** menu, point to **Open**, and then click **File**.
6. In the **Open File** dialog box, navigate to the **D:\Demofiles\Mod03** folder, click **Candidate Keys.sql**, and then click **Open**.
7. Under the comment **View the columns in the Production.Product table and identify the total number of rows**, select the Transact-SQL statement, and then click **Execute**.
8. Review the results, and in the results pane, in the bottom right corner, note the number of rows that the query returned.
9. Under the comment **Assess ProductID as a candidate key**, select the Transact-SQL statement, and then click **Execute**.
10. Review the results, and in the results pane, in the bottom right corner, note that the number of rows that the query returned is the same as for the query in step 7. This means that **ProductID** is a candidate key because it contains a unique value for each of the rows in the table.
11. Under the comment **Assess Name as a candidate key**, select the Transact-SQL statement, and then click **Execute**.
12. Review the results, and in the results pane, in the bottom right corner, note that the number of rows that the query returned is the same as for the query in step 7. This means that **Name** is a candidate key because it contains a unique value for each of the rows in the table.
13. Under the comment **Assess ProductNumber as a candidate key**, select the Transact-SQL statement, and then click **Execute**.

14. Review the results, and in the results pane, in the bottom right corner, note that the number of rows that the query returned is the same as for the query in step 7. This means that **ProductNumber** is a candidate key because it contains a unique value for each of the rows in the table.
15. Under the comment **Assess Color as a candidate key**, select the Transact-SQL statement, and then click **Execute**.
16. Review the results, and in the results pane, in the bottom right corner, note that the number of rows that the query returned is less than the number of rows that the query in step 7 returned. This means that **Color** is not a candidate key because it does not contain a unique value for each of the rows in the table.
17. Close SQL Server Management Studio without saving changes.

Dependencies

In addition to ensuring uniqueness, another goal of normalization is to ensure that non-key values in tables have a dependency on the key. Non-key values are the values that are present in a table but which do not form part of the primary key. Dependency means that the values of non-key attributes are determined by the key itself. Identifying dependencies is part of the process of normalization, and helps designers to decide which columns to include in a table and which ones to move to other tables.



Functional dependency

An attribute is said to have a functional dependency on the key when it is possible to determine the attribute's value solely from the key. For example, imagine an Employees table that has a primary key column called SSN, which contains an individual employee's social security number. Social security numbers are government issued values that are intended to uniquely identify an individual for social security and tax purposes—so they are good candidate keys.

Other columns in the Employees table include Employee Last Name, Date of Birth, Job Title, Phone Number, and Department. The SSN value in a row determines the values in the other columns; every time you look up a particular SSN, it will always return the same values for these columns. This relationship is often expressed by using the notation A → C, which means that the value A determines the value C. In the Employee table example, the following dependencies exist:

- **SSN → Employee Last Name**
- **SSN → Date of Birth**
- **SSN → Job Title**
- **SSN → Phone Number** (this assumes that an employee has only one phone number)
- **SSN → Department**

You can also summarize these dependencies as follows:

- **SSN** → (**Employee Name**, **Date of Birth**, **Job Title**, **Phone Number**, **Department**).

Note that functional dependencies do not work in reverse. For example, Job Title → SSN does not work because the value in the Job Title column could be the same in many rows, so looking up a particular job title value is not guaranteed to return the same, single SSN value every time. Similarly, Employee Last Name → Department is not a valid dependency. The company might employ several people with the same last name, who work in different departments, so looking up a name would not reliably return the same department value.

Determining functional dependencies in tables requires knowledge of the real world. For example, if you did not know the real-world fact that an SSN is uniquely associated with an individual, you would not be able to determine the functional relationships in the earlier example. If SSNs were shared in the real world, then returning any given SSN would not be guaranteed to return the same attribute values every time—so there would not be a functional dependency.

Transitive dependency

Transitive dependencies are indirect dependencies that can be summarized as A → B → C. This means that the value of C is not directly dependent on the key value A, but only indirectly dependent on it. For example, in the Employees table previously shown, imagine that there is also a column called Salary, and that the salary that an employee earns depends on their job title, not on the individual. The dependency SSN → Salary is not valid in this case, and you would correctly express the relationship as SSN → Job Title → Salary.

The problem with transitive dependencies is that, if you remove or change the intermediate value, the relationship between the other values is no longer valid. In the Employees table example, changing the Job Title value for a row would break the link between the SSN and Salary values, resulting in inconsistent data. Furthermore, it would be possible to have employee rows with the same Job Title value, but with different Salary values—and that would not be correct.

To correctly identify dependencies, it is essential to understand the data. To identify the transitive dependency in the earlier example required knowledge about how salaries were linked to job titles. Without this knowledge, we might have assumed that salaries were linked to the individual, and consequently that the dependency SSN → Salary was valid.

Multivalued dependency

Multivalued dependency refers to the situation where the value in a key column can have multiple possible corresponding values in attributes that depend upon that column. This contrasts with functional dependencies, which always return the same value for any given key value. For example, imagine a table with three columns: Training Courses, which stores training course codes; Trainers, which stores the names of the trainers who deliver the courses; and Course Books, which stores the titles of the course books. Any training course can be delivered by multiple different trainers, and each trainer can deliver multiple courses; so each time a training course appears in a table, it might be associated with a different trainer value. In other words, for any given value of Training Course, there are multiple possible values for Trainers.

The relationship between Training Courses and Trainers is an example of a many-to-many relationship. A second many-to-many relationship exists between Training Courses and Course Books because a course can use more than one book, and any book might be used in multiple courses. To represent the various possible combinations of courses, trainers, and books, the same data values will appear in the table multiple times. For example, there might be rows including:

- Course1, Trainer1, Book1
- Course1, Trainer1, Book2
- Course1, Trainer2, Book1
- Course1, Trainer2, Book2

The multivalued dependencies are Training Courses → Trainers and Training Courses → Course Books. For example, for the Training Course value "Course1", values of the dependent attribute Trainers include Trainer1 and Trainer2. This situation leads to redundancy because the same values are stored multiple times.

Check Your Knowledge

Question	
What type of key can you use when you cannot identify an appropriate primary key from the existing columns for a table?	
Select the correct answer.	
	Composite primary key
	Surrogate key
	Candidate key

Lesson 2
Normal Form

You can normalize a database by implementing a set of guidelines that are collectively referred to as "normal form". The majority of database designs only take account of the first three levels of normal form—called first normal form, second normal form, and third normal form—because they are more widely applicable than the others. When a database complies with, for example, first normal form, the database is said to be "in first normal form". The levels of normal form are cumulative—this means, for example, that for a database to be in second normal form, it must also be in first normal form. This lesson explains the different levels of normal form and how to implement them.

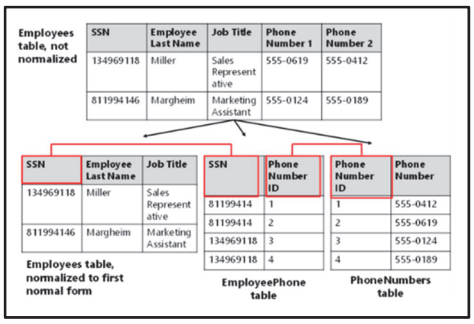
Lesson Objectives

After completing this lesson, you will be able to:

- Explain first normal form.
- Explain second normal form.
- Explain third normal form.
- Normalize to third normal form.
- Explain Boyce-Codd Normal Form (BCNF).
- Explain fourth normal form.
- Explain fifth normal form.

First Normal Form

For a database to be in first normal form, its tables should not contain repeating groups of data; each row in a table should be unique, and have the same number of columns. A primary key ensures that all rows are unique, and the table structure itself determines the number of columns, so this topic focuses on the elimination of repeating groups of data.



Repeating data groups

Continuing with the Employees example from the previous lesson, imagine that you want to store more than one phone number for each employee. You could implement this by including columns called Phone Number 1 and Phone Number 2 in the Employees table. This is an example of a repeating data group—you can see that problems will quickly arise with this table if any employee has a third phone number because, to accommodate this, you will need to add a new column. Also, rows should not contain repeating groups within a column. Data in a column should be atomic, which means that each row can contain one, and only one, value for that column. For example, a row in an Employees table should not contain a value for a Phone Number column that comprises a comma-separated list of three telephone numbers.

When assessing whether a table complies with first normal form, you can use the following checks:

- Are there any columns that contain data that includes separator values, such as commas?
- Are there any columns with similar names, perhaps differentiated by numbers at the end, such as Address1, Address2, and so on?
- Is there a column or set of columns that uniquely identifies each row?

Putting a table in first normal form

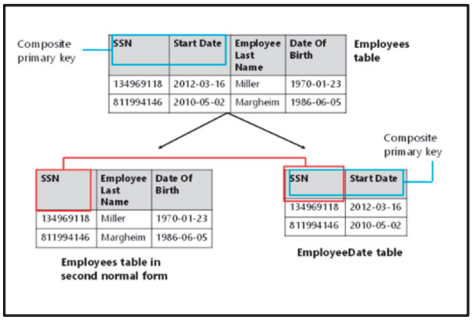
To handle repeating data groups correctly and put a table into first normal form, you could remove the phone number values from the Employees table, and create a PhoneNumbers table to store the numbers.

Note: Breaking data out from a single table into multiple tables is called "lossless decomposition", and is a large part of the process of normalization. When you decompose the tables into multiple tables, the process is lossless because you never remove any data from the database; you just relocate it to different tables.

The new PhoneNumbers table would have two columns, a unique primary key identifier called Phone ID and a column called Phone Number. Neither of these tables contain repeating data groups, but both have unique identifiers and a fixed number of columns—so both are in first normal form. However, this still leaves you with a problem: how do you associate the phone numbers with the employees? You cannot simply add columns to the Employees table again to contain the primary key values for the associated phone numbers because this defeats the purpose of removing the phone numbers in the first place. To solve this problem, you would create a third table called EmployeePhone with two columns, called Employee ID and Phone ID. Each Employee ID could appear in the table multiple times, each time with a different Phone ID. This also allows a given Phone ID to appear with multiple Employee ID values, which reflects the real-world situation that some phone numbers might be shared by employees. In the EmployeePhone table, the combination of the Employee ID and Phone ID columns would be unique for each row, so it is a candidate key. You could also add a dedicated surrogate key column to the table if required. To ensure that the values in the EmployeePhone table are valid, you would use foreign key constraints.

Second Normal Form

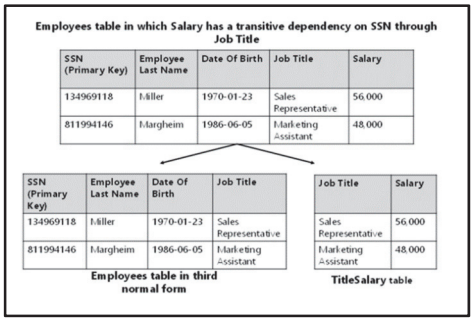
For a table to be in second normal form, it should be in first normal form; additionally, all non-key columns in the table should be functionally dependent on the whole of the primary key. When a table's primary key consists of a single column, then all other columns are automatically dependent on it. However, if you have a composite primary key, this may not be the case. In this situation, the result might be that your table contains data that repeats across multiple rows within the table.



For example, imagine a variation on the Employee table, in which there is a composite primary key that consists of the SSN column and another column called Start Date. The Start Date column records the date that an employee started working at the organization. Employees frequently leave and then return to the company—this means that the organization can differentiate between the different periods of employment. The other columns in the Employees table, Employee Last Name, Date of Birth, and Address, all have a dependency on the SSN, but they do not depend on the Start Date. The problem with this situation is that it leads to redundancy. Every time you add a new row for an employee who has worked for the organization before, you would need to repeat all of the Employee Last Name, Date of Birth, and Address data that was in the previous row for that employee.

To eliminate this problem, you could decompose the table into two tables: Employees, which would contain the SSD, Employee Last Name, and Date of Birth columns, and EmployeeDate, which would contain the SSD and Start Date columns. The combination of SSN and Start Date form the primary key in the EmployeeDate table, and SSN is the primary key in Employees. The Employee Last Name and Date of Birth columns now depend entirely on the key value. A foreign key constraint ensures that the SSN values in EmployeeDate are consistent with the SSN values in Employees.

Third Normal Form



For a table to meet the requirements of third normal form, the table should be in second normal form, and there should be no transitive dependencies between non-key attributes on the primary key. Imagine that the Employees table includes the columns SSN, which is the primary key, Employee Last Name, Date of Birth, Job Title, and Salary. In the company, an individual's salary depends on their job title, so everyone whose title is Sales Representative earns the same salary. This means that the Salary column in the Employees table is transitively dependent on the primary key, through the Job Title column. The transitive relationship can be summarized as SSN → Job Title → Salary. This is potentially a problem because there is nothing to prevent the addition of rows to the Employees table that have the same value in the Job Title column, but different values for the Salary column.

To solve this problem, you would decompose the table into two tables:

- An Employees table with only the columns SSN, which is the primary key, Employee Last Name, Date of Birth, and Job Title.
- A TitleSalary table, with the columns Job Title and Salary.

Now the salary for any given job title is stored only once, in the TitleSalary table, and referenced from the Employees table. Consequently, there is now no possibility that different salaries could be added for the same job title.

Demonstration: Normalizing to Third Normal Form

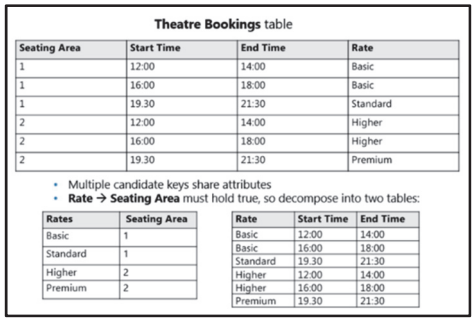
In this demonstration, you will see examples of:

- Normalizing to first normal form.
- Normalizing to second normal form.
- Normalizing to third normal form.

Demonstration Steps

1. In the **D:\Demofiles\Mod03** folder, double-click the **Normalize Book Data.xlsx** file.
2. On the **Raw Data** worksheet, review the data and note the following points:
   a. The worksheet contains data about a collection of books.
   b. There are multiple copies of the same book titles, and each copy might be a different pressing. This information is stored in repeating data group columns called **Copy 1**, **Copy 2**, and **Copy 3**.
3. On the **1NF** worksheet, note the changes that have been made to bring the table into first normal form. These changes include:
   a. Adding a composite primary key consisting of the **Book Title**, **Copy Number**, and **Pressing** columns.
   b. The repeating data group columns have been removed.
4. Note that the columns **Year Published**, **Author**, **Author Year of Birth**, and **Author Age** all contain repeating data values. This is because these columns are not functionally dependent on the whole of the primary key.
5. On the **2NF** worksheet, note the changes that have been made to conform to second normal form. The table has been decomposed into four tables: **Book**, **Author**, **Copy**, and **Pressing**. The columns in each table depend on the whole of the primary key in their respective tables.
6. On the **3NF** worksheet, note the change that has been made to ensure that the **Author** table conforms to third normal form. The **Author Age** column has been removed because it was transitively dependent on the primary key, through the **Author Date of Birth** non-key column. If required, the age of each author can be calculated by using the date of birth value, so it not necessary to store it.

Boyce-Codd Normal Form



There are additional levels of normal form that you can apply during the design phase to normalize database schemas, but they are not as commonly used as the first three normal forms. BCNF is a refinement of second normal form and third normal form that addresses relatively unusual scenarios that third normal form does not fully deal with, and which can result in data inconsistencies. Specifically, BCNF helps you to normalize when you have multiple composite candidate keys that have at least one attribute in common. To be in BCNF, a table must also be in third normal form. Consider the **Theatre Bookings** table below:

Seating Area	Start Time	End Time	Rate
1	12:00	14:00	Basic
1	16:00	18:00	Basic
1	19.30	21:30	Standard
2	12:00	14:00	Higher
2	16:00	18:00	Higher
2	19.30	21:30	Premium

There are two seating areas, one of which has a better view than the other, so it has a higher rate. However, the cost of booking a ticket also varies with the time of the performance; the later performance time being the most expensive. There are four candidate keys in the table:

- Seating Area and Start Time.
- Seating Area and End Time.
- Rate and Start Time.
- Rate and End Time.

The candidate keys have attributes in common—for example, End Time is part of two candidate keys. Every attribute is part of a candidate key, so there are no non-key attributes present in the table. Consequently, because second normal form and third normal form deal with dependencies of non-key attributes on candidate keys, the table is already in third normal form. However, there is another dependency in the table that is problematic. Any given Rate value is always associated with the same Seating Area value, so the dependency Rate → Seating Area must hold for every row. With the limited data set in the previous table, the dependency does appear to hold. In fact, it would be possible to enter a booking for seating area 1 with a rate of Premium, which should not logically be possible, and which would make Rate → Seating Area invalid.

To prevent this, you would decompose the table into the following two tables:

Rates

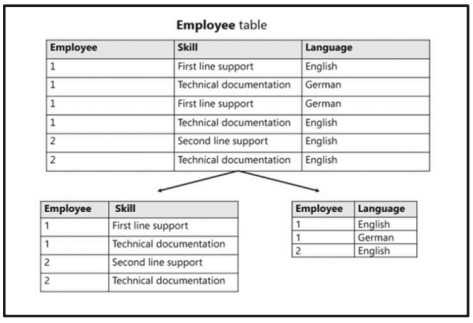
Rate	Seating Area
Basic	1
Standard	1
Higher	2
Premium	2

Bookings

Rate	Start Time	End Time
Basic	12.00	14.00
Basic	16.00	18.00
Standard	19.30	21.30
Higher	12.00	14.00
Higher	16.00	18.00
Premium	19.30	21.30

Now it is not possible to enter a booking for a rate that is incompatible with the seating area, and the tables are in BCNF. In limited situations, you might also normalize to fourth normal form and fifth normal form.

Fourth Normal Form



Multivalued dependencies can lead to the storage of redundant data. Fourth normal form involves eliminating multivalued dependencies from a table by decomposing the table into multiple tables. To be in fourth normal form implies that a database is already in BCNF.

The following Employee table has the columns Employee, Skill, and Language. One employee can have multiple skills, such as delivering first line support, or writing technical documentation; similarly, one skill can be possessed by multiple employees. Consequently, the relationship between Employee and Skill is a many-to-many relationship. Employee and Language is also a many-to-many relationship, because an employee may speak multiple languages and any language may be spoken by multiple employees. The multivalued dependencies Employee → Skill and Employee → Language exist in the table because, for any given value of Employee, there are multiple possible values for Language and Skill.

Employee	Skill	Language
1	First line support	English
1	Technical documentation	German
1	First line support	German
1	Technical documentation	English
2	Second line support	English
2	Technical documentation	English

The table redundantly stores multiple Skill values for each language spoken (or conversely, multiple Language values for each skill). To eliminate this redundancy, you can decompose the table to create two new tables:

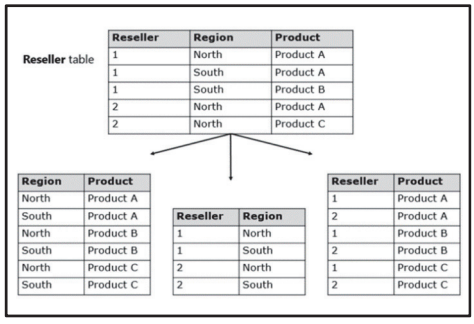
- EmployeeSkill, which contains the Employee column and the Skill column.
- EmployeeLanguage, which contains the Employee column and the Language column.

Now each skill and each language is stored only once per employee.

An important point to note is that, once again, knowledge of the real-world data is essential when deciding whether to decompose a table. In the previous example Employee table, there is an assumption that each employee can perform each skill in every language. As a result, it is necessary to add a row for every combination of skill and language for each employee, which leads to redundancy. However, imagine if some skills could only be performed in one language; for example, employee 1 can do first line support and technical documentation in English.

However, in German, they can only do first line support because their written skills are not good enough for technical documentation. The row that combines technical documentation and German is therefore not required, and so data redundancy is reduced. If this were a very common scenario with other employees' skills and languages, you might consider leaving the table in third normal form or BCNF, and not decomposing it to comply with fourth normal form.

Fifth Normal Form



Putting a database into fifth normal form involves decomposing tables to the point where it is not possible to decompose them further without losing information; in other words, until the decomposition is no longer lossless. With lossless decomposition, it is always possible to reconstruct any table that you decompose by using an SQL query, and specifying the columns and rows that you require from the tables. SQL queries return result sets as tables. However, if you decompose tables to the point that you lose information, it is no longer possible to reconstruct that information.
To be in fifth normal form implies that tables are already in fourth normal form.

The following Reseller table stores data about the resellers that a company uses to sell its products. The table contains data about two resellers, two regions in which the resellers operate, and three products. Resellers can operate in either region, or in both, and they can sell all of the three products, or only some of them.

Reseller	Region	Product
1	North	Product A
1	South	Product A
1	South	Product B
2	North	Product A
2	North	Product C

Note the following points about the table:

- There is one candidate key, consisting of Reseller, Region, and Product.
- The table is in fourth normal form because there are no multivalued dependencies.
- Reseller 1 operates in both the North and South regions, and sells only products A and B.
- Reseller 2 operates only in the North region, and sells only products A and C.
- The table does not store other possible combinations, such as reseller 1, North, and Product C because, in the real world, this combination is meaningless—reseller 1 does not sell Product C.

In this scenario, it is appropriate to leave the table as it is because only limited subsets of the possible combinations of attribute values are stored. This leads to some repetition of data, but this is not redundant data—it is required to maintain the information about which resellers sells which products, in which regions.

However, if the real-world situation was that *all* resellers operate in *all* regions and sell *all* products, then the table would need to store all possible combinations of attribute values to accurately represent this fact. This would lead to the storage of redundant data because, if all combinations are valid, this can simply be assumed; there is no need to store all of the combinations. To eliminate the redundant data, you could decompose the table into three tables, each of which has two columns, as follows:

- ResellerRegion, which contains the Reseller and Region columns to represent all possible combinations of these columns.
- RegionProduct, which contains the Region and Product columns to represent all possible combinations of these columns.
- ResellerProduct, which contains the Reseller and Product columns to represent all possible combinations of these columns.

In practice, normalizing to fourth normal form and fifth normal form is relatively uncommon; most transactional relational databases are normalized to third normal form or BCNF at most.

Categorize Activity

Match each description with the correct level of normal form. Indicate your answer by writing the category number to the right of each item.

Items	
1	Provide a primary key as a unique identifier for rows, and eliminate data groups that repeat across columns.
2	Ensure that all non-key columns are functionally dependent on the whole of the primary key.
3	Eliminate attributes that are only transitively dependent on the primary key.

Category 1		Category 2		Category 3
First normal form		Second normal form		Third normal form

Lesson 3
Denormalization

When a database will be used primarily for create, read, update, and delete (CRUD) operations, using the normalization guidelines described in the preceding lesson will generally result in better performance. However, it is not always the case that normalization will produce a better performing database, so you should regard normalization and normal forms as guidelines rather than rules when designing a database schema.

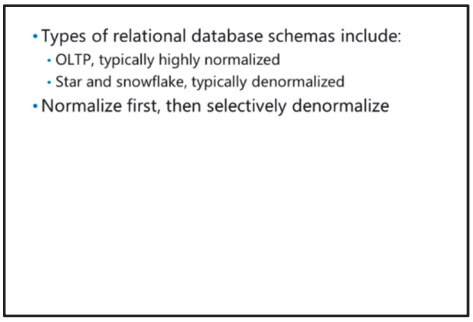
Lesson Objectives

After completing this lesson, you will be able to:

- Explain what denormalization means.
- Describe the benefits of denormalization.
- Use denormalization techniques.
- Describe some considerations for denormalization.

Introduction to Denormalization

Decomposing tables to implement the various levels of normal form is standard practice for OLTP databases, but it is important to be aware of situations in which normalization is not optimal, and can actually be counter-productive.



Relational database schemas

Broadly speaking, there are two types of commonly used database schema, and a key difference between them is the extent to which they are normalized.

- **OLTP schemas, which are used in databases that principally process CRUD operations**. These databases generally benefit from normalization because normalizing databases minimizes redundancy and modification anomalies; these can occur when the same data value exists in more than one location.
- **Star and snowflake schemas, which are used in data warehouses**. Data warehouses support analytical operations, and experience almost exclusively read-based workloads, with periodic data loads to bring them up to date. These databases are generally not highly normalized because storing redundant data can improve read performance.

It is probably best to regard these two types of schema as two extreme endpoints on a spectrum, with most databases sitting somewhere in between; albeit that many databases will be closer to one end or the other. You should not simply assume that, because you are designing an OLTP database, you should always normalize it as much as possible.

Normalize then denormalize

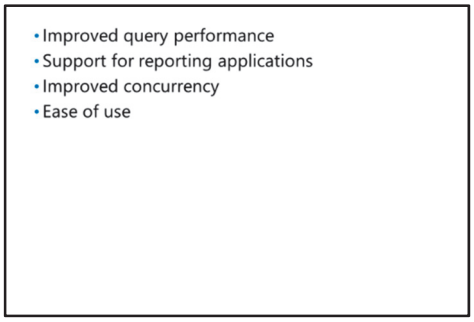
Denormalization is the process of adding redundancy back in to a database schema that has already been normalized; a denormalized database is not simply a database that has never been normalized. You can achieve denormalization in various ways, including:

- Re-introducing tables that were decomposed during the normalization process.
- Duplicating attributes across multiple tables.
- Duplicating entire tables or parts of tables.
- Storing derived data that you can calculate from existing data.

The optimal design for any given database is highly specific, and depends on the way that it will be used. For example, if you are designing a schema for an OLTP database, but you know that certain attributes will also be used frequently for queries, you should normalize the database for OLTP performance, and then consider whether to selectively denormalize for those attributes. Denormalization usually reduces the number of joins needed to process queries. Reintroducing redundancy in this way is likely to have an adverse effect on CRUD operations, and a good design will find the right balance between read performance and CRUD performance.

Benefits of Denormalization

Normalizing a database through lossless decomposition reduces data redundancy; this then reduces the amount of data in the database. However, by definition, decomposing tables increases the number of tables in the database. In other words, a highly normalized database will be smaller in terms of data volume, but it will have a more complex schema than a version of the same database that has not been normalized. This complexity can lead to various problems that you can address through denormalization.



Improved query performance

A normalized schema can improve performance for databases that perform CRUD operations. When there is no redundant data, you only need to perform actions such as updates once, instead of once per occurrence of the data item. However, if the database needs to support a large amount of query activity, a normalized schema can impede response times. This is because data is spread out across many tables, and recovering the required data involves looking up rows in one table, and then matching them against rows in a second table, in a process called a table join. Queries can often require joins that involve many tables, and the process can be relatively slow when compared to retrieving rows from just a single table or a couple of tables. Denormalizing a database can reduce the number of joins required to process a given query, which improves performance. An important part of planning denormalization is understanding the types of queries that the database will experience, so that you can identify where denormalization might be useful.

Support for reporting applications

Report applications often generate very complex queries that might include calculations and aggregations, as well as numerous joins. These types of operations can take a significant amount of time, particularly with very large databases (VLDBs). Denormalization can improve performance for these types of queries, making reporting faster and more efficient.

Improved concurrency

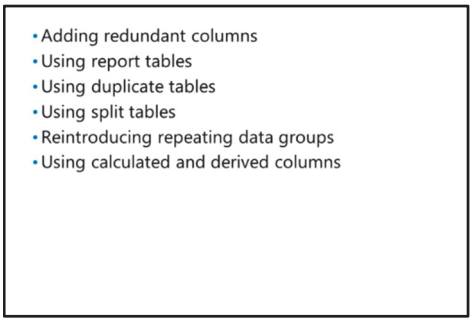
When users attempt to access data concurrently, an RDBMS must ensure that inconsistencies don't occur. For example, if one user reads a data value from a table, and a second user simultaneously updates the same data value, the first user's data is now out of date, and any subsequent decisions that they make using this data will be based on inaccurate information. RDBMSs use mechanisms such as locking to handle these kinds of situations. Locks prevent applications from making changes to data if that data is being used by other applications. When many users need access to the same tables at the same time, locking can impede performance because applications have to wait until resources are released before making changes. By introducing redundant data, you can potentially reduce the impact of locks and improve performance.

Ease of use

Users who query a database need to understand the database schema so that they can locate the required attributes and understand how those attributes relate to attributes in other tables. A denormalized database can reduce the complexity of the schema for users, making it easier to work with. An alternative to denormalization that addresses this specific problem is to use views. A view is a database object that provides a view of a part of a database that is easier to understand. When using a view, a user might see only one logical table when, in reality, this logical table is comprised of several database tables. However, views do not address performance or concurrency issues that arise from normalization.

Techniques for Denormalizing Databases

There are several techniques that you can use to denormalize databases to help address various different challenges.



Adding redundant columns

If queries frequently perform table joins to return the required columns, it can be helpful to duplicate one or more of the attributes so that one table contains them all. This approach eliminates the cost of performing the table join, and can improve query times. For example, imagine a table called OrderLineItems that has columns including OrderNumber, ProductID, UnitPrice, and Quantity, and a Products table that includes the ProductID, Size, and ProductName columns. Queries frequently generate a result set that includes the columns in the OrderLineItems table, and the ProductName column from the Products table. To return all of these columns in a single result set entails joining the two tables. By storing the ProductName column in the OrderLineItems table, in addition to the Products table, you remove the need to join the tables for this type of query, which can improve performance. Implementing redundant columns violates second normal form because, every time you add a new row to the OrderLineItems table, you must add the ProductName value, which will consequently repeat across multiple rows.

An important point to note with this approach is that future insert, update, and delete operations must be performed in both tables to maintain the integrity of the data.

Report tables

You create a report table particularly to handle queries for a specific report or set of reports. Reports can generate queries that require a great deal of processing power and take a long time to run, because they often contain multiple joins and complex calculations. A report table, also sometimes called a pre-joined table, is a table that contains all of the attributes and calculations required to service reporting queries, so that the joins and calculations do have to be performed when the report is generated. Because a report is typically a snapshot in time, it is not usually necessary to keep report tables synchronized with other tables—you would typically repopulate the report table periodically.

Duplicate tables

A duplicate table, also called a mirror table, is an exact copy of an existing table. You would usually create a duplicate table to address issues with contention. For example, imagine a table that is heavily used by two applications, one that inserts new data and updates existing data, and one that reads data. In this scenario, there will be frequent occasions when one application must wait for the other to complete its actions before it can perform its own actions. By creating an exact copy of a table and dedicating each copy to one of the applications, you can reduce contention, and speed up response times.

Split tables

A split table is a single table that is divided in two, based on the needs of the applications that use the table. You can create split tables when two or more applications exclusively use different parts of the table. Contention exists in this situation because applications can cause the entire table to be locked. When you split a table, you place all of the columns used by one application in the first table, and the columns used by the other application in the second table. However, both tables need to store the same primary key value, so that you can logically recreate the original table by using a query.

The scenario just described is called a vertical split because it involves splitting the table by columns. An alternative is to split horizontally, by row. This might be useful if applications access different rows in the same table—for example, one application might deal with current data rows, and a second application might deal with historical data rows. When you split a table, you can either remove the original table, or retain it in addition to the new tables—for example, so that a third application can use it. Splitting tables is also sometimes referred to as over-normalizing because it involves the lossless decomposition of a table.

Repeating data groups

To be in first normal form, a table must not contain any data groups that repeat across columns. For example, an Employee table that includes one column for each phone number that an employee has is not in first normal form. Decomposing the table to put it into first normal form results in a single column that stores the same data. For example, instead of multiple columns for storing phone numbers, there should be only one column, with the phone numbers stored on separate rows. However, storing more rows can negatively affect performance for queries that return a large number of rows. This is because it increases the number of disk reads required to locate and read the rows from disk. In this situation, you could choose to break first normal form and reintroduce the repeating data group columns to improve response times.

Calculated and derived columns

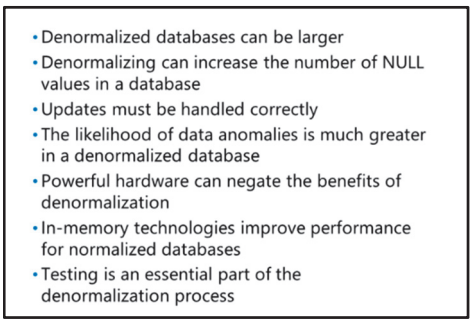
Queries that mathematically calculate values based on the data in a table, or which use string concatenation to derive new values, can cause response times to increase. String concatenation is the process of taking two pieces of text data and joining them together to create a single piece of text data.

For example, you can use the FirstName and LastName columns in an Employee table and concatenate them in a query to create a FullName column. Adding a calculated column or derived column that stores this information in the table can speed up response times, because it is not necessary to concatenate data or perform calculations at query time. When you add a column whose values derive from other non-key columns in the table, you increase redundancy; furthermore, because this process involves adding a transitive dependency, the table will no longer meet the requirements of third normal form.

All of the techniques that this topic describes are advanced, and using them requires a solid understanding of the concepts involved, in addition to the knowledge of the data and the types of queries associated with a specific database. Additionally, any schema designs should be thoroughly tested before implementation to ensure that they meet the particular requirements for the database.

Considerations for Denormalization

When you are planning denormalization, consider the following points:



- **Storage space**. Denormalization reintroduces redundancy to the database design, which means that the database will be bigger and require more storage space. When duplicating tables, this can be a significant increase.
- **NULL values**. Denormalizing can increase the number of NULL values in your database, which might lead to complications when working with the data. NULL values require special handling to ensure that data is represented correctly.
- **Updates**. Updates to denormalized data can be problematic because you need to update each piece of data consistently—this can be difficult because the data item might be stored multiple times. Remember that eliminating this kind of redundancy is one of the key reasons why you normalize databases in the first place. For example, with duplicate tables, you have two copies of the same data. You need to either ensure that the application code performs inserts, updates, and deletes updates on both tables at the same time; or, if some latency is acceptable, that the tables are synchronized on a periodic schedule.
- **Data anomalies**. When a database stores redundant data, the likelihood of data anomalies is much greater because of the difficulty of updating data consistently.
- **Hardware**. The more powerful the hardware, the smaller the effect that denormalization will have on performance. For example, servers that have large amounts of memory to cache query results will be able to deliver fast query response times for similar queries, even for highly normalized databases.
- **In-memory technologies**. In-memory technologies, such as In-Memory OLTP and Columnstore indexes in SQL Server, make it easier to maintain a normalized database without incurring the query response time cost—this minimizes the need to denormalize. However, these technologies are not suitable for every database.
- **Testing**. Before introducing denormalized tables into a production environment, it is essential that you test query response times to ensure that the cost of denormalization is offset by the benefits.

Verify the correctness of the statement by placing a mark in the column to the right.

Statement	Answer
True or false? A denormalized database and an unnormalized database are essentially the same thing.	

Lab: Normalizing Data

Scenario

The retail company that you work for uses a spreadsheet to track customer orders, but as the number of orders increases, the data has become more difficult to maintain. Management have decided to switch to a relational database. You have been charged with designing a new database schema for the orders data. You have obtained a reduced copy of the current orders spreadsheet that contains a small representative sample of the data; you will use this to help you to design the schema.

Objectives

After completing this lab, you will have:

- Normalized a data set to first normal form.
- Normalized a data set to second normal form.
- Normalized a data set to third normal form.
- Denormalized a data set.

Estimated Time: 60 minutes

Ensure that the **MT17B-WS2016-NAT**, **10985C-MIA-DC**, **10985C-MIA-SQL**, and **10985C-MIA-CLI** virtual machines are all running, and then log on to **10985C-MIA-CLI** as **Student** with the password **Pa55w.rd**.

Exercise 1: Normalizing to First Normal Form

Scenario

Having obtained the sample data set, you will analyze it and normalize it to first normal form.

The main tasks for this exercise are as follows:

1. Review the Data Set

2. Normalize the Data Set to First Normal Form

▶ Task 1: Review the Data Set

1. Ensure that the **10985C-MIA-CLI** virtual machine is running, and then log as **Student** with the password **Pa55w.rd**.
2. Review the data on the **Raw Data** tab in **D:\Labfiles\Lab03\Starter\Normalize to 1NF.xlsx** and note any aspects that cause it to violate first normal form.

▶ Task 2: Normalize the Data Set to First Normal Form

1. Decompose the table in the **Raw Data** worksheet to create a new version of the data that is in first normal form. Document your solution in the **Normalize to 1NF** worksheet.
2. On the **1NF Solution** worksheet, review the suggested solution.
3. Close **Normalize to 1NF.xlsx**, saving any changes if prompted.

Results: After completing this exercise, you will have normalized a data set to first normal form.

Exercise 2: Normalizing to Second Normal Form

Scenario

Having normalized the data set to first normal form, you will now normalize it to second normal form.

The main tasks for this exercise are as follows:

1. Review the Data Set in First Normal Form
2. Normalize the Data Set to Second Normal Form

▶ Task 1: Review the Data Set in First Normal Form

- In the **D:\Labfiles\Lab03\Starter** folder, open **Normalize to 2NF.xlsx**, and then on the **Normalize to 2NF** tab, review the table to identify any features of the data that might cause the table to violate second normal form.

▶ Task 2: Normalize the Data Set to Second Normal Form

1. Decompose the table in the **Normalize to 2NF** worksheet to create a new version of the data that is in second normal form. Document your solution in the free space in the **Normalize to 2NF** worksheet.
2. On the **2NF Solution** worksheet, review the suggested solution.
3. Close **Normalize to 2NF.xlsx**, saving any changes if prompted.

Results: After completing this exercise, you will have normalized the data set to second normal form.

Exercise 3: Normalizing to Third Normal Form

Scenario

Having normalized the data to second normal form, you will now normalize it to third normal form.

The main tasks for this exercise are as follows:

1. Review the Data Set in Second Normal Form
2. Normalize the Data Set to Third Normal Form

▶ Task 1: Review the Data Set in Second Normal Form

- In the **D:\Labfiles\Lab03\Starter** folder, open **Normalize to 3NF.xlsx**, and then on the **Normalize to 3NF** tab, review the table to identify any features of the data that might cause the table to violate third normal form.

▶ Task 2: Normalize the Data Set to Third Normal Form

1. Decompose the table in the **Normalize to 3NF** worksheet to create a new version of the data that is in third normal form. Document your solution in the free space in the **Normalize to 3NF** worksheet.
2. On the **3NF Solution** worksheet, review the suggested solution.
3. Close **Normalize to 3NF.xlsx**, saving any changes if prompted.

Results: After completing this exercise, you will have normalized the data set to third normal form.

Exercise 4: Denormalizing Data

Scenario

Having normalized the data set to third normal form, you will now denormalize a table. During your research, you identified the types of queries that will be issued against the database. These include a query that returns the total sales amount for each order. This query runs frequently, so you want to investigate ways to ensure that the response times for it are as fast as possible. To do this, you will denormalize the Orders table.

The main tasks for this exercise are as follows:

1. Review the Data Set in Third Normal Form

2. Denormalize the Orders Table

▶ Task 1: Review the Data Set in Third Normal Form

- In the **D:\Labfiles\Lab03\Starter** folder, open **Denormalize.xlsx**, and then on the **Denormalize** tab, review the tables.

▶ Task 2: Denormalize the Orders Table

1. Denormalize the **Orders** table to meet the requirements of the scenario for this exercise.
2. On the **Denormalized Solution** tab, review the suggested solution.
3. Close **Denormalize.xlsx**, saving any changes if prompted.

Results: After completing this exercise, you will have denormalized a table.

Question: In the suggested solution, the Orders table has a surrogate key as its primary key. Why do you think this key was chosen instead of a composite candidate key, such as the OrderDate and Customer columns?

Question: In the suggested solution, in the Customers table, are there any other columns that you could break down even further? If so, why might you do this?

Module Review and Takeaways

In this module, you have seen how to:

- Outline the benefits of normalization and some of the concepts that underpin it.
- Describe the various levels of normal form.
- Explain the benefits of denormalization and outline some techniques for denormalization.

Review Question(s)

Question: Can you think of a data set that you are familiar with, from your place of work or elsewhere, that you could normalize? What steps would you take to normalize to third normal form?

Module 4

Relationships

Contents:

Module Overview

In Module 2 of this course, you learned about entities and the relationships that can exist between them. You also saw how it is possible to model these relationships logically by using entity-relationship diagrams (ERDs). In Module 3, you learned about normalizing a design to obtain the optimal level of data redundancy. In this module, you will learn how to implement table relationships in a SQL Server® database.

Objectives

After completing this module, you will be able to:

- Explain how to implement various types of relationships in a SQL Server database.
- Describe the considerations for planning referential integrity in a SQL Server database.

Lesson 1
Introduction to Relationships

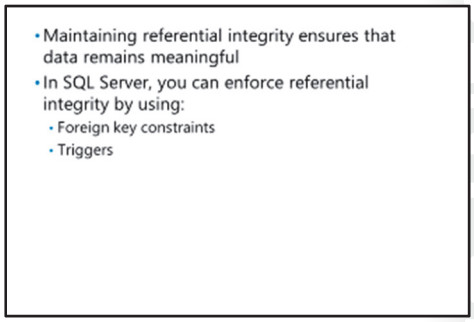
An ERD describes the entities in a database and the relationships between them. Schema mapping is the process of converting those logical designs into a physical implementation by creating tables, constraints, and other objects in a database. This lesson describes the various types of relationships that you can implement in a SQL Server database, and explains how to create them.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain the importance of referential integrity.
- Describe how to implement one-to-one relationships.
- Explain the considerations for NULL values in optional and required relationships.
- Describe how to implement one-to-many relationships.
- Describe how to implement many-to-many relationships.
- Describe the considerations for sub-types and super-types.
- Describe self-referencing relationships.

Referential Integrity



In an ERD, you specify the relationships that exist between the attributes in entities. In a relational database, you implement entities as tables, attributes as columns, and implement relationships by enforcing referential integrity.

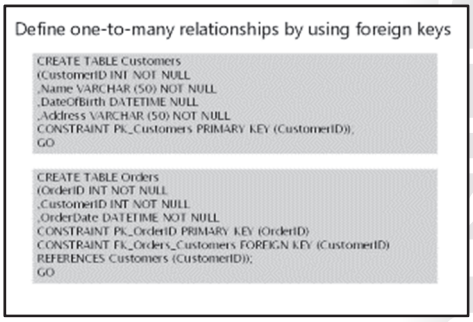
Referential integrity

Enforcing referential integrity between attributes involves ensuring that the values in one attribute have corresponding values in a second attribute. This ensures that the data remains meaningful. For example, in an Orders table in an OLTP database, there might be a column called CustomerID that stores a value that uniquely identifies the customer who placed any given order. However, in a normalized database, the Orders table would not contain detailed customer data; this data would be in a dedicated Customers table. The values in the CustomerID column in the Orders table must have corresponding values in the Customers table to ensure that the data makes sense. Any row in the Orders table with a CustomerID value, that does not exist in the Customers table, would represent an order without a matching customer. By maintaining referential integrity, you can prevent this kind of scenario from occurring.

Foreign keys and triggers

You can enforce referential integrity by using foreign key constraints and triggers. A trigger is an object in a database that contains a stored Transact-SQL statement. The statement in the trigger executes in response to other actions that occur in the database; for example, when a user updates data in a table. The code in the trigger ensures that any changes to the data in tables happen in a way that maintains referential integrity.

One-to-Many Relationships



Usually, the most common type of relationship in a relational database is the one-to-many relationship. In this kind of relationship, a value can appear only once in a column in the first table, but the same value can appear many times in a column in the second table. Typically, the two columns are in different tables, but this does not have to be the case.

One-to-many relationships are usually maintained by using primary keys and foreign keys. The primary key in the first table ensures that all values are unique and can therefore only appear once. The foreign key in the second table prevents the addition of any values to the column that do not have matching values in the primary key column of the first table.

The Customers and Orders table in the previous topic is an example of a one-to-many relationship. Each customer appears once in the Customers table but can appear multiple times in the Orders table. This reflects the business fact that one customer can place many orders, but each order can only be placed by one customer. Other examples of one-to-many relationships include:

- **The relationship between states and countries**. A country can contain many states, but a state can only be in one country.
- **The relationship between equipment and workers in an equipment inventory**. One worker can borrow multiple pieces of equipment, but each piece of equipment can only be loaned to one worker.

Note that sometimes the relationship is defined by real-world facts—as is the case with states and countries— and sometimes by the logic that the database embodies, as is the case with the equipment inventory example. In the latter, it is assumed that workers do not share equipment, so the database models this fact as a one-to-many relationship. If workers did share equipment, then this would not be a one-to-many relationship. Understanding the business processes that a database will model is an essential part of the design process.

Implementing a one-to-many relationship

You can implement one-to-many relationships by using a foreign key constraint. You can create a foreign key as part of the Transact-SQL table definition, or you can create it separately, after creating a table. The foreign key definition specifies the table that contains the primary key column that the foreign key references.

The following code example creates two tables, called Customers and Orders. The table definition for Customers includes a primary key on the CustomerID column. The table definition for Orders includes a primary key on the OrderID column and a foreign key on the CustomerID column, which references the CustomerID column in Customers.

Creating Tables and Constraints

```
CREATE TABLE Customers
(CustomerID INT NOT NULL
,Name VARCHAR (50) NOT NULL
,DateOfBirth DATETIME NULL
,Address VARCHAR (50) NOT NULL
CONSTRAINT PK_Customers PRIMARY KEY (CustomerID));
GO
CREATE TABLE Orders
(OrderID INT NOT NULL
,CustomerID INT NOT NULL
,OrderDate DATETIME NOT NULL
CONSTRAINT PK_OrderID PRIMARY KEY (OrderID)
CONSTRAINT FK_Orders_Customers FOREIGN KEY (CustomerID)
REFERENCES Customers (CustomerID));
GO
```

Considerations for creating foreign keys

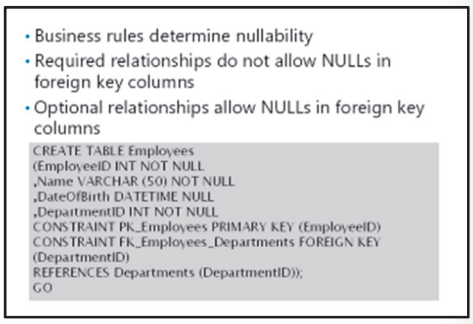
Consider the following points when planning foreign keys:

- You can only create foreign key relationships between columns that are in the same database.
- You can create a foreign key that references a different type of constraint called a unique constraint instead of a primary key. A table can only have one primary key, so if you need to impose uniqueness on an additional column, you can use a unique constraint to do this. However, the majority of foreign keys reference primary keys.

Optional and Required Relationships

When creating tables and constraints, it is important to consider NULL values. When you create a table, you can define the nullability status for each column. Nullability refers to whether or not the column is allowed to contain NULL values.

In the following code example, you can see that the table definition allows NULL values for the DateOfBirth column, but not for the other columns.



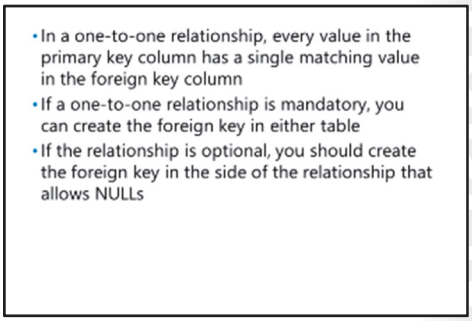
Create table statement that specifies nullability.

```
CREATE TABLE Employees
(EmployeeID INT NOT NULL
,Name VARCHAR (50) NOT NULL
,DateOfBirth DATETIME NULL
,DepartmentID INT NOT NULL
CONSTRAINT PK_Employees PRIMARY KEY (EmployeeID)
CONSTRAINT FK_Employees_Departments FOREIGN KEY (DepartmentID)
REFERENCES Departments (DepartmentID));
GO
```

You cannot create a primary key on a column that allows NULLs, but a foreign key column does not have this restriction. Whether or not you allow NULLs in a foreign key column depends on the business logic that defines the relationship. For example, in the Employees table in the previous code example, the foreign key columns DepartmentID is designated NOT NULL, meaning that the column cannot contain NULLs. The logic that determines this is that every employee must be associated with a department within a company—so every employee must have an associated, valid DepartmentID value. This is an example of a mandatory relationship, in which allowing NULLs would break the business rule.

However, the business rule could state that employees did *not* have to be associated with a specific department, perhaps because employees are associated with multiple departments, change departments frequently, or because the company does not have a clearly defined departmental structure. In this scenario, you could allow NULLs in the foreign key DepartmentID column. This would mean that, wherever possible, employees could be associated with a department, but this would also allow for employees whose departmental status was uncertain or unpredictable. This is an example of an optional relationship.

One-to-One Relationships

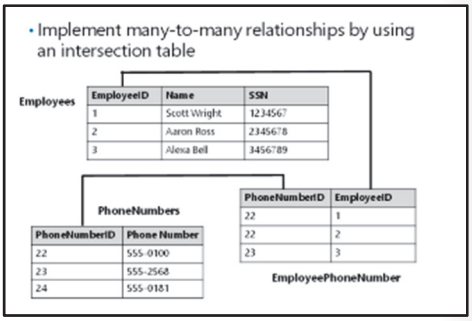


A one-to-one relationship is similar to a one-to-many relationship, the difference being that, in a one-to-one relationship, every value in the primary key column has a single matching value in the foreign key column. Generally, one-to-one relationships are not as common as one-to-many relationships. For example, imagine that a gym database stores information about its customers in a Customers table—this could include their height, age, weight, and other relevant physical characteristics. If the majority of queries against the Customers table return only the Name and Address columns, and not the Age, Weight, Height, and other columns, it might be more efficient to decompose the Customers table and store these columns in a separate CustomerDetails table. Decomposing the table in this way might improve response times because it would be possible to read the smaller Customers table from disk more quickly. In this example, both tables would probably use CustomerID as the primary key.

To maintain referential integrity, you would create a foreign key relationship between the CustomerID column in CustomerDetails and the CustomerID column in Customers. If the relationship is mandatory, the "direction" of the foreign key (from CustomerDetails to Customers or from Customers to CustomerDetails) does not really matter. You could create it in either table, referencing the other table in the foreign key definition. However, if the relationship is optional, and NULLs will be allowed on one side of the relationship, you should create the foreign key on the side of the relationship that allows NULLs.

Note: A foreign key does not guarantee uniqueness in the way that a primary key does so, in a one-to-one relationship, the foreign key column could contain duplicate values. To ensure uniqueness in the foreign key column, you can use a primary key or a unique constraint.

Many-to-Many Relationships



A many-to-many relationship exists when the data values in both sides of the relationship can be represented multiple times. This differs from a one-to-many relationship, in which the values on one side of the relationship must be unique. For example, imagine an Employees table that has been normalized so that the data about each employee's phone number and department are stored in separate tables, called PhoneNumbers and Departments. Employee includes the column EmployeeID, PhoneNumbers includes the column PhoneNumberID, and Departments includes the column DepartmentID. Each table contains additional columns that depend on these primary key columns. The business logic states that each employee can have multiple phone numbers, and that any given phone number might be used by multiple employees—so this is a many-to-many relationship. Similarly, business logic states that an employee can belong to more than one department, and of course, each department includes multiple employees.

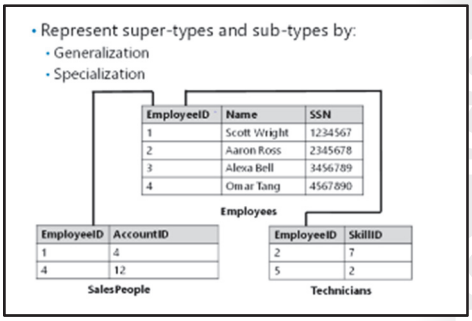
Representing a many-to-many relationship in a database is not as simple as representing a one-to-many relationship. To implement a many-to-many relationship between columns in two separate tables you usually create a third table, which is sometimes referred to as an intersection table, a link table, or a joining table. For example, to represent the many-to-many relationship between employees and their phone numbers described above, you could:

- Create a table called EmployeePhoneNumber with the columns EmployeeID and PhoneNumberID. These two columns serve as a composite primary key for the table.
- Create a foreign key relationship on the EmployeeID column in Employees that references EmployeeID in EmployeePhoneNumber.
- Create a foreign key relationship on the PhoneNumberID column in Employees that references PhoneNumberID in EmployeePhoneNumber.

The intersection table EmployeePhoneNumber can contain multiple values for the EmployeeID and PhoneNumberID columns, but the combination of these two values is unique for each row. You could create an intersection table called EmployeeDepartment in the same way to enable the many-to-many relationship between Employees and Departments.

Super-Types and Sub-Types

It is sometimes possible to identify entities in a database that are actually different types of the same thing. For example, imagine an ERD that contains the entities SalesPerson and Technician, which represent the types of job roles individuals hold within an organization. Both salespeople and technicians are sub-types of the category Employee. In this scenario, Employee is the super-type of SalesPerson and Technician. Sub-types have multiple attributes in common; in the Employee example, this might include Address, DateOfBirth, and SSN.

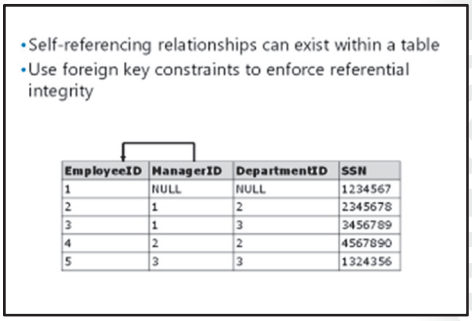


When implementing tables to represent sub-types, you have two options:

- **Generalization**. Generalization involves storing all of the attributes of the sub-types in a single table. For example, you could create a single Employees table that includes all of the attributes required to describe the sub-types SalesPerson and Technician. In this scenario, there would not be a dedicated table for each of the sub-types. Generalized tables can often contain a large number of NULL values because many attributes will not be relevant to all of the sub-types. For example, the attribute AccountID, which identifies the account that a salesperson manages, is not relevant to technicians. Similarly, the SkillID attribute, which identifies the technical skill of a technician, is not relevant to salespeople. Consequently, you would expect to see a lot of NULLs in these two columns. A table that has a large number of NULL values is sometimes called a sparse table.
- **Specialization**. Specialization involves creating a table to store the common attributes, and additionally a separate table for each sub-type to store the attributes that are uniquely relevant to that sub-type. For example, you could create a parent Employees table that contains the shared attributes, and a Technicians table and a SalesPerson table to store the attributes that are specific to those sub-types. All three tables could use EmployeeID as a primary key, and the sub-type tables would have foreign keys that reference the EmployeeID column in Employees in a one-to-one relationship.

Self-Referencing Relationships

A self-referencing relationship occurs when a one-to-many or one-to-one relationship exists within a single table. For example, imagine an Employees table that includes the columns EmployeeID and ManagerID. The ManagerID column indicates the employee ID of the manager for any given employee. The values in the ManagerID column are a subset of the values in the EmployeeID column. To maintain referential integrity between the columns, you must create a foreign key on the ManagerID column that references the EmployeeID column.



A self-referencing relationship might be optional or it might be required. For example, in the Employees table, a business rule might be that every employee should have a manager (including the managers themselves); this would be a required relationship because no NULLs would be allowed in the ManagerID column. However, if the business rule states that some managers do not need to have managers of their own (for example, the CEO), then it becomes an optional relationship because the ManagerID column must allow NULLs.

Check Your Knowledge

Question	
You are planning a database that tracks the courses that students attend at a college. Each course can include up to 25 students, and students can enroll on multiple courses at the same time. You have created an ERD that includes the entities Students and Courses. What type of relationship exists between these entities?	
Select the correct answer.	
	A one-to-many relationship.
	A one-to-one relationship.
	A many-to-many relationship.
	A super-type/sub-type relationship.

Lesson 2
Planning Referential Integrity

Foreign key constraints limit the values that you can add to a column, and they also prevent the update or deletion of values in the referenced column, when those values also exist in the foreign key column. While this is the default behavior for foreign key constraints, there are some scenarios in which you might want to alter this behavior—for example, by propagating updates or deletes from the referenced column to the foreign key column. This lesson describes the options for implementing referential integrity, and explains the considerations for implementing cascading referential integrity.

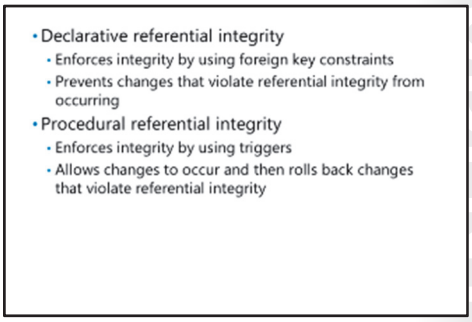
Lesson Objectives

After completing this lesson, you will be able to:

- Describe the options for implementing referential integrity.
- Describe the options for cascading changes to data.

Options for Implementing Referential Integrity

Broadly speaking, there are two approaches to implementing referential integrity—these are declarative referential integrity and procedural referential integrity.



Declarative referential integrity

Declarative referential integrity involves using constraints to maintain integrity, as the previous lesson in this module described. Constraints are usually the preferred method of maintaining integrity because they are efficient in the way that they operate. For example, when a Transact-SQL statement attempts to insert a new row into a table, any foreign key constraints on that table will check the new data values in the statement against the referenced columns, to ensure that they are valid *before* inserting the new row. Only if the data values pass this check is the new row inserted. The same approach applies to updates to the rows; the updated value must match a value in the referenced primary key column. Adding a foreign key constraint also prevents updates and deletes in the referenced primary key column that would render data in the foreign key invalid. For example, the deletion of the value "23" from a primary key column that is referenced by a foreign key would not be allowed if the foreign key column contained the value "23"—this is because it would break referential integrity between the two columns.

Procedural referential integrity

Procedural referential integrity involves using triggers to enforce integrity rules. A trigger is a database object that contains a Transact-SQL statement. A trigger is associated with a particular table, and it occurs in response to updates, inserts, or deletes on that table. The Transact-SQL statement in a trigger can perform actions to ensure that referential integrity is maintained in response to changes in the data. SQL Server supports data manipulation language (DML) triggers. DML triggers include AFTER triggers and INSTEAD OF triggers.

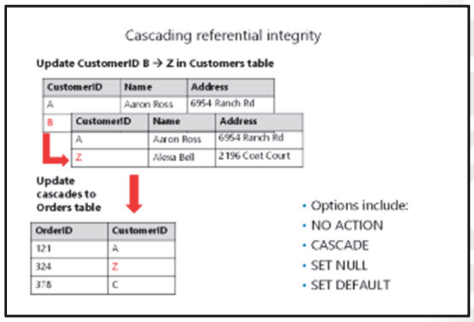
- **AFTER triggers**. Typically, you use AFTER triggers to maintain referential integrity, but only when a foreign key constraint is not suitable. For example, you might use an AFTER trigger if it is necessary to maintain integrity across multiple databases because foreign key constraints can only enforce integrity within a database. As the name suggests, an AFTER trigger occurs after an operation to insert, update, or delete data completes. If you used an AFTER trigger to duplicate the behavior of a foreign key constraint and only allow valid data values in a particular column, the trigger would achieve this by first allowing the change to occur—and only then checking the data values and reversing or rolling back the data change if required. This approach is less efficient than just preventing invalid changes in the first place, as a foreign key constraint does.
- **INSTEAD OF triggers**. INSTEAD OF triggers operate by executing the Transact-SQL statement stored in the trigger instead of performing the triggering action, such as an update. For example, if you want to prevent a column from being updated at all, you could use an INSTEAD OF trigger to return a message explaining that updates are not allowed. The trigger would occur in response to an update, the update would not occur, and the message would be returned instead. INSTEAD OF triggers can be useful for enforcing data rules that constraints cannot enforce. For example, you could use a CHECK constraint to enforce a limit of the quantity of items allowed in any one order. The constraint would check the number of items, and allow the transaction to go ahead if the number of items is below the threshold, and prevent it if not. However, if this kind of checking operation requires referencing data in a different table, a CHECK constraint is not suitable, so you could use a trigger instead. You can use the Transact-SQL CREATE TRIGGER statement to create triggers.

For more information about creating triggers, see:

DML Triggers

http://aka.ms/jv2m72

Cascading Referential Integrity



Updates to existing data can potentially cause referential integrity to be broken. For example, imagine a Customers table and an Orders table, both of which include a column called CustomerID; the CustomerID column in Orders has a foreign key constraint that references the CustomerID column in Customers, and this constraint ensures that all orders are associated with a valid CustomerID value. If you updated a CustomerID value in Customers, then the corresponding CustomerID values in Orders would be invalid and the data set would be compromised—because it would contain orders with no associated customer. A foreign key prevents this behavior by default. However, sometimes, instead of preventing changes to data in the primary key, you might want to allow these changes and propagate them, so that all related columns are updated accordingly. This is called cascading referential integrity.

When you create a foreign key constraint, you can define how to handle cascading referential integrity. You can define a separate action for updates and for deletes. The options include:

- **NO ACTION**. If you configure NO ACTION, then the change is not allowed, and the user receives an error message. This is the default setting, which applies if you do not specify any other action.
- **CASCADE**. The CASCADE option allows the change to occur in the primary key column and also causes the change to propagate to the column in the foreign key table. For example, if you update a CustomerID value in Customers, all corresponding instances of the value in CustomerID in Orders will also be updated to the new value. Whilst this might initially seem like the ideal solution, you should be very cautious about implementing this option because it can have serious consequences, such as:
  - If you configure the foreign key to cascade deletes from the primary key table, the deletion of a row in the primary key table will cause the deletion of all corresponding rows in the foreign key table. For example, deleting a customer from the Customers table would also delete all of that customer's orders.
  - If you cascade updates, the tables in the database will maintain referential integrity. However, it is often important to maintain integrity across multiple databases; for example, you might periodically load order data that is older than one month into a data warehouse. The data warehouse might contain orders for customer A. If you update customer A's CustomerID record in Customers in the OLTP database to customer Z, the older records in the data warehouse will not be updated. When you next load orders into the data warehouse, the records for Customer A and Customer Z will not be recognized as the same customer. You can use triggers to maintain integrity between different databases.
  - If cascading is configured across multiple tables, it is possible for a single delete action to have far-reaching consequences. It is important, therefore, to plan carefully to decide whether cascading is appropriate behavior in the specific context of any given database.
- **SET NULL**. The SET NULL option sets the value in the referencing foreign key column to NULL if the primary key value is deleted or updated. This can lead to rows in the foreign key column that are sometimes called orphans, because they have no corresponding value in the parent primary key table.
- **SET DEFAULT**. The SET DEFAULT option works in a similar way to the SET NULL option, except that instead of setting values to NULL, it sets them to a default value, such as "unknown" or "parent deleted".

 **Note:** The ability to cascade changes to data across related tables and columns is extremely powerful, but you should always plan very carefully before implementing it. Without proper planning and implementation, cascading changes can have serious consequences and can potentially result in the unintended deletion of data.

Demonstration: Implementing Referential Integrity by Using a Foreign Key

In this demonstration, you will see how to:

- Implement foreign key constraints.
- Implement cascading referential integrity.

Demonstration Steps

1. Start the 10985C-MIA-DC and 10985C-MIA-SQL virtual machines, and then log on to 10985C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. In the **D:\Demofiles\Mod04** folder, run **Setup.cmd** as Administrator.
3. In the **User Account Control** dialog box, click **Yes**.
4. Open **Microsoft SQL Server Management Studio** and then connect to the **MIA-SQL** instance of the database engine by using Windows Authentication.
5. On the **File** menu, point to **Open**, click **File**, browse to **D:\Demofiles\Mod04**, click **Referential Integrity.sql**, and then click **Open**.
6. In the query window, under the comment **Create a database and change database context**, select the Transact-SQL statement and then click **Execute**.
7. In the query window, under the comment **Create Customers table with a primary key constraint and four rows**, review the Transact-SQL statement, select the Transact-SQL statement, and then click **Execute**.
8. In the query window, under the comment **Create a table with a foreign key constraint**, review the Transact-SQL statement, select the Transact-SQL statement, and then click **Execute**.
9. In the query window, under the comment **Test foreign key constraint by adding an order with a valid CustomerID value**, review the Transact-SQL statement, select the Transact-SQL statement, and then click **Execute**.
10. Note that the statement completes successfully.
11. In the query window, under the comment **Add an order with an invalid CustomerID value**, review the Transact-SQL statement, select the Transact-SQL statement, and then click **Execute**.
12. In the Results pane, review the message that states that the insert statement conflicted with the foreign key constraint.
13. In the query window, under the comment **Update a CustomerID value in Customers that has no matching value in Orders**, review the Transact-SQL statement, select the Transact-SQL statement, and then click **Execute**.
14. Note that the statement completes successfully.
15. In the query window, under the comment **Update a CustomerID value in Customers that has a matching value in Orders**, review the Transact-SQL statement, select the Transact-SQL statement, and then click **Execute**.
16. In the Results pane, review the message that states that the update statement conflicted with the constraint.
17. In the query window, under the comment **Drop foreign key and add new foreign key that specifies cascading referential integrity**, review the Transact-SQL statement, select the Transact-SQL statement, and then click **Execute**.

18. In the query window, under the comment **Test cascading referential integrity**, review the Transact-SQL statement, select the Transact-SQL statement, and then click **Execute**.
19. In the Results pane, review the results, noting that the row in the **Orders** table was successfully updated.
20. Close SQL Server Management Studio without saving changes.

Check Your Knowledge

Question	
A junior administrator reported that, when she deleted a row in a table in your organization's OLTP database, the action unexpectedly caused the deletion of more than 100 rows. You successfully verify that the necessary foreign key constraints exist. What was the most likely cause of the unexpected deletions?	
Select the correct answer.	
	A foreign key constraint was configured with the ON DELETE SET NULL option.
	A foreign key constraint was configured with the ON DELETE SET DEFAULT option.
	A foreign key constraint was configured with the ON DELETE CASCADE option.
	A foreign key constraint was configured with the ON UPDATE CASCADE option.

Lab: Planning and Implementing Referential Integrity

Scenario

You work for an organization called Adventure Works, which is a retailer of bicycles and associated products. You are planning referential integrity for a section of a database called OrdersDatabase, an OLTP database that tracks customers, the orders that they place, and the products that are included in the orders. To implement referential integrity, you will use foreign keys. First, you will plan the foreign keys that you will need to enforce integrity; you will then implement the foreign keys; and finally, you will implement cascading referential integrity.

Objectives

After completing this lab, you will have:

- Planned referential integrity for an OLTP database.
- Implemented referential integrity by using foreign keys.
- Implemented cascading referential integrity.

Estimated Time: 60 minutes

Ensure that the 10985C-MIA-DC and 10985C-MIA-SQL virtual machines are both running, and then log on to 10985C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.

Exercise 1: Planning Referential Integrity

Scenario

You have established that the following business rules must be enforced:

- Every order must be associated with a specific customer.
- Every line item must be associated with a product and a valid order.
- Customer details must be associated with a valid customer.

In this exercise, you will use a database diagram to help you to plan which foreign key constraints you will need to enforce these rules.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment

2. Identify Relationships

3. Plan Foreign Keys

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 10985C-MIA-DC and 10985C-MIA-SQL virtual machines are both running, and then log on to 10985C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. Browse to the **D:\Labfiles\Lab04\Starter folder**, and then run **Setup.cmd** as Administrator.

▶ Task 2: Identify Relationships

1. Open SQL Server Management Studio and connect to the MIA-SQL Database Engine instance by using Windows Authentication.

2. In the **CustomerOrders** database, create a database diagram, and then add the following tables to it:
   - **CustomerDetails**
   - **Customers**
   - **Orders**
   - **Products**
   - **LineItems**
3. Review the database diagram and determine the relationships between the tables and what types of relationships they are (one-to-one, one-to-many, or many-to-many).

▶ Task 3: Plan Foreign Keys

1. Use the database diagram and the business rules to decide which foreign key constraints you need to create to enforce referential integrity. Include in your considerations which columns should have a foreign key, and which column each foreign key should reference.
2. After planning the foreign keys, save the database diagram as **Keys**.
3. Close the database diagram pane.

Results: After completing this exercise, you will have identified the keys required to enforce referential integrity rules.

Exercise 2: Implementing Referential Integrity by Using Constraints

Scenario

The following relationships exist in the database:

- A one-to-one relationship between Customers and CustomerDetails.
- A one-to-many relationship between Customers and Orders.
- A one-to-many relationship between Orders and LineItems.
- A one-to-many relationship between Products and LineItems.

To implement the required referential integrity, you need to create the following foreign key constraints:

- A foreign key on OrderID in LineItems that references OrderID in Orders.
- A foreign key on ProductID in LineItems that references ProductID in Products.
- A foreign key on CustomerID in Orders that references CustomerID in Customers.
- A foreign key on CustomerID in CustomerDetails that references CustomerID in Customers.

In this exercise, you will implement these foreign key constraints.

The main tasks for this exercise are as follows:

1. Implement a Foreign Key in the Orders Table
2. Implement a Foreign Key in the CustomerDetails Table
3. Implement Foreign Keys in the LineItems Table

▶ Task 1: Implement a Foreign Key in the Orders Table

1. In a query window in SQL Server Management Studio, write a Transact-SQL ALTER TABLE statement to create a foreign key constraint called **FK_Orders_Customers** on the **CustomerID** column in the **Orders** table that references the **CustomerID** column in the **Customers** table. Do not include an ON UPDATE or ON DELETE clause.
2. To test the foreign key, write a Transact-SQL statement INSERT statement to add a row to the **Orders** table using the following values:
   - **OrderID** = **105**
   - **CustomerID** = **2**
   - **OrderDate** = **GETDATE()**
3. This INSERT statement should succeed.
4. To test the foreign key again, write a Transact-SQL statement INSERT statement to add a row to the **Orders** table using the following values:
   - **OrderID** = **106**
   - **CustomerID** = **5**
   - **OrderDate** = **GETDATE()**
5. This INSERT statement should be prevented by the foreign key constraint because the **CustomerID** value **5** does not exist in the **Customers** table.

▶ Task 2: Implement a Foreign Key in the CustomerDetails Table

1. In the same query window, write a Transact-SQL ALTER TABLE statement to create a foreign key constraint called **FK_CustomerDetails_Customers** on the **CustomerID** column in the **CustomerDetails** table that references the **CustomerID** column in the **Customers** table. Do not include an ON UPDATE or ON DELETE clause.
2. To test the foreign key, write a Transact-SQL statement INSERT statement to add a row to the **CustomerDetails** table using the following values:
   - **CustomerID** = **5**
   - **Address** = **'9832 Mt. Dias Blv.'**
   - **City** = **'Chicago'**
   - **Postal Code = '97321'**
   - **DateOfBirth** = **'08/09/1970'**
3. This INSERT statement should be prevented by the foreign key constraint because the **CustomerID** value **5** does not exist in the **Customers** table.

▶ Task 3: Implement Foreign Keys in the LineItems Table

1. In the same query window in SQL Server Management Studio, write a Transact-SQL ALTER TABLE statement to create a foreign key constraint called **FK_LineItems_Orders** on the **OrderID** column in the **LineItems** table that references the **OrderID** column in the **Orders** table. Do not include an ON UPDATE or ON DELETE clause.

2. To test the foreign key, write a Transact-SQL statement INSERT statement to add a row to the **LineItems** table using the following values:
   - **OrderID** = **101**
   - **ProductID** = **33**
   - **UnitPrice** = **30.00**
   - **Quantity** = **1**
3. This INSERT statement should succeed.
4. To test the foreign key again, write a Transact-SQL statement INSERT statement to add a row to the **LineItems** table using the following values:
   - **OrderID = 106**
   - **ProductID = 44**
   - **UnitPrice = 30.00**
   - **Quantity = 1**
5. This INSERT statement should be prevented by the foreign key constraint because the **OrderID** value **106** does not exist in the **Orders** table.
6. In the same query window in SQL Server Management Studio, write a Transact-SQL ALTER TABLE statement to create a foreign key constraint called **FK_LineItems_Products** on the **ProductID** column in the **LineItems** table that references the **ProductID** column in the **Products** table. Do not include an ON UPDATE or ON DELETE clause.
7. To test the foreign key, write a Transact-SQL statement INSERT statement to add a row to the **LineItems** table using the following values:
   - **OrderID** = **102**
   - **ProductID** = **22**
   - **UnitPrice** = **15.00**
   - **Quantity** = **1**
8. This INSERT statement should succeed.
9. To test the foreign key again, write a Transact-SQL statement INSERT statement to add a row to the **LineItems** table using the following values:
   - **OrderID = 104**
   - **ProductID = 66**
   - **UnitPrice = 30.00**
   - **Quantity = 1**
10. This INSERT statement should be prevented by the foreign key constraint because the **ProductID** value **66** does not exist in the **Products** table.
11. Close the query window and save the file as **CreateForeignKeys.sql** in the **D:\Labfiles\Lab04\Starter** folder.
12. Open the **Keys** database diagram and review it, noting that it includes the foreign key relationships that you created in this exercise.

Results: After completing this exercise, you will have implemented referential integrity in the OrdersDatabase database using constraints.

Exercise 3: Implementing Cascading Referential Integrity

Scenario

An additional business rule states that whenever a CustomerID is removed from the Customers table, which sometimes happens when a customer requests that their data is removed from the database, all associated information should also be removed, including customer details. However, the Orders table references the CustomerID column in the Customers table, and you do not want to cascade deletions from Customers to Orders because the order information is important for accounting and reporting purposes. Instead, you must ensure that when a row is deleted in Customers, the CustomerID value in Orders updates to a default value. In this exercise, you will configure referential integrity to implement these rules.

The main tasks for this exercise are as follows:

1. Configure Cascading Referential Integrity in the CustomerDetails Table
2. Configure SET DEFAULT in the Orders Table

▶ Task 1: Configure Cascading Referential Integrity in the CustomerDetails Table

1. In SQL Server Management Studio, in a new query window, type and execute a Transact-SQL statement to delete the row from the **Customers** table that has the **CustomerID** value **2**. The delete should fail because of a foreign key constraint.
2. Type and execute an ALTER TABLE Transact-SQL statement to drop the **FK_CustomerDetails_Customers** constraint in the **CustomerDetails** table.
3. Type and execute an ALTER TABLE Transact-SQL statement to create a new foreign key constraint called **FK_CustomerDetails_Customers** on the **CustomerDetails** table. Include a clause to cascade delete actions.
4. Attempt to delete the row from the **Customers** table that has the **CustomerID** value **2** again. The delete should fail again because of the foreign key constraint in the **Orders** table.

▶ Task 2: Configure SET DEFAULT in the Orders Table

1. In the same Transact-SQL query window, type and execute an ALTER TABLE Transact-SQL statement that creates a default constraint on the **CustomerID** column in **Orders**, with a value of **0**.
2. Type and execute a Transact-SQL statement that adds a row to the **Customers** table using the following values:
   - **CustomerID** = **0**
   - **FirstName** = **'Not Applicable'**
   - **LastName** = **'Not Applicable'**
3. Type and execute an ALTER TABLE Transact-SQL statement to drop the **FK_Orders_Customers** constraint in the **Orders** table.
4. Type and execute an ALTER TABLE Transact-SQL statement to create a new foreign key constraint called **FK_Orders_Customers** on the **Orders** table. Include a clause to set the value to default for delete actions.
5. Attempt to delete the row from the **Customers** table that has the **CustomerID** value **2** again. The delete should succeed.
6. Check that the Order 101 now has a **CustomerID** value of 0.
7. Save the query file as **ImplementCascadingIntegrity.sql** in the **D:\Labfiles\Lab04\Starter** folder.
8. Close SQL Server Management Studio.

Results: After completing this exercise, you will have implemented cascading referential integrity.

Question: Do you think that it was a good idea to implement the ON DELETE CASCADE and the ON DELETE SET DEFAULT options in the final exercise in the lab? What problems might this potentially cause? What might you have done instead to prevent these problems?

Module Review and Takeaways

In this module, you have seen how to:

- Explain the implementation of various types of relationships in a SQL Server database.
- Describe the considerations for planning referential integrity in a SQL Server database.

Review Question(s)

Question: Think about the kinds of processes that exist in your organization. What entities can you identify and what types of relationships exist between them? Think of three or four specific entities—what constraints would you implement to enforce referential integrity, and would you use options such as CASCADE or SET NULL?

Module 5

Performance

Contents:

Module Overview

This module describes the effects of database design on performance.

Objectives

At the end of this module, you will have gained an appreciation of the performance effects of database design and be able to discuss the implications of:

- Indexing
- Query Performance
- Concurrency

Performance improvements come from:

- Logical design
- Physical design
- Index design
- Query design
- Execution

Some of these items, such as Logical and Physical design, are beyond the scope of this lesson. You will be concentrating on the last three items.

Lesson 1
Indexing

Discuss the performance effects of indexing.

For report and query writers, having a basic understanding of indexes is useful in identifying poorly performing queries. Being able to identify problems with indexes, such as spotting a table scan when you expected an index to be used, can be helpful in tuning an application. Similarly, SQL Server® might opt to scan an entire table when you expect the use of a nonclustered index. Understanding that nonclustered indexes involve an additional lookup operation versus scanning the table in a single pass, might help you understand the choices of the optimizer.

For more information, see *How Indexes are Used by the Query Optimizer* in the topic *Clustered and Nonclustered Indexes Described*, in Microsoft Docs:

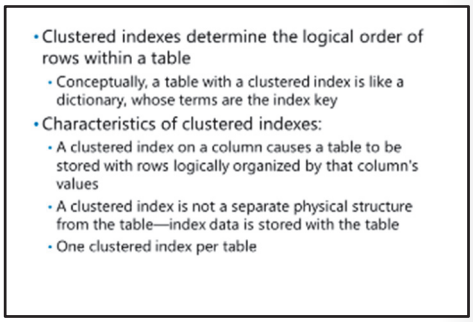
Clustered and Nonclustered Indexes Described

http://aka.ms/N5ef6y

Clustered Index

In SQL Server, a table may not have any indexes, or it may have one or more. Indexes are not required for data access, although there are some features, such as constraints, that create indexes to support them and cannot be removed without dropping the constraint.

If a table has no clustered indexes, it is said to be a heap. In a heap, there is no order to the way the table's rows are organized.



Note: For more information on table structures and objects such as data and index pages, see Microsoft course *20762: Developing SQL Databases*.

A database developer or administrator may choose to add a clustered index to a table. A clustered index causes the rows in the table to be logically stored in order of the column(s) specified in the index, called the index key. The columns used as the index key in a clustered index are called the clustering key. Tables with clustered indexes are maintained in index order, and rows are inserted into the correct logical location determined by their index key value. Rows are stored with their index key value. Clustered indexes are not stored as separate structures in the database.

Because the clustered index determines the order of the rows in the table, only one clustered index is permitted per table.

When SQL Server searches for and locates an index key value in the clustered index, it also locates data for that row. No additional navigation is required, except in the case of special data types. Conceptually, a table with a clustered index is like a dictionary, whose terms are the index key. The terms appear in the dictionary in alphabetical order. When you search the dictionary for a term and locate it, you are also at the definition for the term.

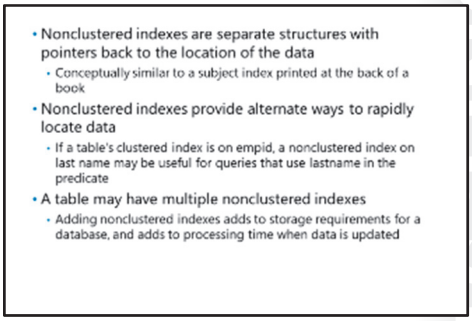
Because there is only one clustered index permitted per table, care should be taken when choosing the column(s) used as the clustering key. For more information, see *Create Clustered Indexes* in Microsoft Docs:

Create Clustered Indexes

http://aka.ms/Jw5jlv

Nonclustered Index

In addition to clustered indexes, SQL Server supports another common type, called a nonclustered index. Nonclustered indexes are separate structures that contain one or more columns as a key, in addition to a pointer back to the source row in the table. Nonclustered indexes may be created on tables that are heaps, or on tables organized by clustered indexes. The pointer information stored in the nonclustered index depends on whether the underlying table is organized as a heap or by a clustered index, but is otherwise transparent to a query.



Because the nonclustered index is a separate structure and stores only key data and a pointer, queries that use a nonclustered index may require an additional lookup operation before accessing the underlying data. Conceptually, a nonclustered index is like a subject index printed at the back of a book. A sorted list of items appears, along with a page number that points to the location of the subject in the main section of the book. When a key value is located in a nonclustered index, SQL Server may use the pointer to locate the data in the table itself.

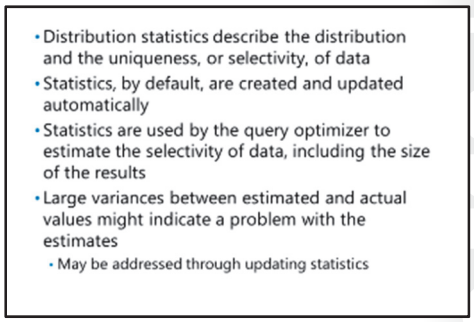
Nonclustered indexes are often added to tables to improve specific query performance and avoid resource-intensive table scans. However, nonclustered indexes require additional disk space for storage, and are updated in real time as the underlying data changes, adding to the duration of transactions. For these reasons, database administrators may decide not to add nonclustered indexes on every column referenced by a query.

For more information, see *Create Nonclustered Indexes* in Microsoft Docs:

Create Nonclustered Indexes

http://aka.ms/F8njtm

Distribution Statistics



SQL Server creates and maintains statistics on the distribution of values in columns, in tables. Distribution statistics are used by the query optimizer to estimate the number of rows involved in a query (selectivity). The optimizer uses this information to help determine the optimal access method for the query. For example, statistics about the number of rows in a table might cause SQL Server to seek through an index rather than scan the entire table.

By default, distribution statistics are automatically created and maintained by SQL Server for:

- The leading key column in all indexes.
- Any column used as a predicate as part of a WHERE clause or JOIN ON clause.
- Columns used in temp tables (although not in table variables).

Administrators and database developers may also manually create statistics and update them manually, or by way of scheduled jobs.

Note: Manually creating, updating, and reviewing statistics is beyond the scope of this course. For more information, see Microsoft course *20762: Developing SQL Databases*.

Demonstration: Testing Index Performance

In this demonstration, you will see how to evaluate the performance impact of indexes.

Demonstration Steps

1. In the virtual machine, on the taskbar, click **Microsoft SQL Server Management Studio**.
2. In the **Connect to Server** dialog box, in the **Server name** box, type **MIA-SQL**. In the **Authentication** box, ensure **Windows Authentication** is selected, and then click **Connect**.
3. On the **File** menu, point to **Open**, click **File**, navigate to the **D:\Demofiles\Mod05\Demo1.sql** script file, and then click **Open**.
4. To turn statistics on, highlight the code under **Step 1** and click **Execute**.
5. To run a query with no indexes, highlight the code under **Step 2** and click **Execute**.
6. To create indexes, highlight the code under **Step 3** and click **Execute**.
7. To run a query with indexes, highlight the code under **Step 4** and click **Execute**.
8. To drop the indexes, highlight the code under **Step 5** and click **Execute**.
9. Close SQL Server Management Studio without saving any changes.

Question: How do you choose between index types?

Lesson 2
Query Performance

Describe performance effects of join and search types.

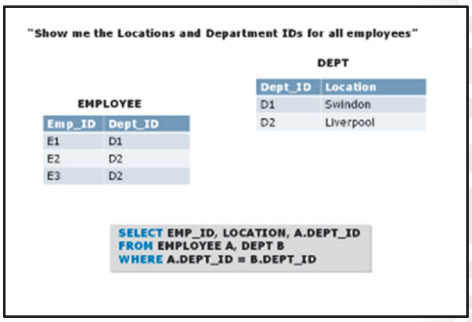
Lesson Objectives

At the end of this lesson, you will have gained an appreciation of the performance effects of the following join and search types:

- Joins
- Hash
- Merge
- Loop
- Searches
- Search Arguments
- Non-Search Arguments

Joins

Here is a quick overview of JOINs. There are five main types of JOINs: INNER, LEFT OUTER, RIGHT OUTER, FULL and CROSS. Note that LEFT and RIGHT OUTER are normally truncated to LEFT or RIGHT. An OUTER JOIN defaults to LEFT. The purpose of a JOIN is to gather information from one or more tables and return it from a single query.



In the following slide, the JOIN poses no significant performance problem, because the first table has only two rows, while the second table has three rows.

However, what would occur if the first table had 100 rows while the second table had 100,000 rows? This would result in 100 by 100,000—or 10 million potential comparisons—before all the related rows are joined.

There are JOIN techniques that can be used in an attempt to improve performance.

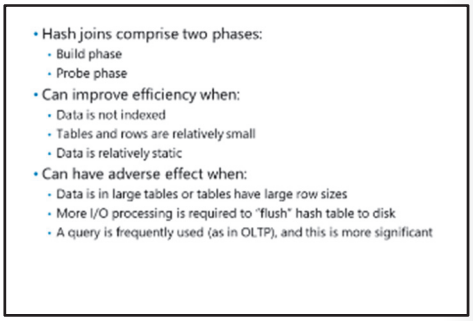
Hash Joins

The first join technique is to make use of hash joins.

Hash joins are built using hash tables, which are assembled in-memory by SQL Server. Using a hash table consists of two phases.

Build phase

In this phase, the smaller of the two tables is read and the keys of the tables together with the predicate on which the JOIN is based (the equi-join predicate, for example, ... ON a.id = b.id) are hashed using a proprietary hash function, then put into a hash table.



Probe phase

In this phase, each hash is read and compared against the computed hashes of the rows in the second table, with the output results segregated until the second table has been read in full. Finally, the output results are retrieved and presented as the query results.

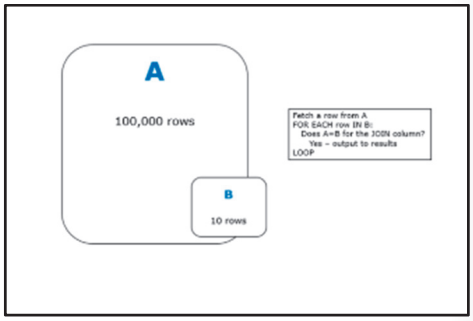
Efficiency considerations when using hash joins

Hash joins are efficient when comparing data that isn't indexed or when comparing a small data set against a larger one. This is simply because the algorithms used during the second phase will disregard an index if it exists, preferring instead to calculate the hash values and return the results. While the algorithms to do this are efficient, it's worth noting that hash joins need an initial memory allocation and require that the build phase is completed before the probe phase starts and results can begin to be returned.

Consequently, for very large tables and for tables with large row sizes, the hash tables may have to be flushed to disk—this can incur penalties for I/O, particularly when using non-SSD disk drives. This constraint also naturally limits the number of hash joins that can be stored in memory and executed at any one time. Consequently, forcing a hash join in a frequently-used stored procedure in a typical OLTP can hinder, rather than help, query optimization.

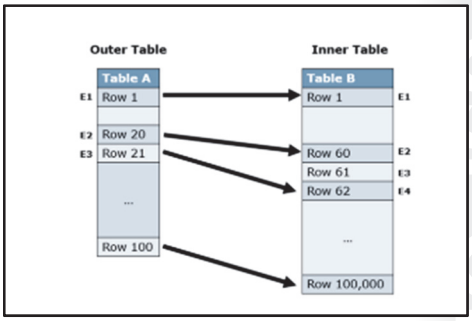
Loop Joins

A loop join operates differently from a hash join, in that it is best suited to a join on a small table to a larger one. The join is constructed in a nested fashion—that is, the smaller table is "placed inside" the larger one and each row from the outer table is compared against each row in the smaller table, until all matches have been found.



Consequently, these types of join are very efficient when using a small table very frequently to sort data; however, they can suffer in proportion to the size of the smaller table. Inner loop joins also come in a variety of types—a naive inner loop join, which is a comparison of every row in A to B; a temporary index inner loop join, where an index is created (b-tree) during joining, which is discarded after use; and an index inner loop join, where a permanent index is used to optimize the loop join. Consequently, good indexes on the tables can influence the decision of the query optimizer to select the loop join, if it will complete more efficiently than a hash join.

Merge Joins



A merge join is ideally suited to presorted tables of similar size. The query optimizer will use a merging algorithm to compare the rows in a similar way to the "merge sort" computer science principle. Each sorted row in A is compared to the equivalent sorted row in B. If duplicate rows exist on the left or right, the set (Cartesian product) of rows is compared before moving on; however, in most cases, the merge operation is a straightforward comparison rather than a many-to-many operation. This is an efficient way of joining with large data sets, especially those that are already presorted (such as clustered indexes) using the join predicate.

The scans are performed in parallel. When a match is found, the next search continues where the last one left off.

In the slide, Employee E2 in row 20 of the outer table has a match in row 60 of the sorted inner table. The next comparison will be between row 21 of the outer table and row 61 of the inner table. A match will be found in row 62 of the inner table. The next scan will be between row 22 of the outer table and row 63 of the inner table, and so on, until the scan between row 100 of the outer table and row 100,000 of the inner table.

Although this join is faster, the rows in the table must be sorted if no index exists. This incurs a performance penalty.

Imagine that the outer table contains 100 rows and the inner table contains 100,000 rows. This results in only 100,000 comparisons (the number of rows in the inner table).

Real-world Issues and Scenarios

Summary

Here is a summary of when to use the various types of join:

HASH JOIN

Tables are either fairly evenly-sized or large.

Indexes are practically irrelevant unless they are filtering on additional WHERE clauses; good for heaps.

Arguably the most versatile form of join.

LOOP JOIN

The query has a small table on the left side of the join.

One or both tables are indexed on the JOIN predicate.

MERGE JOIN

Tables are fairly even in size.

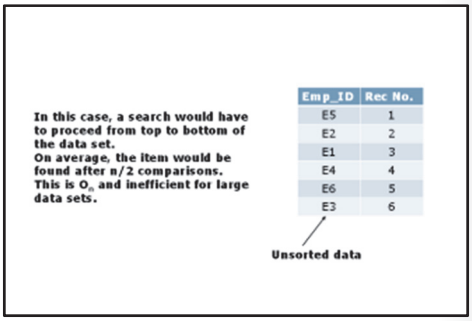
This works best when tables are well-indexed or presorted.

Uses a very efficient sort algorithm for fast results.

Unlike hash join, there is no memory reallocation; good for parallel execution.

If in doubt, let the optimizer decide.

Searches



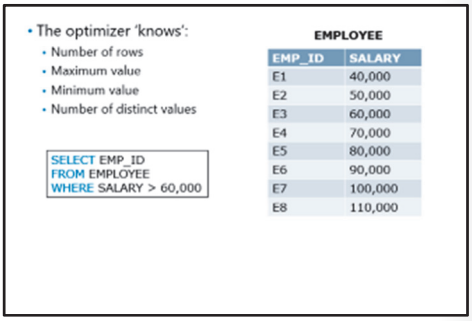
Without any data structures, such as indexes, to aid searching, the best option is to start at the beginning of the data and stop when you find what you are looking for. To give you some idea about the issues this causes, consider the search algorithm necessary to find a telephone number in a directory, where the entries are in a random order.

On average, you will find what you are looking for after n/2 comparisons, where n is the number of entries. This is called "Order n" but does not scale well, because the time taken to search increases as n increases (the factor of two is not important for this discussion).

When a RDBMS accesses a table that has no index associated to it, the RDBMS must resort to a sequential search (or relational scan) of each record until the desired record is found. This poses no great problem if the table is small (10 to 200 rows); the optimizer usually loads the entire table into memory and performs the scan there. With this scenario, performance is generally not a problem.

If the table is large, (about 100,000 to 1,000,000 rows), sequential access will definitely cause performance to degenerate. In this case, you would like to utilize a suitable index, so as to avoid the performance penalty.

Search Arguments



A typical SQL query can be divided into six distinct components or clauses:

The SELECT clause, the FROM clause, the WHERE predicate, the GROUP BY clause, the HAVING predicate and the the ORDER BY clause.

The WHERE predicate is of particular interest. A predicate is an expression which can be TRUE, FALSE or UNKNOWN. You can apply the operator to values to produce a truth value (that is, TRUE, FALSE or UNKNOWN). How predicates are used affects performance. Each of the predicates falls into one of the following categories:

- Sargable
- Nonsargable

A sargable predicate is one that can use created indexes for faster searches and faster execution of a query.

Nonsargable predicates cannot take advantage of search arguments.

In this example, the query reads:

SELECT EMP_ID

FROM EMPLOYEE

WHERE SALARY > 60,000

The optimizer uses the high and low key values from the statistics in the database catalogues and uses them in the following filter formula (FF):

FF = (VALUE - LOWKEY)/(HIGHKEY - LOWKEY)

= (60,000 – 40,000)/(110,000 – 40,000)

= (20,000/70,000) = 0.286 (approximately)

This tells the database to start looking just under a third of the way down the column.

Non-Search Arguments

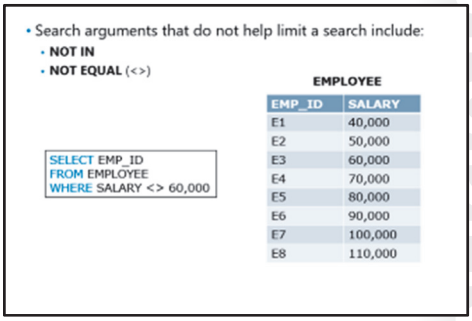
Non-search arguments do not help limit a search. These predicates include:

- NOT IN
- NOT EQUAL

These predicates may require that a sequential table scan be performed to obtain the necessary rows. If indexes cover the query, the sequential table scan will not be required.

Another example where indexes may be of no value is where an attribute is used in an expression, such as:

SALARY * 1.5 >= 100,000



Demonstration: Nonsargable Queries

In this demonstration, you will see the effects of a nonsargable query.

Demonstration Steps

1. On the taskbar, click **Microsoft SQL Server Management Studio**.
2. In the **Connect to Server** dialog box, in the **Server name** box, type **MIA-SQL**. Ensure **Windows Authentication** is selected in the **Authentication** box, and then click **Connect**.
3. On the **File** menu, point to **Open**, click **File**, navigate to the **D:\Demofiles\Mod05\Demo2.sql** script file, and then click **Open**.
4. To turn statistics on, highlight the code under **Step 1** and click **Execute**.
5. To run a query with no indexes, highlight the code under **Step 2** and click **Execute**.
6. To create indexes, highlight the code under **Step 3** and click **Execute**.
7. To run a query with indexes, highlight the code under **Step 4** and click **Execute**.

8. To drop the indexes, highlight the code under **Step 5** and click **Execute**.
9. Close SQL Server Management Studio without saving any changes.

Verify the correctness of the statement by placing a mark in the column to the right.

Statement	Answer
True or false? The use of the * predicate in SELECT statements has an adverse effect on query performance.	

Lesson 3
Concurrency

This lesson describes the performance effects of concurrency. When more than one user accesses a resource at the same time, they are said to be accessing that resource concurrently. This requires certain mechanisms to be in place to prevent adverse effects when a user tries to modify resources that other users are actively using. This is called concurrency control. If a data storage system has no concurrency control, users could see the following side effects:

- Lost updates.
- Uncommitted dependency (dirty read).
- Inconsistent analysis (nonrepeatable read).
- Phantom reads.
- Missing and double reads caused by row updates.

Lesson Objectives

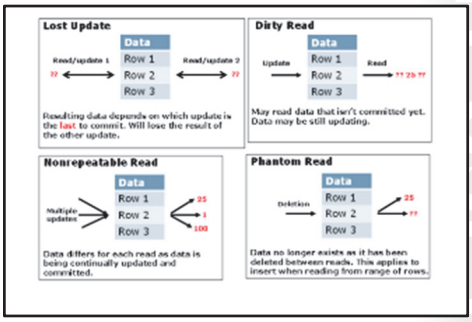
On completion of this lesson, you will have gained an understanding of the performance effects of the following concurrency issues:

- Transactions
- Isolation levels
- Locking

Transactions

As mentioned in the lesson overview, lack of concurrency control can lead to the following data errors:

- Lost updates.
- Uncommitted dependency (dirty read).
- Inconsistent analysis (nonrepeatable read).
- Phantom reads.
- Missing and double reads caused by row updates.



Lost updates

Lost updates occur when two or more transactions select the same row, and then update the row based on the value originally selected. Each transaction is unaware of the other transactions. The last update overwrites updates made by the other transactions, which results in lost data.

For example, two users have downloaded the same file and are updating it independently of each other. When the first user has finished, they upload the changed file, overwriting the original. The second user then completes their updates and uploads *their* changes to the file. This overwrites the first user's changes which are then lost.

This problem might be avoided if one user could not access the file until the other had finished and committed the transaction. This is further described in the *Locking* topic later in this module.

Uncommitted dependency (dirty read)

Uncommitted dependency occurs when a second transaction selects a row that is being updated by another transaction. The second transaction is reading data that has not yet been committed and might be changed by the transaction updating the row.

For example, a user might make changes to a file that another user accesses before the first user has completed their updates. This second user then distributes the file. Subsequently, the first user decides that the changes made are wrong and should not have been made. This means that the distributed file contains changes that no longer exist and should be treated as if they had never existed.

This problem could be avoided if no one could read the changed data until the first transaction has been completed and committed.

Inconsistent analysis (nonrepeatable read)

Inconsistent analysis occurs when a second transaction accesses the same row several times and reads different data each time. Inconsistent analysis is similar to uncommitted dependency in that another transaction is changing the data that a second transaction is reading. However, in inconsistent analysis, the data read by the second transaction was committed by the transaction that made the change. Also, inconsistent analysis involves multiple reads (two or more) of the same row, in addition to each time the information is changed by another transaction—the nonrepeatable read.

For example, a user reads the same data twice, but between each reading, another user has modified the data. When the first user reads the data for the second time, it has changed. The original read was not repeatable.

This problem might be avoided if the data could not be changed until the first transaction had finished reading it for the last time.

Phantom reads

Phantom reads occur when an insert or delete action is performed against a row that belongs to a range of rows being read by a transaction. The transaction's first read of the range of rows shows a row that no longer exists in the second or succeeding read, as a result of a deletion by a different transaction. Similarly, the transaction's second or succeeding read shows a row that did not exist in the original read, as the result of an insertion by a different transaction.

For example, a user reads data from a range of rows in a table. Later, the same user performs the same read operations. In the meantime, another user has deleted or inserted rows in the range being read. This results in data being in the first read but not the second (as a result of a delete operation); or, conversely, data being in the second read but not the first (as a result of an insert operation).

Similar to the nonrepeatable read situation, this problem could be avoided if no one could read data until the transactions that are adding or deleting data have completed and been committed.

Missing and double reads caused by row updates

Missing an updated row or seeing an updated row multiple times

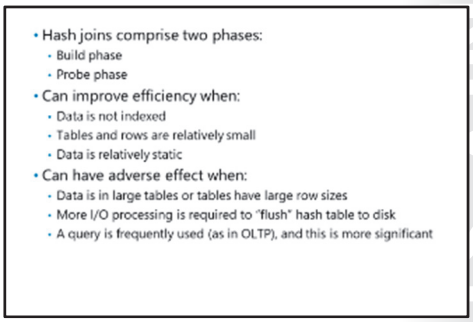
Transactions that are running at the READ UNCOMMITTED level do not issue shared locks to prevent other transactions from modifying data read by the current transaction. Transactions that are running at the READ COMMITTED level do issue shared locks, but the row or page locks are released after the row is read. In either case, when you are scanning an index, if another user changes the index key column of the row during your read, the row might appear again if the key change moved the row to a position ahead of your scan. Similarly, the row might not appear if the key change moved the row to a position in the index that you had already read. To avoid this, use the SERIALIZABLE or HOLDLOCK hint, or row versioning.

Missing one or more rows that were not the target of the update

When you are using READ UNCOMMITTED, if your query reads rows using an allocation order scan (using IAM pages), you might miss rows if another transaction is causing a page split. This cannot occur when you are using read committed because a table lock is held during a page split. This will not happen if the table does not have a clustered index, because updates do not cause page splits.

Additional Reading: For more information, see the *Isolation Level* and *Locking* topics later in this module.

Isolation Level



You can specify an isolation level for transactions. This defines the degree to which one transaction must be isolated from data modifications made by other transactions. Isolation levels are described in terms of which concurrency side effects, such as dirty reads or phantom reads, are allowed.

Transaction isolation levels control:

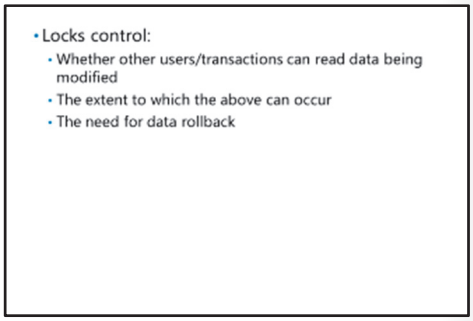
- Whether locks are implemented when data is read, and what type of locks are requested.
- How long the read locks are held.
- Whether a read operation references rows that are being modified by another transaction.
- Blocking the read operation until the exclusive lock on the row is freed. This is the highest isolation level and does not allow any of the concurrency side effects.
    - **Retrieves the last-committed version of the row**. This will be the one that existed at the time the read operation started. This isolation level will potentially result in data updates being lost, but will at least ensure that the data being read is stable (though not necessarily the latest version).
    - **Reads the uncommitted data modification**. This is the lowest isolation level and allows all the concurrency side effects.

Choosing a transaction isolation level does not affect the locks necessary to protect data modifications. When modifying data, a transaction always gets an exclusive lock on that data, holding the lock until it has completed the modification, regardless of the isolation level set for that transaction. For read operations, transaction isolation levels primarily define the level of protection from the effects of modifications made by other transactions.

With a lower isolation level, many users can access data at the same time, but this increases the number of concurrency effects (such as dirty reads or lost updates) that users might encounter. On the other hand, a higher isolation level reduces the types of concurrency effects that users may encounter, but requires more system resources and increases the chances that one transaction will block another. Choosing the appropriate isolation level is a balancing act between the data integrity requirements of the application and the overhead intrinsic in each isolation level. The highest isolation level, serializable, guarantees that a transaction will retrieve exactly the same data every time it repeats a read operation—it does this by performing a level of locking that is likely to impact other users in multiuser systems. The lowest isolation level, read uncommitted, may retrieve data that has been modified but not committed by other transactions. All of the concurrency side effects can happen in read uncommitted, but there is no read locking or versioning, so overhead is minimized.

Locking

Data can be locked to prevent other users from accessing it when it is being modified by another user or transaction. These locks are part of concurrency control.



Concurrency control theory has two classifications for the methods of instituting concurrency control:

- Pessimistic concurrency control
- Optimistic concurrency control

Pessimistic concurrency control

A system of locks prevents users and transactions from modifying data in a way that affects other users and transactions. After an action is performed that causes a lock to be applied, other users cannot perform actions that would conflict with the lock until the owner releases it. This is called pessimistic control because it is mainly used in environments where there is high contention for data; where the cost of protecting data with locks is less than the cost of rolling back transactions if concurrency conflicts occur.

Optimistic concurrency control

In optimistic concurrency control, users do not lock data when they read it. When a user updates data, the system checks to see if another user changed the data after it was read. If another user updated the data, an error is raised. Typically, the user receiving the error rolls back the transaction and starts again. This is called optimistic because it is mainly used in environments where there is low contention for data, and where the cost of occasionally rolling back a transaction is lower than the cost of locking data when read.

Microsoft SQL Server supports a range of concurrency control. Users specify the type of concurrency control by selecting transaction isolation levels for connections or concurrency options on cursors. These attributes can be defined using Transact-SQL statements, or through the properties and attributes of database application programming interfaces (APIs), such as ADO, ADO.NET, OLE DB, and ODBC.

Demonstration: Locking and Blocking

In this demonstration, you will see the effects of locking and blocking.

Demonstration Steps

1. On the taskbar, click **Microsoft SQL Server Management Studio**.
2. In the **Connect to Server** dialog box, in the **Server name** box, type **MIA-SQL**. Ensure **Windows Authentication** is selected in the **Authentication** box, and then click **Connect**.
3. On the **File** menu, point to **Open**, click **File**, navigate to the **D:\Demofiles\Mod05\Demo3A.sql** script file, and then click **Open**.
4. On the **File** menu, point to **Open**, click **File**, navigate to the **D:\Demofiles\Mod05\Demo3B.sql** script file, and then click **Open**.
5. To start the first query, switch to the **Demo3A** tab, highlight the code under **Step 1** and click **Execute**.
6. Immediately switch to the **Demo3B** tab, highlight the code under **Step 2** and click **Execute**.

7. Note that the query does not complete because it is being blocked.
8. Switch to the **Demo3A** tab and, on the toolbar, click **Cancel Executing Query**.
9. Wait until the cancellation has completed and switch to the **Demo3B** tab.
10. Notice that the query has now completed because the locks have been released.
11. Close SQL Server Management Studio without saving any changes.

Verify the correctness of the statement by placing a mark in the column to the right.

Statement	Answer
True or false? Isolation levels affect the locks imposed when a transaction is modifying data.	

Lab: Performance Issues

Scenario

You are planning a database and need to decide on an indexing strategy. There are many rows in the database, but at the moment the number of different queries that will run is limited; you will revise the indexing when the database is in use.

Initially, you will plan indexing for data that is static and has no insert, updates, or deletes; you will then also consider the effects of large numbers of transactions.

Objectives

After completing this lab, you will be able to:

- Plan an indexing strategy.

Lab Setup

Estimated Time: 30 minutes

Virtual machine: **10985C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa55w.rd**

This is a paper-based lab and the virtual machines are only required to access supporting material.

Exercise 1: Designing for Efficient Queries

Scenario

You are planning a database and need to decide on an indexing strategy. There are many rows in the database, but at the moment the number of different queries that will run is limited; you will revise the indexing when the database is in use.

Initially, you will plan indexing for data that is static and has no insert, updates, or deletes, but you will then also consider the effects of large numbers of transactions.

The main tasks for this exercise are as follows:

1. Plan Indexes for Querying

2. Plan Indexes for Concurrency

▶ Task 1: Plan Indexes for Querying

1. Ensure that the 10985C-MIA-DC and 10985C-MIA-SQL virtual machines are both running, and then log on to 10985C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. In the **D:\Labfiles\Lab05\Starter** folder, double-click **DatabaseQueryDesign.docx**.
3. Plan which clustered and nonclustered indexes you would add to this design, based on the design of the tables and the common queries.
4. In the **D:\Labfiles\Lab05\Solution** folder, double-click **PossibleIndexStrategy.docx**.
5. Review and discuss any differences between your strategy and the proposed index strategy in the document.

▶ Task 2: Plan Indexes for Concurrency

1. If the database in the design has handled many transactions, in addition to the listed queries, note the effects this could have on the transactions.
2. Note the effects this could have on the listed queries.
3. Note the changes that you might make to the index strategy to support transactions.

Results: After completing this exercise, you will have created an index plan.

Module Review and Takeaways

Best Practice: Indexes, joins and the types of search that you perform have a huge effect on query performance so you should consider index and query design carefully. Furthermore, most databases will have many concurrent users—you should consider whether a database that performs optimally in a test environment with one use, would perform optimally in a production environment.

Module 6

Database Objects

Contents:

Module Overview

SQL Server® databases can contain many different types of objects, including tables, stored procedures, and views. Most implementations of SQL Server databases include these types of objects, so it is important to understand how to use them and the benefits that they provide.

Objectives

After completing this module, you will be able to:

- Create tables and use data types and constraints to ensure integrity.
- Create and use views.
- Create and use stored procedures, functions, and triggers.

Lesson 1
Tables

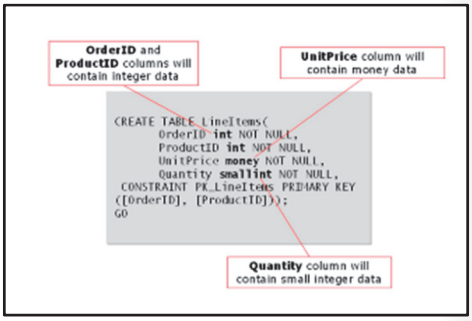
Entities are implemented in databases as tables, and attributes are implemented as columns in tables. Each row in a table represents a discrete instance of the entity; for example, a row in a table called Customers represents an individual customer. You can use primary key constraints and foreign key constraints to ensure uniqueness and to implement referential integrity, which helps to ensure that the data is meaningful. However, there are other configuration options that you can use to help maintain data integrity, including data types, UNIQUE constraints, CHECK constraints, DEFAULT constraints, NULL, and NOT NULL.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the use of data types when creating SQL Server tables.
- Describe the numeric data types in SQL Server.
- Describe the character string data types in SQL Server.
- Describe some of the other data types in SQL Server.
- Explain how to use constraints, including CHECK, UNIQUE, and DEFAULT constraints.
- Explain how to use the IDENTITY property when creating tables.
- Describe the role of schemas in SQL Server.

Introduction to Data Types



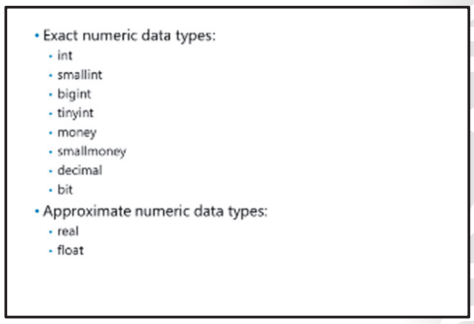
When you create a table, you define each column by specifying a name for the column, and a data type. The data type specifies the type of data that you can insert into the column. For example, if you specify the "int" data type for a column, the column can only contain integers within a defined range of values. Any attempt to insert data that is not in the correct format, such as textual data, will not succeed. A data type acts as a constraint, like a primary key constraint or a foreign constraint, because it restricts the data that you can include in a column. SQL Server supports a range of data types, helping you to manage simple data, such as character strings and numerical values, in addition to core complex data such as XML data.

The following code example uses the int, money and smallint data types in a CREATE TABLE Transact-SQL statement:

CREATE TABLE Transact-SQL Statement

```
CREATE TABLE LineItems(
      OrderID int NOT NULL,
      ProductID int NOT NULL,
      UnitPrice money NOT NULL,
      Quantity smallint NOT NULL,
 CONSTRAINT PK_LineItems PRIMARY KEY
([OrderID], [ProductID]));
GO
```

Numeric Data Types



Numeric data types include exact numeric data types, which store integer values, and approximate numeric data types, which store decimal values.

Exact numeric data types

Exact numeric data types include the following:

- **int**, which uses four bytes to store integer data values between -2,147,483,648 and 2,147,483,647.
- **smallint**, which uses two bytes to store integer data values between -32,768 and 32,767.
- **bigint**, which uses eight bytes to store integer data values between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807.
- **tinyint**, which uses one byte to store integer data values between 0 and 255.
- **money**, which uses eight bytes to store monetary values to four decimal places.
- **smallmoney**, which uses four bytes to store monetary values to four decimal places.
- **decimal**, which stores exact decimal numbers. The number of bytes required varies, depending on the precision that you specify. The data type numeric duplicates the functionality of decimal.
- **bit**, which uses a single bit to store the values 1 or 0. The bit data type is useful for columns that can only contain two different values; for example, a MaritalStatus column in a Customers table might use the value 1 to indicate married and 0 to indicate unmarried.

It is important to select a data type that is appropriate for the data that a column will store, because this can have an impact on the size of a database and on performance. For example, in a LineItems table, in an order-processing OLTP database, there might be a column called Quantity that specifies the number of items of a particular product that were included in an order. You could use the int data type for this column, which would require storing four bytes of data for every row.

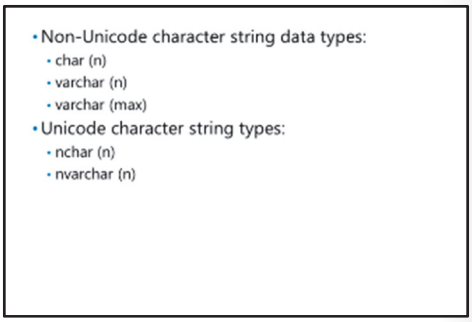
However, if you know that the number of product items in a given order will always be fewer than 500, for example, you could use smallint instead, because this would require only two bytes to store the same data. This would reduce storage requirements by two bytes for each row, which could result in a significant reduction in size for tables with a large number of rows. Additionally, smaller rows can result in better performance. SQL Server performs best when it can service transactions by using data cached in memory, instead of retrieving the data from disk. Ensuring that rows are as small as possible means that more rows can fit into memory, and consequently that SQL Server does not need to access data on disk as frequently.

Approximate numeric data types

In SQL Server, approximate numeric data types store decimal values by rounding them up. The available approximate numeric data types are float and real. You would usually only use these data types if the range of precision offered by the decimal data type is insufficient.

Character String Data Types

You can store character string data by using the following data types:



- **char (n)**, which stores character data as fixed length strings. You can specify the number of characters (n) when you define the char data type for a column. Each character requires one byte to store it. For example, if you specify char (20), then the column will use 20 bytes to store up to 20 characters in a string. Note that when you use char, every value in the column will be the same length, as specified by the value (n), even if some values are shorter than (n) characters.
- **varchar (n)**, which stores character data as variable length strings. With the varchar data type, the value (n) represents a maximum number of characters. Unlike char, varchar stores only the required number of characters for each row. For example, if you specify varchar (20) for a column, but specify a value that is six characters in length, then SQL Server will only use six bytes to store the character instead of 20, as it would if you used char (20). This results in smaller rows, which can be beneficial for the reasons described earlier. The varchar data type is normally used when string lengths are likely to be inconsistent, and char is often used when character string lengths are predictable.
- **varchar (max)**, which means you can store character strings that are larger than the maximum 8,000 bytes.

If a database includes data in multiple alphabets, it can be better to use Unicode storage for character strings because this helps you to store a wider range of characters. Unicode uses two bytes to store each character instead of one; so for each character, there are 65,536 possibilities instead of just the 256 possibilities that one byte provides. You can specify Unicode storage by using data types, including nchar (n), nvarchar (n), and nvarchar (max). These data types work in the same way as their non-Unicode equivalents, except that they use twice as much storage space per character.

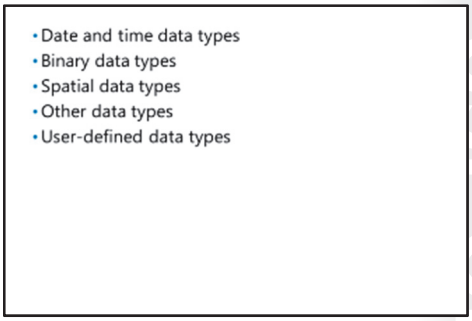
Note that the following character string data types are deprecated:

- text
- ntext

For new databases you should use varchar(max), nvarchar(max) instead.

Other Data Types

SQL Server includes numerous additional data types that you can use to store a range of data. These include date and time data types, data types for handling specialist data, such as xml or hierarchical data, and binary data types. With SQL Server, you can also create user-defined data types that you can use to ensure consistency of data type usage across multiple columns.



Date and time data types

The data types that you can use to store date and time data include:

- **datetime**, which uses eight bytes to store dates between January 1, 1753 and December 31, 9999, and times between 00:00:00 and 23:59:59.997.
- **datetime2**, which uses between six and eight bytes (depending on precision) to store dates between 0001-01-01 and 9999-12-31, and times between 00:00:00 and 23:59:59.9999999. The datetime2 data type has a wider date range than datetime and supports user-defined precision.
- **date**, which uses three bytes to store dates between 0001-01-01 and 9999-12-31.
- **time**, which uses five bytes to store times between 00:00:00.0000000 and 23:59:59.9999999.
- **smalldatetime**, which uses four bytes to store dates between 1900-01-01 and 2079-06-06, and times between 00:00:00 and 23:59:59.

Binary data types

To store binary data, you can use the following data types:

- **binary (n)**, which stores fixed-length binary data. The number of bytes used depends on the user-defined value (n).
- **varbinary (n)**, which stores variable-length binary data. As with the varchar data type, the maximum number of bytes used depends on the user-defined value (n), and the actual number of bytes used in a given row depends on the length of the actual data value for that row.
- **varbinary (max)**, which stores larger binary values. You should use varbinary (max) instead of the image data type, which is deprecated.

Spatial data types

- **geography**, which you can use to store ellipsoidal data.
- **geometry**, which you can use to store planar spatial data.

For more information about spatial data types, see *Spatial Data (SQL Server)* in Microsoft Docs:

Spatial Data (SQL Server)

http://aka.ms/Of1ksy

Data types for storing other types of data

- **xml**, which you can use to store XML data in columns in tables. The xml data type supports features such as fine-grained queries, updates to XML data, and creating views on xml columns, so you can use Transact-SQL to query them.

  For more information about the xml data type, see *XML Data Type and Columns (SQL Server)* in Microsoft Docs:

  XML Data Type and Columns (SQL Server)

  http://aka.ms/th3v7b

- **hierarchyID**, which you can use to store data that is hierarchically related to data in other columns. The hierarchyID data type stores a path that represents a given value's place in a hierarchy.

  For more information about the hierarchyID data type, see *hierarchyID (Transact-SQL)* in Microsoft Docs:

  hierarchyID (Transact-SQL)

  http://aka.ms/mdu4wx

User-defined data types

A user-defined data type is a data type that you define, based on an existing system data type. For example, if you have multiple columns, perhaps in different tables, that must all have identical length, data type, and nullability options, you can use a CREATE TYPE statement to create a user-defined data type, and use this data type when defining the columns.

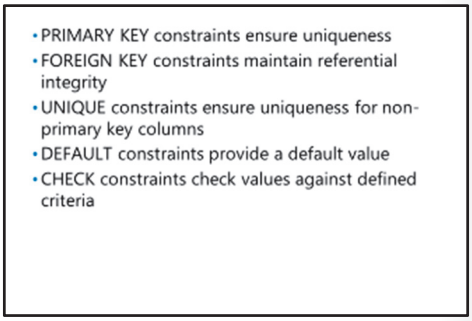
The following Transact-SQL statement creates a data type called SSN that is based on the char (n) data type, which does not allow NULLs.

CREATE TYPE Transact-SQL Statement

```
CREATE TYPE SSN
FROM char(11) NOT NULL ;
GO
```

Keys and Constraints

Constraints are database objects that you can create to restrict the type of data that a column can contain. In previous modules, you learned about primary keys, which ensure uniqueness, and foreign keys, which maintain referential integrity. SQL Server supports other constraints, including UNIQUE constraints, DEFAULT constraints, and CHECK constraints.



UNIQUE constraints

A UNIQUE constraint checks the values in a column to ensure that they are unique, in much the same way as a PRIMARY KEY constraint does. UNIQUE constraints are useful when it is necessary to apply uniqueness to columns that do not form part of a table's primary key. There are several important differences between the two types of constraint:

- It is possible to create multiple UNIQUE constraints on a table, but a table can have only one PRIMARY KEY constraint.
- A column with a UNIQUE constraint can include a single NULL value. A column with a PRIMARY KEY constraint cannot include any NULLs.
- By default, creating a PRIMARY KEY constraint also creates a clustered index that sorts the table by the primary key column or columns. Creating a UNIQUE constraint on a column creates a nonclustered index on that column by default.

The following Transact-SQL statement adds a UNIQUE constraint to the SSN column:

Add a UNIQUE constraint

```
ALTER TABLE Employees
ADD CONSTRAINT AK_SSN UNIQUE (SSN);
GO
```

DEFAULT constraints

A DEFAULT constraint includes a value that is inserted into a column if the INSERT statement does not include a specific value for that column. This can be useful in helping to minimize or eliminate NULLs in tables. NULLs can be problematic in tables because they can give rise to misleading or inaccurate query results if not handled correctly. For example, imagine a CustomerDemographics table with a column called NumberOfChildren, which allows NULLs to account for situations where a customer does not disclose this information. A person reading this data could potentially interpret the NULLs to mean 'zero children', when in fact it means that the number of children is unknown. To remove this ambiguity, you could create the column by using the NOT NULL option, and additionally using a DEFAULT constraint that enters the value 'Unknown' when no value is supplied for the NumberOfChildren column.

The following Transact-SQL statement adds a DEFAULT constraint that supplies the value 'Unknown' for the NumberOfChildren column:

Add a DEFAULT constraint

```
ALTER TABLE CustomerDemographics
ADD CONSTRAINT DF_NumberChildren DEFAULT 'Unknown' FOR NumberOfChildren;
GO
```

CHECK constraints

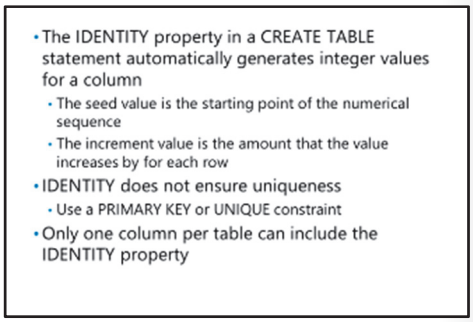
A CHECK constraint checks values that are entered into a column against the defined criteria. This helps to ensure that the values entered are valid. For example, it might be necessary to ensure that the dates that an application enters into the OrderDate column in the Orders table are for the current date only. A CHECK constraint could check the entered order dates and prevent the entry of orders that have a past or future order date. Alternatively, you could allow dates that are up to two days before or after the current date to allow users to enter a wider range of order dates if required.

The following Transact-SQL statement adds a CHECK constraint that checks the values in the OrderDate column to ensure that they are for the current date. The WITH NOCHECK option prevents the application of the constraint to existing values in the OrderDate column:

Add a CHECK constraint

```
ALTER TABLE Orders WITH NOCHECK ADD CONSTRAINT CK_OrderDate
    CHECK (OrderDate = GETDATE());
GO
```

The Identity Property



You can use the IDENTITY property when you create a table to automatically add a value to a column for each row. The IDENTITY property generates incremental values, beginning with a user-defined seed value, and increasing by a user-defined increment value. For example, you could use the IDENTITY property to automatically populate the OrderID column in the Orders table, starting with a seed value of 1, and incrementing by 1.

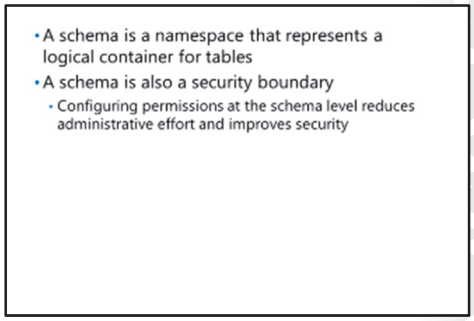
The following Transact-SQL code example creates a table called Orders that includes the IDENTITY property for the OrderID column:

CREATE TABLE statement with the IDENTITY property

```
CREATE TABLE Orders
(OrderID INT IDENTITY (1,1) NOT NULL
,CustomerID INT NOT NULL
,OrderDate DATETIME NOT NULL
CONSTRAINT PK_OrderID PRIMARY KEY (OrderID));
```

Note that the IDENTITY property does not ensure uniqueness. With the IDENTITY property configured, a user can still add a duplicate value to the IDENTITY column. To ensure uniqueness, you need to use a PRIMARY KEY or a UNIQUE constraint. It is very common to configure the IDENTITY property to supply values for surrogate PRIMARY KEY columns, in which the value only needs to be unique, and does not represent anything that exists outside the scope of the database. You can designate only one IDENTITY column for each table.

Schemas



When you create a table in a SQL Server database, you can specify a schema for the table as part of the CREATE TABLE statement. A schema is a namespace that represents a logical container for tables. For example, you could create a schema called Product, and then create all product-related tables in that schema.

The following Transact-SQL code example creates a schema:

CREATE SCHEMA Transact-SQL statement

```
CREATE SCHEMA Product;
GO
```

After creating a schema, you can create tables in it.

The following Transact-SQL code example creates a table called ProductCategory in the Product schema:

CREATE TABLE Transact-SQL statement that includes a schema

```
CREATE TABLE Product.ProductCategory
(ProductCategoryID int IDENTITY PRIMARY KEY
,ProductCAtegoryName varchar (50));
GO
```

Schemas as security boundaries

To help you to logically group tables in a database in a more intuitive way, you can also use schemas to manage database security more efficiently. So that users can access resources, or to prevent them from doing so, you define permissions. For example, you might give read permission on the ProductCategory table to a database user called Jim. This would mean the user could view, but not change, data in this table. When you have a large number of tables and users, defining and maintaining permissions can be a time-consuming task. Using schemas reduces the time required to define and maintain permissions because tables inherit the permissions that you define at the schema level. Consequently, if you configure permissions at the schema level, there is no need to also configure them at the table level, unless they need to be different from the schema permissions. This is more efficient from an administrative point of view and also more secure, because it reduces the complexity of configuring permissions, which in turn reduces the likelihood of incorrectly configured permissions.

Demonstration: Creating Databases and Tables

In this demonstration, you will see how to:

- Create a database.
- Create tables.
- Create schemas.
- Create a CHECK constraint.
- Create a DEFAULT constraint.
- Create a UNIQUE constraint.

Demonstration Steps

1. Start the 10985C-MIA-DC and 10985C-MIA-SQL virtual machines, and then log on to 10985C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. Open **Microsoft SQL Server Management Studio**, and then connect to the **MIA-SQL** instance of the database engine by using Windows Authentication.
3. On the **File** menu, point to **Open**, click **File**, browse to **D:\Demofiles\Mod06**, click **Create Tables.sql**, and then click **Open**.
4. In the query window, under the comment **Create a database and change database context**, review the Transact-SQL statement, select it, and then click **Execute**.
5. In the query window, under the comment **Create schemas**, review the Transact-SQL statement, select it, and then click **Execute**.
6. In the query window, under the comment **Create tables**, review the Transact-SQL statement, select it, and then click **Execute**.
7. In the query window, under the comment **Add CHECK constraint to Sales.Order**, review the Transact-SQL statement, select it, and then click **Execute**.
8. In the query window, under the comment **Test CHECK constraint**, review the Transact-SQL statement, select it, and then click **Execute**.
9. Review the results, and note that the insert fails because the order date is yesterday's date.
10. In the query window, under the comment **Add DEFAULT constraint to Person.CustomerDemographics**, review the Transact-SQL statement, select it, and then click **Execute**.
11. In the query window, under the comment **Test DEFAULT constraint**, review the Transact-SQL statement, select it, and then click **Execute**.
12. Review the results, and note that the insert succeeds and the default value 'Unknown' is entered.
13. In the query window, under the comment **Add UNIQUE constraint to Person.Employee**, review the Transact-SQL statement, select it, and then click **Execute**.
14. In the query window, under the comment **Test UNIQUE constraint**, review the Transact-SQL statement, select it, and then click **Execute**.
15. Review the results, and note that the first insert succeeds, but the second fails because it contains the duplicate **SSN** value '123'.
16. Close SQL Server Management Studio without saving changes.

Categorize Activity

Place each item into the appropriate category. Indicate your answer by writing the category number to the right of each item.

Items	
1	Checks the values that are added to a column and prevents the insertion of duplicate values.
2	Inserts a pre-configured value into a column when an INSERT statement does not provide a value for that column.
3	Compares a value to a pre-configured condition to assess whether the value is valid.

Category 1		Category 2		Category 3
Unique constraint		Default constraint		Check constraint

Lesson 2
Views

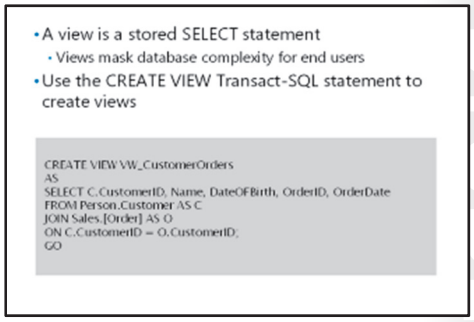
Databases can sometimes be complex, and may include hundreds of tables, or even more. For end users who access the data in a database, this complexity can be confusing. In SQL Server, you can create views to help you to present the data to end users in a more user-friendly way.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe views in SQL Server.
- Explain how to use views securely.
- Explain how to implement views to optimize performance.

What Is a View?



The complexity of a database can be confusing for end users. A highly normalized database will contain many tables, and the data that a user might require is often spread across multiple columns in multiple tables. Writing a Transact-SQL query to return data may involve using more advanced queries that include JOINs, for example. A JOIN is an operation that matches rows in one table against rows in another table, based on the values in a column that exists in both tables. Usually, this is a foreign key column in one table and a primary key column in the other. Views can be useful in this kind of scenario because they mask this underlying complexity, and help users to write much simpler queries.

A view is a stored Transact-SQL SELECT statement. To create a view, you write a CREATE VIEW Transact-SQL statement, and include the SELECT statement that you want the view to return.

The following CREATE VIEW statement creates a view called VW_CustomerOrders that includes a SELECT statement that combines columns from two tables, called Sales.Orders and Person.Customer. The SELECT statement allocates each table an alias (C for Person.Customer and O for Sales.Orders), which is used in the JOIN clause:

CREATE VIEW statement

```
CREATE VIEW VW_CustomerOrders
AS
SELECT C.CustomerID, Name, DateOfBirth, OrderID, OrderDate
FROM Person.Customer AS C
JOIN Sales.[Order] AS O
ON C.CustomerID = O.CustomerID;
GO
```

To use a view, you write a SELECT statement that references the view as if it were a table. A SELECT statement against a view returns a data set in the same way as a SELECT statement against a table does.

The following Transact-SQL SELECT statement selects all rows from the view called VW_CustomerOrders:

SELECT statement that references a view

```
SELECT * FROM VW_CustomerOrders;
GO
```

You can also specify the columns that you want to return from a view by listing them in the SELECT clause, in the same way as you would if querying against a table. In addition to selecting data by using a view, you can also update, insert, and delete data in the underlying tables, providing the changes only reference columns from a single base table.

SQL Server includes an extensive set of system views, which you can use to access metadata about SQL Server for maintenance and troubleshooting.

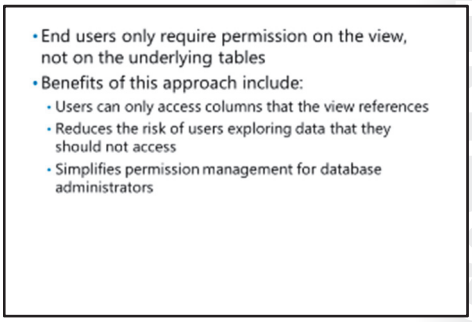
For more information about system views in SQL Server, see *System Views (Transact-SQL)* in Microsoft Docs:

System Views (Transact-SQL)

http://aka.ms/l3d5zc

Views and Security

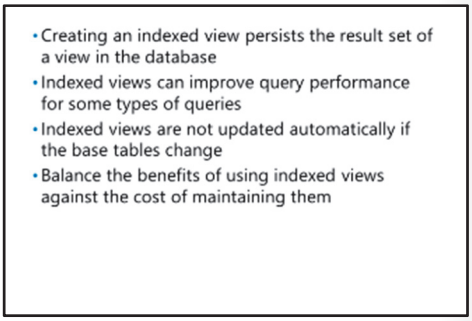
So that users can access tables and other database objects, you configure permissions on those objects. When you configure permissions on a view, you do not need to also configure permissions to grant access on the underlying tables. For example, with the VW_CustomerOrders view, you could give users READ permission against the view, and they could write SELECT statements against that view without needing permissions on the Person.Customer or Sales.Order tables. However, the user who creates a view *does* require permission to access the underlying tables that the view references.



This approach to security is useful, because:

- The only way that users can access the tables is through the view, unless they also have explicit permission to access the tables. Furthermore, users can only access the columns that the view references, and not the other columns in the underlying tables.
- Users can treat a view in the same way as they would treat a table (for the most part); users might not even be aware that they are using a view to access data, and they only see the data that the view presents to them. This reduces the risk of users trying to explore data that they should not be able to access, which is likely to happen if you simply give users permission directly on the underlying tables.
- It provides database administrators with a fast and easy way to configure permissions; it is less complex and less prone to error than configuring permissions on multiple tables.

Views and Performance



A view is a logical table, and the data set that a view returns is not persisted in the database. Every time that a user runs a SELECT statement against the view, SQL Server creates the result set in the same way as if the user had run the SELECT statement against the tables directly. Consequently, a view does not offer any performance benefits over running a standard SELECT statement. However, it is possible to improve performance by using views if you create an indexed view. With an indexed view, the index is the results set that the view returns. The index is persisted in the database, and can be used any time that the view is referenced in Transact-SQL statements. Persisting a view by using an index in this way is sometimes referred to as materializing a view.

As with any other index, an indexed view provides SQL Server with a potentially faster way to access data. An index created on a view contains less data than an index on an underlying table because most views only select a subset of the columns and rows in a table. Smaller indexes are generally faster to read than larger indexes. Furthermore, SELECT statements against multiple tables might involve using multiple table indexes to return the data; however, with an indexed view, all of the data is present in a single index, which is again potentially more efficient.

Considerations for indexed views

Indexed views are not guaranteed to improve performance for every query. They are particularly effective for views that include aggregations and calculations, because an index on this type of view will contain the completed aggregations and calculations. Consequently, when a user queries the view, SQL Server does not need to perform aggregations or calculations; it can simply read them directly from the index.

Indexed views are not suitable for every scenario. You should consider carefully the trade-off between any performance gain and the cost of maintaining the index, which can be considerable if the data in the base tables changes frequently, and it is important to keep the indexed view up to date. An indexed view represents a snapshot of the data at the point in time that the index was created and is not automatically updated as the base tables change. There are various requirements for creating indexes on views, such as the requirement to use the WITH SCHEMABINDING option when creating a view—you should review this before you attempt to use the feature.

For more information about creating indexed views, see *Create Indexed Views* in Microsoft Docs:

Create Indexed Views

http://aka.ms/tprnh4

Demonstration: Creating and Using Views

In this demonstration, you will see how to:

- Create a view.
- Query a view.
- Create an index on a view.
- Locate views and indexes by using Object Explorer.

Demonstration Steps

1. In the **D:\Demofiles\Mod06** folder, right-click **Setup.cmd**, and then click **Run as administrator**.
2. In the **User Account Control** dialog box, click **Yes**, and wait for setup to complete.
3. Open **Microsoft SQL Server Management Studio**, and then connect to the **MIA-SQL** instance of the database engine by using Windows Authentication.
4. On the **File** menu, point to **Open**, click **File**, browse to **D:\Demofiles\Mod06**, click **Create Views.sql**, and then click **Open**.
5. Under the comment **Create a view**, review the Transact-SQL statements, select them, and then click **Execute**.
6. Under the comment **Query the view**, review the Transact-SQL statements, select them, and then click **Execute**.
7. Under the comment **Query the view with a WHERE clause**, review the Transact-SQL statements, select them, and then click **Execute**.
8. Under the comment **Create an index on the view**, review the Transact-SQL statements, select them, and then click **Execute**.
9. In Object Explorer, expand **Databases**, expand **Demo**, expand **Views**, and note the view that you created in step 5.
10. Expand **dbo.VW_CustomerOrders**, expand **Indexes**, and note the index that you created in step 8.
11. Close SQL Server Management Studio without saving changes.

Verify the correctness of the statement by placing a mark in the column to the right.

Statement	Answer
True or false? To create an indexed view, you must include the WITH SCHEMABINDING option in the view definition.	

Lesson 3
Stored Procedures, Triggers, and Functions

In addition to the tables, views, and other database objects that this module has described, there are other objects that occur in SQL Server databases. These include stored procedures, triggers, and functions.

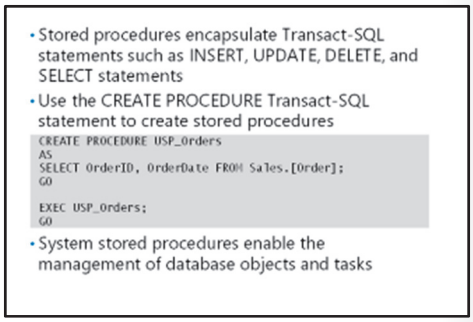
Lesson Objectives

After completing this lesson, you will be able to:

- Explain how to use stored procedures in SQL Server.
- Describe the benefits of using stored procedures.
- Explain how to use input parameters in stored procedures.
- Describe the different types of triggers in SQL Server.
- Describe the different types of functions in SQL Server.

Stored Procedures

A stored procedure is a SQL Server database object that encapsulates a Transact-SQL statement. Unlike views, which only encapsulate SELECT statements, stored procedure definitions can include a wider range of Transact-SQL statements, such as UPDATE, INSERT, and DELETE statements, as well as SELECT statements. Furthermore, you can include procedural logic in Transact-SQL statements in stored procedures—this provides the flexibility to help you to implement complex business rules.



To create a stored procedure, you can use the CREATE PROCEDURE Transact-SQL statement. This statement can also be used in the shortened CREATE PROC format.

The following code example creates a stored procedure called USP_Orders, which includes a SELECT statement. The code example also includes an EXEC statement. You use the EXEC keyword to run stored procedures in SQL Server.

CREATE PROCEDURE Transact-SQL statement

```
CREATE PROCEDURE USP_Orders
AS
SELECT OrderID, OrderDate FROM Sales.[Order];
GO

EXEC USP_Orders;
GO
```

System stored procedures

SQL Server includes a set of system stored procedures, which you can use to manage a wide range of database objects and tasks. For example, you can use the stored procedure sp_help to view information about database objects. To specify the object that you want to view, you include the name of the object when you execute the procedure.

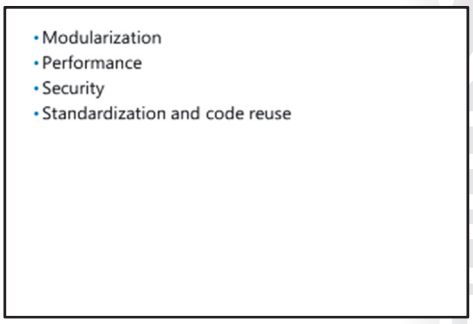
The following code example uses the sp_help system stored procedure to return information about the Sales.Orders table:

The sp_help system stored procedure

```
EXEC sp_help 'Sales.Order'
```

The majority of the system stored procedures in SQL Server have the prefix sp_ in their name. To make it easy to differentiate between user-created stored procedures and system stored procedures, you should avoid creating procedures that use the sp_ prefix.

Benefits of Stored Procedures



Using stored procedures to execute Transact-SQL statements instead of executing statements directly offers numerous benefits, including modularization, performance, security, and standardization.

Modularization

Database users interact with a database by using an application of some kind, such as a webpage. It is more efficient to include the stored procedure in an application's code than it is to include Transact-SQL statements. For example, if an application needs to include an INSERT statement to add a row to a database, you could potentially include all of the code to do this directly in the application. However, by encapsulating the code in a stored procedure and including only the EXEC statement in the application, you gain significant benefits, including:

- Any changes to the Transact-SQL statement can be made in the stored procedure in the database, and there is usually no need to modify the application code.
- The application code is more streamlined and easier to understand.
- If communication between the application and the database server is done over a network, sending a small EXEC statement is more efficient than sending many more lines of Transact-SQL code.

Performance

In addition to reducing the amount of data sent over networks, as described above, stored procedures can also improve performance because they are precompiled. When you execute a Transact-SQL statement, the SQL Server database engine creates an execution plan—by referencing the available indexes and index statistics. The execution plan represents the most efficient way of running the statement at that particular moment, according to the database engine. This process, which is called compilation, can add significantly to the processing time for executing Transact-SQL statements, particularly when there is a large number of queries running simultaneously on a server. The first time a stored procedure executes, the database engine compiles the statement and creates a plan in the same way, but it then stores and reuses this plan for subsequent executions. Because subsequent executions use this precompiled execution plan instead of generating a new one, the time required to process the statement in the procedure is reduced.

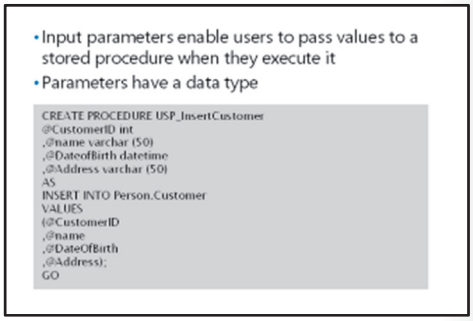
Security

When you enable users and applications to interact with data in a database by using a stored procedure, you only need to configure permissions for the users or applications on the stored procedure itself—not on the database objects that the stored procedure references. This simplifies the security model, and reduces the risk of insecure permissions configuration.

Standardization and code reuse

If the same code is required in multiple applications, it is inefficient to have to write and maintain each instance of the code independently. By using a stored procedure to encapsulate the code, and then using the stored procedure in the various applications, you make the code easier to maintain, and eliminate the duplication of effort involved in writing code more than once.

Input Parameters



When creating a stored procedure, you can include input parameters. An input parameter is a value that the user can supply when they execute the stored procedure. For example, in a stored procedure that inserts a row into a table, you would include a parameter for each value for the columns in the table; for a stored procedure that selects data, you could include a parameter that is used in the where clause to filter the rows that the query returns. To include input parameters in a stored procedure, you declare them when you create the procedure, and assign a data type to each parameter. The data type usually matches the data type of the column that the parameter maps to. Parameter names must begin with the @ symbol.

The following code example creates a stored procedure that inserts a row into a table called Person.Customer, which contains the columns CustomerID, Name, DateOfBirth and Address. The stored procedure includes one parameter for each of these columns.

CREATE PROCEDURE Transact-SQL statement

```
CREATE PROCEDURE USP_InsertCustomer
@CustomerID int
,@name varchar (50)
,@DateofBirth datetime
,@Address varchar (50)
AS
INSERT INTO Person.Customer
VALUES
(@CustomerID
,@name
,@DateOfBirth
,@Address);
GO
```

When a user executes a stored procedure that includes parameters, a value of the correct data type for each of the parameters must be specified. Note that values for char, varchar, and datetime data types must be enclosed in single quotes, but values for int data types should not be.

The following code example uses the EXEC keyword to execute the USP_InsertCustomer stored procedure with parameter values:

EXEC Transact-SQL statement

```
EXEC USP_InsertCustomer @CustomerID = 8
,@name = 'Ken Sanchez'
,@DateOfBirth = '1967-08-07'
,@Address = '1399 Firestone Drive';
GO
```

 **Note:** When creating stored procedures, you should not include parameters that map to columns that have the IDENTITY property defined, because IDENTITY automatically provides system generated values.

Demonstration: Creating and Using a Stored Procedure

In this demonstration, you will see how to:

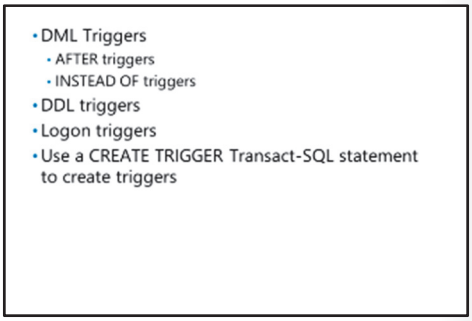
- Create a parameterized stored procedure.
- Execute a stored procedure by passing input parameter values to it.
- Verify the actions performed by the stored procedure.

Demonstration Steps

1. In the **D:\Demofiles\Mod06** folder, right-click **Setup.cmd**, click **Run as Administrator**.
2. In the **User Account Control** dialog box, click **Yes**, and then wait for the script to complete.
3. Open **Microsoft SQL Server Management Studio**, and then connect to the **MIA-SQL** instance of the database engine by using Windows Authentication.
4. On the **File** menu, point to **Open**, click **File**, browse to **D:\Demofiles\Mod06**, click **Create Stored Procedure.sql**, and then click **Open**.
5. Under the comment **Create the stored procedure**, review the Transact-SQL statements, select them, and then click **Execute**.
6. Under the comment **Execute the stored procedure**, review the Transact-SQL statement, select it, and then click **Execute**.
7. Under the comment **View the results**, review the Transact-SQL statement, select it, and then click **Execute**.
8. Review the results, noting that a row has been added to the **Person.Customer** table with the values specified in the EXEC statement.
9. Close SQL Server Management Studio without saving any changes.

Triggers

SQL Server includes database objects called triggers. A trigger is an encapsulated Transact-SQL statement that executes in response to an action performed in a database. You cannot execute a trigger by calling it in a Transact-SQL statement as you can with a stored procedure; only the triggering action causes a trigger to run. There are three types of trigger that SQL Server supports: Data Manipulation Language (DML) triggers, Data Definition Language (DDL) triggers, and logon triggers.



DML triggers

DML is the subset of Transact-SQL that includes INSERT, UPDATE, and DELETE statements. A DML trigger runs in response to INSERT, UPDATE, or DELETE actions on a specific table. When you create a DML trigger, you include the name of the table that the trigger is associated with, and whether the trigger will run after the specified DML action or actions complete or instead of the action or actions.

- AFTER triggers allow the triggering action to complete before running. For example, you might create a trigger that updates a table in an inventory database in response to the orders that customers place in an order processing database. When a customer orders a particular product, the trigger automatically updates the stock level for that product in the inventory database. However, if the initial order entry fails to complete for any reason, the trigger would not run and the inventory database would not be updated.
- INSTEAD OF triggers run instead of the triggering statement. With an INSTEAD OF trigger, the triggering Transact-SQL statement never runs; the code in the trigger runs in its place. A key advantage of INSTEAD OF triggers is that they provide a way to run updates against views that reference more than one table. When you update a view that references only one table, there is no ambiguity about which underlying base table needs to be updated. However, when a view references multiple tables, any attempted updates against the view fail because the view represents a logical table, whereas the updates must be performed against the base tables. An INSTEAD OF trigger created on a view can include code to redirect the updates to the correct base tables.

DDL triggers

DDL is the subset of Transact-SQL that includes CREATE, ALTER, and DROP statements, as well as the GRANT, DENY, and REVOKE statements used to configure permissions. DDL triggers run in response to DDL statements that execute in a database. For example, you might create a DDL trigger to prevent the use of ALTER statements to change a table, or alternatively, to record the changes that are made to database objects. DDL triggers run after the triggering DDL event; there is no option to create INSTEAD OF DDL triggers.

Logon triggers

Logon triggers run in response to the initiation of logon sessions at the level of the SQL Server instance. These types of triggers are usually used to audit logon activity, or to manage logon activity; for example, by restricting the number of simultaneous sessions a given account is allowed to establish. Like DDL triggers, logon triggers run after the triggering DDL event, in this case the successful completion of authentication. There is no option to create INSTEAD OF logon triggers.

Creating triggers

To create triggers, you use a CREATE TRIGGER Transact-SQL statement. When creating a DML trigger, you must specify the name of table that the trigger affects. When creating a DDL trigger that affects a single database, you must include the ON DATABASE clause in the CREATE TRIGGER statement. To create a DDL trigger that affects all databases on a server, you use the ON ALL SERVER clause. To create a logon trigger, you should include the FOR LOGON clause.

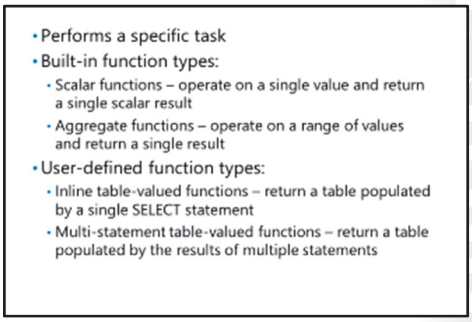
For more information about creating triggers, see *CREATE TRIGGER (Transact-SQL)* in Microsoft Docs:

CREATE TRIGGER (Transact-SQL)

http://aka.ms/m7p9qo

Functions

A function is a database object that performs a specific task, such as summing values in a column or extracting part of a character string. Functions are reusable, which helps to ensure that business logic is implemented consistently; they also hide complexity from users, making Transact-SQL statements easier to write. A function accepts input values and returns a result, either as a scalar value, which is a single, discrete value, or as a table. SQL Server includes a set of built-in functions, and you can also create user-defined functions.



Built-in functions

SQL Server built-in functions include:

- **Scalar functions**. Scalar functions operate on a single value and return a single scalar result. For example, the UPPER function accepts a character string input and returns the same string in upper case. UPPER is an example of a string function. There are many other types of scalar functions, including:
  - Date and time functions such as GETDATE, which returns the current date and time.
  - Mathematical functions such as SQUARE, which returns the squared value of the supplied value.
  - Metadata functions, which return information about database objects. For example, the DB_ID function returns the internal identification number for a given database.
- **Aggregate functions**. Aggregate functions operate on a range of values and return a single result. For example, the SUM function adds a range of data values to produce a single summed total. Other aggregate functions include MAX, which return the maximum value in a set of values, and AVG, which returns the mean value.

The following code example uses the SUM aggregate function to return the total of the values in the UnitPrice column in the Sales.LineItems table:

The SUM aggregate function

```
SELECT SUM(UnitPrice) FROM Sales.LineItems
```

User-defined functions

With user-defined functions, you can encapsulate and reuse code in a consistent way. You can create scalar user-defined functions, such as a function that concatenates separate FirstName and LastName columns to create a single result that represents the full name. You can also create table-valued functions that return a tabular data set as a result instead of a single value. Types of table-valued user-defined functions include:

- **Inline table-valued functions**. This type of function returns a table that is populated from a single SELECT statement included in the function definition. There is no need to declare the structure of the table in the function definition because the function simply uses the same structure as the base table that it selects from. An inline table-valued function is similar in functionality to a view, but unlike a view, you can include parameters. Users can pass parameter values to the function when they use it to filter the results that it returns.
- **Multistatement table-valued functions**. This type of function returns a table that is populated by the results of multiple statements in the function definition. You must explicitly declare the structure of the table that the function returns, including the columns and data types it contains. Because they can include multiple statements, multistatement table-valued functions are much more flexible in the data that they can return than views, which can only include a single SELECT statement.

Creating user-defined functions

You use a CREATE FUNCTION statement to create user-defined functions.

The following code example uses the CREATE FUNCTION statement to create an inline table-valued function that accepts a parameter value for the year, and returns only rows that have that year value in the OrderDate column:

Creating an inline table-valued function

```
CREATE FUNCTION UDF_Orders (@Year int)
RETURNS table
AS
RETURN
(SELECT * FROM Sales.Orders
WHERE YEAR(OrderDate) = @Year);
```

You can use table-valued functions by referencing them in the FROM clause in a SELECT statement, in the same way as you reference views or tables. Although you can use stored procedures that can also return data sets, unlike table-valued functions, you cannot reference stored procedures in the FROM clause of a SELECT statement.

The following code example uses the UDF_Orders function in a SELECT statement:

Using a table-valued function

```
SELECT * FROM UDF_Orders (2016)
```

Check Your Knowledge

Question	
You want to create a database object to encapsulate Transact-SQL statements in a database to ensure consistency and make it easier for users to access data without having to write complex Transact-SQL statements. The Transact-SQL in the database object that you create will reference multiple tables, must accept input parameters, and will be referenced by users in the FROM clause of SELECT statements. What type of database object should you create?	
Select the correct answer.	
	An inline user-defined table-valued function.
	A stored procedure.
	A view.
	An indexed view.
	A multistatement user-defined table-valued function.

Lab: Using SQL Server

Scenario

You work for an organization called Adventure Works, which is a retailer of bicycles and associated products. You are implementing a database called OrdersDatabase, which is an OLTP database that tracks customers, the orders that they place, and the products that are included in the orders. You need to perform the following actions in the new database:

- Implement a table, that you have already designed, to record the line items in each order. You will create the table in the Sales schema, and also add a DEFAULT constraint to the Sales.Order table to insert the current date into the OrderDate column.
- Create and test a view to provide a standardized way of accessing customer data, including the details of the orders that they place.
- Create and test a stored procedure to insert new orders into the Sales.Order table.

Objectives

After completing this lab, you will have:

- Created a table and a DEFAULT constraint.
- Created a view.
- Created a stored procedure.

Estimated Time: 60 minutes

Ensure that the 10985C-MIA-DC and 10985C-MIA-SQL virtual machines are both running, and then log on to 10985C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.

Exercise 1: Creating a Table and Constraints

Scenario

The OrdersDatabase needs to store the details of each order, including the line items, quantity, and unit price. In this exercise, you will create a table called Sales.LineItems to store this information, and implement the required constraints to ensure referential integrity. You will also create a DEFAULT constraint on the OrderDate column in the Sales.Orders table to add the current date as a default value.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment

2. Create the Sales.LineItems Table

3. Create and Test a DEFAULT Constraint

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 10985C-MIA-DC and 10985C-MIA-SQL virtual machines are both running, and then log on to 10985C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. Browse to the **D:\Labfiles\Lab06\Starter** folder, and then run **Setup.cmd** as Administrator.

▶ Task 2: Create the Sales.LineItems Table

Note: You can find the completed Transact-SQL statements for this exercise in the **Create Line Items Table.sql** file in the **D:\Labfiles\Lab06\Solution** folder.

1. Open **SQL Server Management Studio** and connect to the MIA-SQL Database Engine instance by using Windows Authentication.
2. In a new query window, type and execute a Transact-SQL statement to create a table called **LineItems** in the **Sales** schema in the **OrdersDatabase** database. Include the following columns and constraints. None of the columns should allow NULLs:
   - **OrderID** with the data type **int**.
   - **ProductID** with the data type **int**.
   - **UnitPrice** with the data type **money**.
   - **Quantity** with the data type **smallint**.
   - A primary key constraint called **PK_LineItems** on the **OrderID** and **ProductID** columns.
   - A foreign key constraint called **FK_LineItems_Orders** between the **OrderID** column in the **Sales.LineItems** table and the **OrderID** column in the **Sales.Orders** table.
3. Browse to the **D:\Labfiles\Lab06\Starter** folder, and then run **Populate LineItems.cmd**, as Administrator. This script adds rows to the newly created **Sales.LineItems** table.

▶ Task 3: Create and Test a DEFAULT Constraint

1. Type and execute a Transact-SQL statement to create a DEFAULT constraint called **DF_OrderDate** that adds the current date as the default value for the **Orderdate** column of the **Sales.Orders** table. Use the **GETDATE()** function to retrieve the current date.
2. Test the DEFAULT constraint by executing an INSERT statement to add a row to the **Sales.Orders** table using the following values:
   - **OrderID** =**110**
   - **CustomerID** = **3**
   - **OrderDate** = **DEFAULT**
3. Check that the INSERT statement worked by executing a SELECT statement against the **Sales.Orders** table to return the row with the **OrderID** value of **110**. Note that the date was added by the DEFAULT constraint.
4. Close the query window, but leave SQL Server Management Studio open for the next exercise.

Results: After completing this exercise, you will have a new table named Sales.LineItems and a new DEFAULT constraint on the Sales.Orders table.

Exercise 2: Creating a View

Scenario

An application that customers use needs to display all of the orders that a customer has placed and the total value of each order. Instead of hard coding the Transact-SQL query into the application, you have decided to create a view that the application can use to access the required data.

The main tasks for this exercise are as follows:

1. Write a Transact-SQL Query to Retrieve the Required Data
2. Create the VW_CustomerOrders View

▶ Task 1: Write a Transact-SQL Query to Retrieve the Required Data

Note: The file **Create View.sql** in the **D:\Labfiles\Lab06\Solution** folder contains the Transact-SQL statements used in this exercise.

1. In SQL Server Management Studio, in a new query window, in the **OrdersDatabase** database, write a Transact-SQL query that returns the following columns. Include the appropriate JOIN statements in the FROM clause, and use the alias **O** for the **Sales.Orders** table, the alias **C** for the **Person.Customer** table, and the alias **L** for the **Sales.LineItems** table:
   - **FirstName** (in the **Person.Customers** table).
   - **Lastname** (in the **Person.Customers** table).
   - **OrderID** (in the **Sales.Orders** table).
   - **OrderDate** (in the **Sales.Orders** table).
   - **UnitPrice** (in the **Sales.LineItems** table).
   - **Quantity** (in the **Sales.LineItems** table).
2. Execute the query to ensure that it returns the required columns and rows.
3. In the SELECT list, remove the **UnitPrice** and **Quantity** columns, and replace them with the following Transact-SQL code:

```
SUM (UnitPrice * Quantity) AS [Order Total]
```

4. In the SELECT list, remove the **FirstName** and **LastName** columns, and replace them with the following Transact-SQL code:

```
(FirstName + ' ' + LastName) AS [Customer Name]
```

5. After the FROM clause and the JOIN statements, type the following Transact-SQL code:

```
GROUP BY O.OrderID, Firstname, Lastname, OrderDate;
GO
```

6. Execute the query, which should return four rows, including a column that shows the total cost for each order.

▶ Task 2: Create the VW_CustomerOrders View

1. Create a view called **VW_CustomerOrders** that includes the Transact-SQL statement that you created in the previous task.
2. Test the view by writing a Transact-SQL SELECT statement that selects all columns from the view.
3. Close the query window, but leave SQL Server Management Studio open for the next exercise.

Results: After completing this exercise, you will have a new view in the database.

Exercise 3: Creating a Stored Procedure

Scenario

To ensure that new orders are entered consistently into the Sales.Orders table, you will create a stored procedure. The procedure will use input parameters to accept values for the OrderID and CustomerID columns, but the value for the OrderDate column will be supplied by the DEFAULT constraint on that column.

The main tasks for this exercise are as follows:

1. Create the USP_InsertOrders Stored Procedure
2. Test the Stored Procedure

▶ Task 1: Create the USP_InsertOrders Stored Procedure

 **Note:** You can find the completed Transact-SQL statements for this exercise in the **Create Stored Procedure.sql** file in the **D:\Labfiles\Lab06\Solution** folder.

1. In SQL Server Management Studio, in a new query window, ensure that the database context is the **OrdersDatabase** database. Write a Transact-SQL INSERT statement that inserts a row into the **Sales.Orders** table. In the INSERT clause, in the column list, specify the following columns:
   - **OrderID**
   - **CustomerID**

 **Note:** You do not need to specify the **OrderDate** column because this column has a DEFAULT constraint that supplies its value.

2. In the VALUES list, use the following values:
   - **120**
   - **2**
3. Use the INSERT statement that you just typed to write a CREATE STORED PROCEDURE statement. Use the following specifications:
   - Name: **USP_InsertOrders**.
   - Input parameters:
     - **@OrderID** (data type **int**).
     - **@CustomerID** (data type **int**).
   - Replace the literal values (**120** and **2**) in the VALUES list with the input parameter values you added above.
   - Execute the statement to create the procedure.

▶ Task 2: Test the Stored Procedure

1. Test the procedure by executing it using the following values:
   - **@OrderID** = **150**
   - **@CustomerID** = **1**
2. Write a Transact-SQL SELECT statement to view the newly inserted rows in the **Sales.Orders** table.
3. Close SQL Server Management Studio without saving changes.

Results: After completing this exercise, you will have a new stored procedure in the database.

Question: The PRIMARY KEY column in the Sales.Orders table does not have the IDENTITY property set. If the IDENTITY property were set for this column, how would you have to modify the USP_InsertOrders stored procedure?

Question: You created a stored procedure to insert new rows into the Sales.Orders table. What else would you need to do to ensure that new orders are recorded in full, including details of the products in each order, prices, and so on?

Module Review and Takeaways

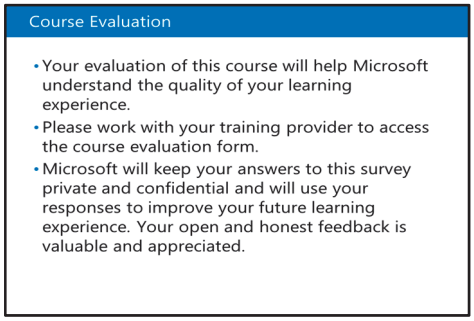
In this module, you have learned how to:

- Create tables and use data types and constraints to ensure integrity.
- Create and use views.
- Create and use stored procedures, functions, and triggers.

Review Question(s)

Question: Think of a table in a database that you are familiar with, perhaps a database in your place of work. Imagine that you need to create a stored procedure to input new rows into the table. Which input parameters would you need to include, and what data types would you assign to these parameters?

Course Evaluation



Your evaluation of this course will help Microsoft understand the quality of your learning experience.

Please work with your training provider to access the course evaluation form.

Microsoft will keep your answers to this survey private and confidential and will use your responses to improve your future learning experience. Your open and honest feedback is valuable and appreciated.

Module 1: Introduction to Databases

Lab: Exploring and Querying SQL Server Databases

Exercise 1: Exploring an OLTP Schema and a Data Warehouse Schema

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 10985C-MIA-DC and 10985C-MIA-SQL virtual machines are both running, and then log on to 10985C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. In the **D:\Labfiles\Lab01\Starter** folder, right-click **Setup.cmd**, and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**, and then wait for setup to complete.

▶ Task 2: Explore an OLTP Database Schema

1. On the taskbar, click **Microsoft SQL Server Management Studio**.
2. In the **Connect to Server** dialog box, in the **Server type** box, click **Database Engine**.
3. In the **Server name** box, type **MIA-SQL**.
4. In the **Authentication** box, click **Windows Authentication**, and then click **Connect**.
5. In Object Explorer, expand **Databases**, expand **AdventureWorks2016**, right-click **Database Diagrams**, and then click **New Database Diagram**.
6. If the **Microsoft SQL Server Management Studio** dialog box appears asking if you wish to create support objects for database diagramming, click **Yes**.
7. In the **Add Table** dialog box, press and hold down the CTRL key, click **Customer (Sales)**, click **SalesOrderDetail (Sales)**, click **SalesOrderHeader (Sales)**, click **SalesPerson (Sales)**, click **SalesTerritory (Sales)**, click **ShipMethod (Purchasing)**, click **Add**, and then click **Close**.
8. Review the tables and note the following points:
   a. The **SalesOrderHeader (Sales)** table contains the **SalesOrderID**, which is the primary key column.
   b. The **SalesOrderDetail (Sales)** table also contains a **SalesOrderID** column.
9. In the **SalesOrderDetail (Sales)** table, right-click the **SalesOrderID** column, and then click **Properties**.
10. In the **Properties** window, click the **Description** field, and then click the ellipsis button (**...**).
11. In the **Description Property** dialog box, note that the column is a primary key column, and that there is a foreign key that references the **SalesOrderID** column in the **SalesOrderHeader** column. Click **Cancel**.
12. Click the line between the **Customer (Sales)** table and the **SalesOrderHeader (Sales)** table. This line represents a foreign key relationship.
13. In the **Properties** window, click **Description**, and then click the ellipsis button (**...**).
14. In the **Description Property** dialog box, note that the foreign key references the **CustomerID** column in the **Customer (Sales)** table. Click **Cancel**.

15. On the **File** menu, click **Save Diagram_0**, in the **Choose Name** dialog box, type **Adventure Works Diagram**, and then click **OK**.
16. Close the database diagram window, leaving SQL Server Management Studio open for the next task.

▶ Task 3: Explore a Data Warehouse Schema

1. In Object Explorer, expand **Databases**, expand **AdventureWorksDW2016**, right-click **Database Diagrams**, and then click **New Database Diagram**.
2. If a dialog box appears asking if you wish to create support objects for database diagramming, click **Yes**.
3. In the **Add Table** dialog box, press and hold down the CTRL key, click **DimCustomer**, **DimDate**, **DimProduct**, **DimProductCategory**, **DimProductSubcategory**, **FactInternetSales**, click **Add**, and then click **Close**.
4. Note that the **FactInternetSales** table has foreign key relationships with the **DimCustomer**, **Dim Product**, and **DimDate** tables. Examine these relationships in the same way as you did in the previous task, noting the columns involved in the relationships.
5. On the **File** menu, click **Save Diagram_0**, in the **Choose Name** dialog box, type **Adventure Works Data Warehouse Diagram**, and then click **OK**.
6. Close the database diagram window, leaving SQL Server Management Studio open for the next exercise.

Results: After completing this exercise, you will have created a database diagram in the AdventureWorks2016 database and a database diagram in the AdventureWorksDW2016 database.

Exercise 2: Querying a Database by Using Transact-SQL

▶ Task 1: Write Transact-SQL SELECT Statements

1. In SQL Server Management Studio, in Object Explorer, right-click **AdventureWorks2016**, and then click **New Query**.
2. In the query window, type the following Transact-SQL statement, and then click **Execute**:

```
SELECT *
FROM Sales.SalesOrderHeader;
GO
```

3. Review the results in the results pane, and in the lower right corner, note the number of rows that the query returned.
4. In the query window, under the existing Transact-SQL statement, type the following Transact-SQL statement:

```
SELECT SalesOrderID, OrderDate, SalesPersonID
FROM Sales.SalesOrderHeader;
GO
```

5. Select the statement that you just typed, and then click **Execute**.
6. Review the results in the results pane, and in the lower right corner, note the number of rows that the query returned.

▶ Task 2: Write Transact-SQL SELECT Statements with a WHERE Clause

1. In the query window, under the existing Transact-SQL statements, type the following Transact-SQL statement:

```
SELECT SalesOrderID, OrderDate, SalesPersonID
FROM Sales.SalesOrderHeader
WHERE SalesPersonID = 279;
GO
```

2. Select the statement that you just typed, and then click **Execute**.
3. Review the results in the results pane, and in the lower right corner, note the number of rows that the query returned.
4. In the query window, under the existing Transact-SQL statements, type the following Transact-SQL statement:

```
SELECT SalesOrderID, OrderDate, SalesPersonID
FROM Sales.SalesOrderHeader
WHERE SalesPersonID = 279 OR SalesPersonID = 282;
GO
```

5. Select the statement that you just typed, and then click **Execute**.
6. Review the results in the results pane, and in the lower right corner, note the number of rows that the query returned.
7. In the query window, under the existing Transact-SQL statements, type the following Transact-SQL statement:

```
SELECT SalesOrderID, OrderDate
FROM Sales.SalesOrderHeader
WHERE SalesOrderID BETWEEN 57000 AND 58000;
GO
```

8. Select the statement that you just typed, and then click **Execute**.

9. Review the results in the results pane, and in the lower right corner, note the number of rows that the query returned.
10. In the query window, under the existing Transact-SQL statements, type the following Transact-SQL statement:

```
SELECT SalesOrderID, OrderDate
FROM Sales.SalesOrderHeader
WHERE SalesPersonID = 279 AND Year(OrderDate) = 2014;
GO
```

11. Select the statement that you just typed, and then click **Execute**.
12. Review the results in the results pane, and in the lower right corner, note the number of rows that the query returned.
13. Close SQL Server Management without saving changes.

Results:

After completing this exercise, you will have:

Written and executed SELECT statements to retrieve all columns and to retrieve specific columns from a table in the Adventure Works OLTP database.

Written and executed SELECT statements that include a WHERE clause to filter the rows that are returned from a table in the Adventure Works OLTP database.

Module 2: Data Modeling

Lab: Identify Components in Entity Relationship Modeling

Exercise 1: Identify Entities

▶ Task 1: Create a List of Suitable Entities

1. Examine the text of the brief.
2. On examination, extract the real nouns (those that refer to actors or actions in the application).
3. Use these real nouns to develop an initial list of suitable entities.
4. Preserve this list for later in the lab.

▶ Task 2: Add Suitable Attributes

1. Take each entity from the initial list in turn.
2. For each entity, identify suitable attributes for it. Use the text in the brief you have been given to identify those pieces of information that are to be recorded for the various entities.
3. For each attribute, develop a suitable domain for its declaration.
4. Add these attributes to the entities in the list you developed in the previous task.
5. From the list of attributes, identify the candidate keys and primary keys. These will be those attributes that (either individually or in combination) uniquely identify instances of that entity.
6. Compare your list with **initial_entities.docx** in the **D:\Labfiles\Lab02\Solution** folder.

Results: After completing this exercise, you will have an initial list of entities and attributes that model the data requirements for the brief provided. The entity definitions will include appropriate domains.

Exercise 2: Identify Relationships

▶ Task 1: Create a List of Named Relationships

1. Take each entity in the model in turn.
2. Identify the relationships this has with other entities.
3. List and name these relationships.

▶ Task 2: Draw an ERD Modeling the Entities and Relationships

1. Using the list of entities and relationships, draw an ERD showing each entity and the relationships between them.
2. Remember to name each relationship.
3. Do not forget about optionality and degree for each relationship.
4. Model these relationships by sharing keys. (Hint: use Primary Keys in one as Foreign Keys in another.)
5. Compare your model with the **initial_ER.docx** in the **D:\Labfiles\Lab02\Solution** folder.

6. What problems do you see for modeling relationships with the model in its current form? (Hint: think about the degree of some of the relationships.)

▶ Task 3: Resolve Any m:n Relationships

1. Identify any m:n relationships between entities in your model.
2. Resolve these relationships by creating an intersection entity, containing the Primary Keys of each entity involved in the original m:n relationship.
3. Compare your model with the **updated_diagram.xps** in the **D:\Labfiles\Lab02\Solution** folder.

Results: After completing this exercise, you will have an initial list of relationships between the entities that model the data requirements for the brief provided. You will have an initial ERD and have resolved any relationships that cannot be modeled (many-to-many relationships).

Exercise 3: Finalize Your Model

▶ Task 1: Develop a List of Constraints and Assumptions

1. In the data brief, constraints can be identified by the verbs "must", or "may".
2. Assumptions are usually implicitly, rather than explicitly, stated and can be identified by examination of the text.
3. List the constraints and assumptions.

▶ Task 2: Finalize the ERD

1. Update your model to include the constraints and assumptions that you identified in the previous task.
2. Compare your model with the **final_ER.docx** in the **D:\Labfiles\Lab02\Solution** folder.

Results: After completing this exercise, you will have a final data model meeting the original specification.

Module 3: Normalization
Lab: Normalizing Data

Exercise 1: Normalizing to First Normal Form

▶ Task 1: Review the Data Set

1. Ensure that the **10985C-MIA-CLI** virtual machine is running, and then log as **Student** with the password **Pa55w.rd**.
2. In the **D:\Labfiles\Lab03\Starter** folder, double-click **Normalize to 1NF.xlsx**.
3. Click the **Raw Data** tab, and then review the data set, noting any aspects of the data that cause it to violate first normal form.

▶ Task 2: Normalize the Data Set to First Normal Form

1. Decompose the table in the **Raw Data** worksheet to create a new version of the data that is in first normal form. Document your solution in the **Normalize to 1NF** worksheet. (Hint: to do this, you should decompose the raw data to create three tables.)
2. Click the **1NF Solution** tab, and then review the suggested solution, comparing it to your own.
3. Close **Normalize to 1NF.xlsx**, saving any changes if prompted.

Results: After completing this exercise, you will have normalized a data set to first normal form.

Exercise 2: Normalizing to Second Normal Form

▶ Task 1: Review the Data Set in First Normal Form

1. In the **D:\Labfiles\Lab03\Starter** folder, double-click **Normalize to 2NF.xlsx**.
2. Click the **Normalize to 2NF** tab, and then review the table to identify any features of the data that might cause the table to violate second normal form.

▶ Task 2: Normalize the Data Set to Second Normal Form

1. Decompose the table in the **Normalize to 2NF** worksheet to create a new version of the data that is in second normal form. Document your solution under the **Orders** table in the **Normalize to 2NF** worksheet.
2. Click the **2NF Solution** tab, and then review the suggested solution, comparing it to your own.
3. Close **Normalize to 2NF.xlsx**, saving any changes if prompted.

Results: After completing this exercise, you will have normalized the data set to second normal form.

Exercise 3: Normalizing to Third Normal Form

▶ Task 1: Review the Data Set in Second Normal Form

1. In the **D:\Labfiles\Lab03\Starter** folder, double-click **Normalize to 3NF.xlsx**.
2. Click the **Normalize to 3NF** tab, and then review the table to identify any features of the data that might cause the table to violate third normal form.

▶ Task 2: Normalize the Data Set to Third Normal Form

1. Decompose the table in the **Normalize to 3NF** worksheet to create a new version of the data that is in third normal form. Document your solution under the **Customers** table in the **Normalize to 3NF** worksheet.
2. Click the **3NF Solution** tab, and then review the suggested solution, comparing it to your own.
3. Close **Normalize to 3NF.xlsx**, saving any changes if prompted.

Results: After completing this exercise, you will have normalized the data set to third normal form.

Exercise 4: Denormalizing Data

▶ Task 1: Review the Data Set in Third Normal Form

1. In the **D:\Labfiles\Lab03\Starter** folder, double-click **Denormalize.xlsx**.
2. Click the **Denormalize** tab, and note that there is one row in the **Orders** table for each order, and one or more rows for each order in the **OrderDetails** table.

▶ Task 2: Denormalize the Orders Table

1. On the **Denormalize** tab, denormalize the **Orders** table to meet the criteria outlined in the scenario for this exercise. Document your solution under the **OrderDetails** table.
2. Click the **Denormalized Solution** tab, and then review the suggested solution, comparing it to your own.
3. Close **Denormalize.xlsx**, saving any changes if prompted.

Results: After completing this exercise, you will have denormalized a table.

Module 4: Relationships

Lab: Planning and Implementing Referential Integrity

Exercise 1: Planning Referential Integrity

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 10985C-MIA-DC and 10985C-MIA-SQL virtual machines are both running, and then log on to 10985C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. In the **D:\Labfiles\Lab04\Starter** folder, right-click **Setup.cmd**, click **Run as Administrator**, and then wait for setup to complete.
3. In the **User Account Control** dialog box, click **Yes**.

▶ Task 2: Identify Relationships

1. On the taskbar, click **Microsoft SQL Server Management Studio**, in the **Connect to Server** dialog box, in the **Server type** field, select **Database Engine**, in the **Server name** field, type **MIA-SQL**, in the **Authentication** field, select **Windows Authentication**, and then click **Connect**.
2. In Object Explorer, expand **Databases**, expand **OrdersDatabase**, right-click **Database Diagrams**, and then click **New Database Diagram**.
3. If a dialog box appears asking if you wish to create support objects for database diagramming, click **Yes**.
4. In the **Add Table** dialog box, press and hold down the SHIFT key, click **Products**, click **Add**, and then click **Close**.
5. Rearrange the tables so that you can see them all at the same time. You can do this by clicking the table header and dragging the table to the required place.
6. Review the database diagram and determine the relationships between the tables and what types of relationships they are (one-to-one, one-to-many, or many-to-many).

▶ Task 3: Plan Foreign Keys

1. Review the tables in the diagram again, noting the columns that are defined as primary keys, and that there are no foreign keys defined.
2. Use the business rules to decide which foreign keys you should create to enforce referential integrity. Consider which columns should have a foreign key, and which column each foreign key should reference.
3. On the **File** menu, click **Save Diagram_0**.
4. In the **Choose Name** dialog box, type **Keys**, and then click **OK**.
5. Close the database diagram pane.

Results: After completing this exercise, you will have identified the keys required to enforce referential integrity rules.

Exercise 2: Implementing Referential Integrity by Using Constraints

▶ Task 1: Implement a Foreign Key in the Orders Table

Note: You can find the completed Transact-SQL statements for this exercise in the **Create ForeignKeys.sql** file in the **D:\Labfiles\Lab04\Solution** folder.

1. In SQL Server Management Studio, click **New Query**.
2. In the query window, type the following Transact-SQL statement, and then click **Execute**:

```
USE OrdersDatabase;
GO
ALTER TABLE Orders
ADD CONSTRAINT FK_Orders_Customers FOREIGN KEY (CustomerID)
REFERENCES Customers (CustomerID);
GO
```

3. To test the foreign key, in the query window, under the existing Transact-SQL statements, type the following Transact-SQL statement, select it, and then click **Execute**:

```
INSERT INTO Orders
VALUES (105,2, GETDATE());
GO
```

4. Note that the INSERT is successful.
5. To test the foreign key again, in the query window, under the existing Transact-SQL statements, type the following Transact-SQL statement, select it, and then click **Execute**:

```
INSERT INTO Orders
VALUES (106,5, GETDATE());
GO
```

6. Note that the INSERT is prevented by the foreign key constraint because the **CustomerID** value **5** does not exist in the **Customers** table.

▶ Task 2: Implement a Foreign Key in the CustomerDetails Table

1. In the query window, under the existing Transact-SQL statements, type the following Transact-SQL statement, select it, and then click **Execute**:

```
ALTER TABLE CustomerDetails
ADD CONSTRAINT FK_CustomerDetails_Customers FOREIGN KEY (CustomerID)
REFERENCES Customers (CustomerID);
GO
```

2. To test the foreign key, in the query window, under the existing Transact-SQL statements, type the following Transact-SQL statement, select it, and then click **Execute**:

```
INSERT INTO CustomerDetails
VALUES (5,'9832 Mt. Dias Blv.', 'Chicago','97321', '08/09/1970');
GO
```

3. Note that the INSERT is prevented by the foreign key constraint because the **CustomerID** value **5** does not exist in the **Customers** table.

▶ Task 3: Implement Foreign Keys in the LineItems Table

1. In the query window, under the existing Transact-SQL statements, type the following Transact-SQL statement, select it, and then click **Execute**:

```
ALTER TABLE LineItems
ADD CONSTRAINT FK_LineItems_Orders FOREIGN KEY (OrderID)
REFERENCES Orders (OrderID);
GO
```

2. To test the foreign key, in the query window, under the existing Transact-SQL statements, type the following Transact-SQL statement, select it, and then click **Execute**:

```
INSERT INTO LineItems
VALUES (101,33,30.00,1);
GO
```

3. Note that the INSERT is successful.

4. To test the foreign key again, in the query window, under the existing Transact-SQL statements, type the following Transact-SQL statement, select it, and then click **Execute**:

```
INSERT INTO LineItems
VALUES (106,44,30.00,1);
GO
```

5. Note that the INSERT is prevented because the **OrderID** value **106** does not exist in the **Orders** table.

6. In the query window under the existing Transact-SQL statements, type the following Transact-SQL statement, select it, and then click **Execute**:

```
ALTER TABLE LineItems
ADD CONSTRAINT FK_LineItems_Products FOREIGN KEY (ProductID)
REFERENCES Products (ProductID);
GO
```

7. To test the foreign key, in the query window, under the existing Transact-SQL statements, type the following Transact-SQL statement, select it, and then click **Execute**:

```
INSERT INTO LineItems
VALUES (102,22,15.00,1);
GO
```

8. Note that the INSERT is successful.

9. To test the foreign key again, in the query window under the existing Transact-SQL statements, type the following Transact-SQL statement, select it, and then click **Execute**:

```
INSERT INTO LineItems
VALUES (104,66,30.00,1);
GO
```

10. Note that the INSERT is prevented because the **ProductID** value **66** does not exist in the **Products** table.

11. On the **File** menu, click **Save SQLQuery1.sql**.

12. In the **Save File As** dialog box, browse to **D:\Labfiles\Lab04\Starter**, in the **File name** box, type **CreateForeignKeys**, and then click **Save**.

13. In Object Explorer, under **OrdersDatabase**, expand **Database Diagrams**, and then double-click **dbo.Keys**. Review the database diagram and note that it includes the foreign key relationships that you created in this exercise.

Results: After completing this exercise, you will have implemented referential integrity in the OrdersDatabase database using constraints.

Exercise 3: Implementing Cascading Referential Integrity

▶ Task 1: Configure Cascading Referential Integrity in the CustomerDetails Table

Note: You can find the completed Transact-SQL statements for this exercise in the **ImplementCascadingIntegrity.sql** file in the **D:\Labfiles\Lab04\Solution** folder.

1. In SQL Server Management Studio, click **New Query**, in the query window, type the following Transact-SQL statement, and then click **Execute**:

```
USE OrdersDatabase;
GO
DELETE Customers
WHERE CustomerID = 2;
GO
```

2. Review the results, and note that the DELETE failed because of a foreign key constraint.
3. In the query window, under the existing Transact-SQL statements, type the following Transact-SQL statement, select it, and then click **Execute**:

```
ALTER TABLE CustomerDetails
DROP CONSTRAINT FK_CustomerDetails_Customers;
GO
ALTER TABLE CustomerDetails
ADD CONSTRAINT FK_CustomerDetails_Customers FOREIGN KEY (CustomerID)
REFERENCES Customers (CustomerID) ON DELETE CASCADE;
GO
```

4. Select the Transact-SQL statement you typed in step 1, and then click **Execute**.
5. Review the results, and note that the DELETE failed because of the foreign key constraint on the **Orders** table.

▶ Task 2: Configure SET DEFAULT in the Orders Table

1. In the query window, under the existing Transact-SQL statements, type the following Transact-SQL statement, which creates a default constraint on the **CustomerID** column in **Orders** with a value of **0**. Select the statement, and then click **Execute**:

```
ALTER TABLE Orders
ADD CONSTRAINT DEF_CustomerID
DEFAULT 0 FOR CustomerID;
GO
```

2. Under the existing Transact-SQL statements, type the following Transact-SQL statement, which adds a row to the **Customers** table with a **CustomerID** value of **0**. Select the statement, and then click **Execute**:

```
INSERT INTO Customers
VALUES(0, 'Not Applicable', 'Not Applicable');
GO
```

3. Under the existing Transact-SQL statements, type the following Transact-SQL statement, select it, and then click **Execute**:

```
ALTER TABLE Orders
DROP CONSTRAINT FK_Orders_Customers;
GO
ALTER TABLE Orders
ADD CONSTRAINT FK_Orders_Customers FOREIGN KEY (CustomerID)
REFERENCES Customers (CustomerID) ON DELETE SET DEFAULT;
GO
```

4. Select the Transact-SQL statement you typed in step 1 of the previous task, and then click **Execute**.
5. Review the results, and note that the DELETE succeeded.
6. Under the existing Transact-SQL statements, type the following Transact-SQL statement, select it, and then click **Execute**:

```
SELECT * FROM Orders;
GO
```

7. Review the results, and note that **OrderID** 101 has a **CustomerID** value of 0.
8. Under the existing Transact-SQL statements, type the following Transact-SQL statement, select it, and then click **Execute**:

```
SELECT * FROM CustomerDetails;
GO
```

9. Review the results, and note that **CustomerID** 2 has been removed.
10. On the **File** menu, click **Save SQLQuery2.sql**.
11. In the **Save File As** dialog box, browse to **D:\Labfiles\Lab04\Starter**, in the **File name** box, type **ImplementCascadingIntegrity**, and then click **Save**.
12. Close SQL Server Management Studio.

Results: After completing this exercise, you will have implemented cascading referential integrity.

Module 5: Performance
Lab: Performance Issues

Exercise 1: Designing for Efficient Queries

▶ Task 1: Plan Indexes for Querying

1. Ensure that the 10985C-MIA-DC and 10985C-MIA-SQL virtual machines are both running, and then log on to 10985C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. In the **D:\Labfiles\Lab05\Starter** folder, double-click **DatabaseQueryDesign.docx**.
3. Plan which clustered and nonclustered indexes you would add to this design, based on the design of the tables and the common queries.
4. In the **D:\Labfiles\Lab05\Solution** folder, double-click **PossibleIndexStrategy.docx**.
5. Review and discuss any differences between your strategy and the proposed index strategy in the document.

▶ Task 2: Plan Indexes for Concurrency

1. If the database in the design has handled many transactions, in addition to the listed queries, note the effects this could have on the transactions.
2. Note the effects this could have on the listed queries.
3. Note the changes that you might make to the index strategy to support transactions.

Results: After completing this exercise, you will have created an index plan.

Module 6: Database Objects

Lab: Using SQL Server

Exercise 1: Creating a Table and Constraints

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 10985C-MIA-DC and 10985C-MIA-SQL virtual machines are both running, and then log on to 10985C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. In the **D:\Labfiles\Lab06\Starter** folder, right-click **Setup.cmd**, and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**, and then wait for setup to complete.

▶ Task 2: Create the Sales.LineItems Table

Note: You can find the completed Transact-SQL statements for this exercise in the **Create Line Items Table.sql** file in the **D:\Labfiles\Lab06\Solution** folder.

1. On the taskbar, click **Microsoft SQL Server Management Studio**, in the **Connect to Server** dialog box, in the **Server type** field, select **Database Engine**, in the **Server name** field, type **MIA-SQL**, in the **Authentication** field, select **Windows Authentication**, and then click **Connect**.
2. Click **New Query**, and then in the new query window, type the following Transact-SQL statement, and then click **Execute**:

```
USE OrdersDatabase;
GO
CREATE TABLE Sales.LineItems
(OrderID INT NOT NULL
,ProductID INT NOT NULL
,UnitPrice MONEY NOT NULL
,Quantity SMALLINT NOT NULL
CONSTRAINT PK_LineItems PRIMARY KEY (OrderID,ProductID)
CONSTRAINT FK_LineItems_Orders FOREIGN KEY (OrderID)
REFERENCES Sales.Orders (OrderID));
GO
```

3. In Object Explorer, expand **Databases**, expand **OrdersDatabase**, expand **Tables**, and verify that the **Sales.LineItems** table appears in the list of tables.
4. On the taskbar, click **File Explorer**, browse to the **D:\Labfiles\Lab06\Starter** folder, right-click **Populate LineItems.cmd**, and then click **Run as administrator**.
5. In the **User Account Control** dialog box, click **Yes**, and then wait for setup to complete. This script adds rows to the newly created **Sales.LineItems** table.

▶ Task 3: Create and Test a DEFAULT Constraint

1. In SQL Server Management Studio, in the query window, under the existing Transact-SQL statement, type the following Transact-SQL statement, select it, and then click **Execute**:

```
ALTER TABLE Sales.Orders
ADD CONSTRAINT DF_OrderDate DEFAULT GETDATE() FOR OrderDate;
GO
```

2. Under the existing Transact-SQL statement, type the following Transact-SQL statement, select it, and then click **Execute**:

```
INSERT INTO Sales.Orders
VALUES (110, 3, DEFAULT);
GO
SELECT * FROM Sales.Orders
WHERE OrderID = 110;
GO
```

3. Review the results, and note that the value in the **OrderDate** column was added by the DEFAULT constraint.
4. Close the **SQLQuery1.sql** window without saving changes.
5. Leave SQL Server Management Studio open for the next exercise.

Results: After completing this exercise, you will have a new table named Sales.LineItems and a new DEFAULT constraint on the Sales.Orders table.

Exercise 2: Creating a View

▶ Task 1: Write a Transact-SQL Query to Retrieve the Required Data

Note: The file **Create View.sql** in the **D:\Labfiles\Lab06\Solution** folder contains the Transact-SQL statements used in this exercise.

1. In SQL Server Management Studio, click **New Query**. In the query window, type the following Transact-SQL statement, and then click **Execute**:

```
USE OrdersDatabase;
GO
SELECT (FirstName + ' ' + LastName) AS [Customer Name]
,O.OrderID, OrderDate
,SUM(UnitPrice*Quantity) AS [Order Total]
FROM Person.Customers as C
JOIN Sales.Orders as O
ON C.CustomerID = O.CustomerID
JOIN Sales.LineItems as L
ON O.OrderID = L.OrderID
GROUP BY O.OrderID, Firstname, Lastname, OrderDate;
GO
```

2. Review the results, noting that the query returns one row for each of the orders in the **Sales.Orders** table, along with a single total cost for each order, which is calculated in the query by multiplying the **UnitPrice** column by the **Quantity** column in the **Sales.LineItems** table. Note also that the **Customer Name** column is created by concatenating the **FirstName** and **LastName** columns in the **Person.Customers** table.

▶ Task 2: Create the VW_CustomerOrders View

1. In SQL Server Management Studio, in the query window, edit the Transact-SQL statement that you typed in the previous task by typing the following Transact-SQL code directly above the SELECT line:

```
CREATE VIEW VW_CustomerOrders
AS
```

2. Click **Execute** to create the view.
3. In the query window, under the existing Transact-SQL statement, type the following Transact-SQL statement, select it, and then click **Execute**:

```
SELECT * FROM VW_CustomerOrders;
GO
```

4. Review the results, noting that they are the same as the results that the SELECT query returned in the previous exercise.
5. Close the **SQLQuery2.sql** window without saving changes.
6. Leave SQL Server Management Studio open for the next exercise.

Results: After completing this exercise, you will have a new view in the database.

Exercise 3: Creating a Stored Procedure

▶ Task 1: Create the USP_InsertOrders Stored Procedure

Note: You can find the completed Transact-SQL statements for this exercise in the **Create Stored Procedure.sql** file in the **D:\Labfiles\Lab06\Solution** folder.

- In SQL Server Management Studio, click **New Query**, type the following Transact-SQL statement, and then click **Execute**:

```
USE OrdersDatabase;
GO
CREATE PROCEDURE USP_InsertOrders
@OrderID int, @CustomerID int
AS
INSERT INTO Sales.Orders (OrderID, CustomerID)
VALUES
(@OrderID, @CustomerID);
GO
```

▶ Task 2: Test the Stored Procedure

1. In the query window, under the existing Transact-SQL statement, type the following Transact-SQL statement, select it, and then click **Execute**:

```
EXEC USP_InsertOrders 150,2;
GO
SELECT * FROM Sales.Orders
WHERE OrderID = 150
```

2. Review the results set, noting that it includes the row that the stored procedure added, and that the **OrderDate** value was added by the DEFAULT constraint.
3. Close SQL Server Management Studio without saving changes.

Results: After completing this exercise, you will have a new stored procedure in the database.