Course Overview

Hi everyone, my name is Gerald Britton, Welcome to my course, Designing Tables and Views in SQL Server. I'm a Senior Solutions Designer specializing in SQL Server database technologies. Did you know that at any given moment, there are more than 20, 000 openings for SQL Server developers and that many of those jobs pay over $100, 000 per year? And that's just in North America. Does that sound like a field you'd like to get into. In this course, we're going to work on fundamental skills needed by everyone who wants to get ready for prime time when it comes to database design and development. We'll do it all in the context of a growing business struggling with managing customer orders. Some of the major topics that we will cover include matching datatypes to usage with an eye on storage utilization, normalizing tables to eliminate redundancy and improve integrity, leveraging the power of data constraints to keep data clean, creating higher-level objects called views to improve reusability, and building high-performance views for growing data. By the end of this course, you'll know how to design the tables and views for a new database, as well as spot problems and improvement opportunities in the databases you may already be using. Before beginning this course, you should be familiar with writing simple database queries using SQL, the structured query language. When you are done, you should feel comfortable diving into advanced database design with courses on indexing, stored procedures, functions, and triggers, and performance analysis and troubleshooting. I hope you'll join me on this journey to learn database design with the Designing Tables and Views in SQL Server course at Pluralsight.

Introducing Tables and Views

Introduction

Hello. Welcome to the course, Designing and Implementing Tables and Views in SQL Server. My name is Gerald Britton. I'm a Senior IT Solutions Designer, SQL Server Specialist, and Pluralsight Author. This course is packed with essential information to get you started designing tables and views, and not just for SQL Server, although that will be our environment. There are differences between SQL dialects to be sure, but most share the same core concepts. In this introductory module, I'll take a few moments to review the background of relational table design and how that has developed since first introduced. I'll also kick the tires of Azure Data Studio, the IDE I'll be using throughout this course. Then, I'll open up a common business problem that we will use throughout the course to develop a good set of tables and views that follow industry best practices. Finally, I'll give you a brief overview of the modules that follow so that you can easily find your way.

Relational Database Foundations

Everything we do today that depends on relational databases depends on this man: Edgar F. Codd. He was a British computer scientist who worked for IBM. Back in 1970, he wrote a paper that became the foundation for relational databases entitled A Relational Model of Data for Large Shared Data Banks. At the time, hierarchical data storage was common, as were various types of flat files. Codd saw the complexities of that approach and came up with a simple, but at the time, revolutionary idea: Store data in tables and define relationships between them for processing. Actually, Codd used the word relations instead of tables. A relation is a set comprised of tuples holding attributes. Today, we commonly speak of tables having rows and columns, like a spreadsheet, but with stricter requirements. For one thing, since a Codd relation, today's table, is a set, that means there can be no duplicate tuples, today's rows. That constraint is usually enforced by identifying a primary key for each row. And by definition, primary keys must be unique. Although that rule is sometimes relaxed for

certain purposes, it is still what we aim for as database designers. This course is all about designing tables and their relationships to solve business problems.

Azure Data Studio Demo

For the demos in this course, I'll be using Azure Data Studio, a free, cross-platform IDE for developing SQL Server applications. At the time of writing, it could be found at the bit.ly link shown. If by chance the page has moved, your favorite search engine should be able to find it quickly. I suggest you take a moment and install it on your own machine before continuing. I also suggest that you install either SQL Express or SQL Developer, both accessible from the bit.ly link shown and easy to find otherwise. I'll be using LocalDB, which comes with SQL Express and is quite lightweight. Alternatively, you can download Docker containers for SQL Developer and SQL Express at the link shown, or just search for them on Docker Hub. I'll launch Azure Data Studio so you can see its major components. The default screen is quite spartan. There are a few icons down the left-hand side and a little help section in the middle showing a few hotkeys. The Ctrl+G hotkey corresponds with the topmost icon, Show Servers. Opening that shows no server names listed, so let's add one. I'll click on the Add button for a new connection and a dialog opens up. I mentioned I'll be using LocalDB, so I'll just paste that connection information in here and hit the Connect button. The connection is successful and I can see the database. Also, Data Studio has opened the Server Dashboard, which shows some basic information and reveals that I'm running the 64-bit Express edition of SQL Server on Windows 10. You will, no doubt, see slightly different details. You should take your time and explore the tabs and actions here. For now, I'll just click on New Query, which brings up a second tab with an edit window. I'll just write a simple query here to show that it works and run it. There. Down below, the Results panel shows, well, the result set. And there's also a Messages panel at the bottom where some statistics and any error messages would appear. The third icon down opens the Explorer view. Right now it says No Folder Opened, so I'll open one. I'll just use this one right here. It gives me the option now to save my query, so I'll do that. So I've saved my little query, and you can see that it

now appears in the Explorer view. There are many more options and capabilities and a whole range of extensions available. I'll let you discover those on your own. We have enough now to get going.

Business Problem: Bob's Shoes

Throughout this course, I'll be working on a simple business problem. Meet Bob. Bob has a passion for shoes, and that passion has led him to build a business called Bob's Shoes. To make his shoes, which come in all shapes and sizes, he has built a factory. The factory churns out shoes day and night to meet the demands of his loyal and growing customer base. But Bob has a problem. He's been taking orders by hand, and his business has grown enough that handwritten orders are not up to the task. Orders get lost. Some get doubled. Sometimes the order sheets are illegible. Clearly, Bob needs a better system. We're going to help him build part of that system by designing database tables and views to keep track of things.

Course Overview

In the upcoming modules of this course, I'll start by looking at the basics of table design, show you how to choose appropriate data types, and what to look for in primary keys. Then, I'll introduce the concept of normalization, why we need it, how it works, and the various levels of normalization available. Once normalized, I'll dig into the types of constraints we can define in our tables to use SQL Server to guard the integrity of our data. Along with tables, views are an important concept to understand. I'll show you how views can be used on top of the tables in a database for frequently-used queries. For many purposes, views can be treated just like tables, and that includes updating. There are some requirements though for updateable views, so I'll cover that. Partitioned views provide a way for the high-performance querying of multiple similar tables and are an important tool for accessing growing databases. I'll finish up the section on views with a discussion of indexed views, which have some performance advantages and a property that might surprise you. Now, let's get started by learning how to design and implement tables, keeping Bob's shoe business in mind.

Designing and Implementing Tables

Introduction

Hello. Welcome back to the course, Designing and Implementing Tables and Views in SQL Server. My name is Gerald Britton. In this module, I'll be taking you through the basics of how to create tables in SQL Server. You'll learn about the data definition language or DDL used for creating, changing, and deleting tables, and how to match the available data types to business requirements, and how to identify primary keys. The demos along the way will address the problems of Bob's shoe business that you looked at in the introduction. Let's get started. Before you create a table, you need to know a few things. First off, you need the name of the database that will hold the table. If an RDBMS were a country, a database would be a city. Next, you need the name of a schema within the database where the new table will live. A schema is like a namespace as found in many programming languages. Sticking with the country analogy, a schema name would be a district or borough within a city. Note that even if you don't specify a schema name, your table will still live in a special schema, the database owner schema, dbo for short. At the risk of sounding obvious, you need a name for your new table. A table is not a table without columns, and every column needs a name. Every column also must have a data type. There are many available to fit various needs. Finally, you need to think about any required constraints on the data that a column can hold. You'll learn more about constraints later in this course. In this module, we'll look at two of them. A PRIMARY KEY constraint is used to define the primary key which must be unique over all the rows the table will hold. The nullability constraint indicates whether it is acceptable for a column to ever have null value for some row. Since there are at least four names required, let's see what SQL Server names can be.

Creating Identifiers

The names used in creating a table and, indeed, all names of things in SQL Server should follow the four simple rules for regular identifiers. First, they must begin with a letter, an underscore,

an at sign, or a number sign. The letters are defined by the Unicode standard and include the alphabets of most languages. Please note that the at sign and the number sign have special meanings in Transact-SQL. Avoid these at the start of database object names. Second, after the first letter, identifiers may contain letters, numbers, and the characters, the at sign, the dollar sign, the number sign, or the underscore. Third, regular identifiers must not be a Transact-SQL reserved word. This means names like database or table or schema are excluded, as are other names such as key, value, date, and time. Fourth, regular identifiers may not contain embedded spaces or special characters. In a way, rule four is redundant since rule two covers these exclusions. Practically, it is good to have this spelled out since attempts to break this rule can lead to odd errors in code that does not work as expected. Wait, did I say there were four rules. There're also a couple of rules for exceptions. A special kind of identifier called a delimited identifier that breaks any of the first four rules must be enclosed in brackets like the example shown. And, finally, though not often stated, an identifier cannot be longer than 128 characters. Practically, that's quite enough. If you can't effectively name an object in 128 characters, chances are you have bigger problems to solve.

Using Naming Conventions and ... The Mice!

It's a great idea to establish a naming convention for a database and even an entire database system. There are several popular schemes in use. First, two things to avoid. Do not use the at sign as the first character for a name since that is used to denote variable names in Transact-SQL. Also, do not use the number sign as the first character since that is used for naming objects in tempdb. Here are a few of the most popular styles. Use a consistent style for naming objects. CamelCase where identifiers use a capital letter for all words in a name, underscore separated where the words are separated by, well, underscores, or maybe a hybrid approach, which is basically like CamelCase but using underscores to eliminate ambiguity, and, finally, at the risk of being too repetitive, do not use delimited identifiers like the one shown here. Now to get our feet wet, let's define a simple table. Suppose you have a database called Rodents where all things rodent-ish live. I start off by entering

that database context with the USE statement. Then I'll create the new table in the Mice schema. The table is called TheQuestion. The new table has just one column called TheAnswer, which has a data type of integer and the NOT NULL constraint, which means that this column must always have some value. That's it. A simple, complete table definition. To finish up this example, I'll populate the table with a proposed answer. On a side note, TheQuestion is not really a very good table name. On the other hand, that's the problem. The Mice don't know TheQuestion. For more information, see the Hitchhiker's Guide to the Galaxy by the late British author, Douglas Adams.

Using Character Data Types

Now let's think about the kinds of things Bob will need to capture about each shoe order. There will be textual data such as the customer's name, address, shoe style, and maybe salesman's name. Some of the data will be integers like the quantity ordered and maybe an order ID. Other data will need a few decimal places. I'm thinking of the shoe size and pricing and discount information. Then there will be a need for dates like the order date, the requested date, and the actual delivery dates. SQL Server has data types to meet these requirements. In fact, there are multiple choices for each of these four types. Let's look at the possibilities to see if we can find a good fit. There are four data types available for storing character strings in SQL Server. Char(n) is for fixed length non-Unicode data and specifies the length from 1 to 8000. This data type always takes n bytes per row. Use it if most of your columns will have the same or mostly the same length or if the length is less than 3. Doing so will ensure less wasted space when compared to the next type. Varchar event is for variable length non-Unicode data. As the name implies, the actual space used is variable up to a maximum length event, which can also range from 1 to 8000. This is an efficient data type for highly variable data only using the actual data length per row. Names and addresses usually fall into this category. varchar(max) can hold up to 2 GB per column. However, this data type can use more disk space leading to extra I/O. Use it sparingly. Nchar(n) is for a fixed length Unicode data, and n can range from 1 to 4000. The storage size used is two times n bytes. Use for uniform length or short length

character data that requires Unicode. Most systems that store text in multiple languages need Unicode for example. Nvarchar(n) is similar to varchar(n) but for Unicode data. N can again range from 1 to 4000. Use it as you would varchar(n) but for circumstances that require Unicode. Nvarchar(max) is similar to varchar(max) and can hold up to 1 GB of characters, since with Unicode data, 2 bytes are used for each character. So, what should we use for Bob's orders? If we assume that Bob wants to expand internationally, nvarchar is the best type for customer information. The length depends on the business expectations. Customer names and addresses can be long. A recent search found a full name 225 characters long and an address almost 200 characters long. But plan for the data you actually expect, and get your customer to agree with the maximum data lengths. Also setting a reasonable length acts as a constraint. SQL Server will reject data that is too long, or at least warn you that truncation may occur. This can help you identify bad data before it gets into your system. Product names and SKUs are probably better served by the varchar type, since they will consume less storage. On the other hand, SKUs may be of a fixed length format so char could be an even better choice in that case.

Using Integer Types

There are also four data types available for storing integer data in SQL Server. Tinyint, which is an unsigned integer with values from 0 to 255. They are handy when you know you have a small set of integer values to store. Only 1 byte is used. Smallint, which is a 2-byte signed integer with a value of about -32 K to +32 K. Int, a 4-byte signed integer with a value range of approximately plus or minus 2.1 billion in the English system. And bigint, an 8-byte signed integer with a value of approximately plus or minus 9.2 sextillion in the English system. Now Bob's orders have some obvious integer types, including quantity and possibly an order ID. Let's use a small int for the quantity of each order item and an int for the order ID. Later on in this course, we'll come back to what to do about negative order quantities.

Using Decimal Types

There are four types, well, actually three, that can be used for numbers with fixed precision and scale. Decimal and numeric are synonyms and can be used interchangeably. The precision, the lowercase p above, is the total number of digits that will be stored ignoring the decimal point. The scale, the lowercase s, is the number of digits that will be stored to the right of the decimal point. Precision and skill are optional, which is why they are shown in brackets. If not specified, the default position is 18 and the default scale is 0. When the scale is 0, this type effectively stores integers and can store very large one. whereas the maximum value of a bigint can hold 19 decimal digits, a decimal or numeric type can hold twice as many, up to 38 digits, and the storage length depends on the precision and can range from 5 to 17 bytes. The money type is a special type with exactly 4 decimal places and the range shown here. Eight bytes are used to store these values. They are also unique in that you can use most common currency symbols when storing them or comparing them, although SQL Server stores the numeric value only, not the currency symbol itself. The smallmoney type is like the money type but with a precision of only 10 and a storage size of 4 bytes. Note that the money types are unique to SQL Server. Some recommend against using them for just that reason. So what should we use for Bob's order data? For shoe sizes, 2 digits before the decimal point and 2 after should do. A numeric type with precision 4 and scale 2 accommodates those values including half sizes. But let's think about this some more. Shoe sizes will never be part of a mathematical operation, so it probably makes more sense just to store them as varchar. Then shoe widths can be stored in the same column where you might think they belong. But what about prices? Custom shoes can be expensive, but if we use a numeric type with precision 7 and scale 2, we can handle very expensive shoes, indeed, should we need to. Discount percentages shouldn't need more than 2 digits before the decimal point or 2 after. Let's use the numeric type with precision 4 and scale 2 for this one.

Using Date and Time Types

SQL Server has a total of six data types for storing dates and times. Part of that richness is due to history. Part is due to differing targets. Let's talk about the easy ones first. A date is, well, a date. It can store dates in the range shown here and only takes 3 bytes. Sometimes you might see date stored as integers, which are 4 bytes. When you do, remind yourself that the data type is actually more storage efficient. The time type stores times of day. The optional parameter specifies the number of fractional seconds to be stored. The default is 7, which stores times as precise as 100 nano seconds. Regardless of the fractional seconds, this type takes 5 bytes to store. Datetime is an older data type in SQL Server with definite limits. The date part cannot hold any date before January 1, 1753, and the time part is in 1/1000 of a second, but due to rounding is not stored exactly at that precision. The system stored values are always rounded to increments of .0, .003, or .007 seconds. It requires 8 bytes of storage. Smalldatetime has a more limited date range and does not store fractional seconds, but only uses 4 bytes of storage. The datetime2 type is like the date type combined with the time type. The ranges of dates and times are the same. And as with the time type, you can specify the precision of fractional seconds with a default of 7 digits or 100 nanoseconds. And depending on the precision, 6, 7, or 8 bytes are required to store this type. Finally, datetimeoffset is a type that combines datetime2 with a time zone. The ranges of dates and times are the same as for the datetime2 type, and this type always takes 10 bytes to store. So which type is a good choice for the order system? Thinking about it, Bob will never need high precision time to track the orders. Also, the times are captured in the local time zone of the factory. Datetime2 with a precision of 0 can handle this easily and only takes 6 bytes to store. Let's go with that. So just when you are ready to create the order table, you find out that you need to flag some orders as expedited. To do that, you can use the bit type, custom-made for flags and switches.

Demo 1: Creating the Order Tracking Table

In this demo, I'll create a database for Bob's shoe orders and a table to track them. If you have downloaded the demo files and have a SQL Server instance to work with, you can run this code as I do. Now I'm connected to my LocalDB, which has no user databases at the moment. The first thing to do is create one. In Explorer, I'll use this CreateDatabase script to do that. If I click the Run button, the script succeeds, and now I have a user database called BobsShoes. You may wonder what magic happens when you do that. For one thing, SQL Server records the database information in its internal tables. You could learn a lot from this query on the sys.databases system view. I won't go into all the details here but encourage you to read through the Microsoft documentation on it. You'll easily find it with your favorite search engine. Another thing creating a database does is put files in the file system. The system stored procedure sp_helpfile will display them. You can see that two files were created, one for data and one for the log. Also note the filegroup names. PRIMARY is the default filegroup created If you don't specify one explicitly, and I didn't. I'll also go ahead and create an order schema, which is where I'll put my tables. Using schemas for user tables is a good practice. Apart from the convenience of having the extra namespaces schemas provide, they're also great for managing security and granting and restricting access. Now you can create multiple filegroups and put multiple files in each one. The best practice is to put the data in log files on separate drives. The reason is simple. Separating them reduces contention on any one drive and spreads the load around. I can easily add new filegroups to my database. These commands will do that and set up separate files for data and logs. Note that there are actually three names at play here. The first is the name of the filegroup. The second is the logical name of the file as SQL Server refers to it. Think of it as a nickname for the file. And, lastly, the physical name of the file as it exists on the file system. Note the difference between the file types; .mdf is used for data files, and .ldf is used for log files. And if you have multiple data files, then they would take the file type .ndf. I'll keep all three names in sync, but that's not required. Still for standard environments, keeping names in correspondence is a good practice. A filegroup can also have more than one file in it, which can also be helpful for performance

tuning in some environments. As well, I could create a separate filegroup for any indexes using similar commands. Now the sharp eye will have noticed that my new files are all on the same drive. Well, I only have one available on my laptop. In a production setting, these files should be separated. The next thing to do is create the order tracking table itself. Putting together the needed columns with the data types we want, I can construct a CREATE TABLE statement. Here's the script for that. Let me walk you through it. From the top, the USE command enters the context of the database we just created, BobsShoes. The GO command is called a batch separator. Basically commands you write are not sent to the server until a GO command is reached or the end of the input, whichever comes first. Now that I'm working in the right database, I can create the order tracking table. This begins with the command CREATE TABLE, followed by the new table name. Then in parenthesis the list of columns to be created is written. Most of these are the result of the data requirements and types I just reviewed, although there are a couple of new things. The OrderId column has a property you might not have seen before, IDENTITY. This property means that whenever a new row is inserted into this table, a new order ID is created. SQL Server tracks the current value of an IDENTITY column in its metadata for the table, and there can be only one such column per table. The values in parentheses are the seed or start value and an increment value. In this case, the start value is set to 1 as is the increment. Because these are the defaults, I can actually leave them out. However, I believe that explicit is better than implicit, so I've included them here. You'll also notice that I'm using the property NOT NULL on most of the columns and NULL on a few of them. This is actually a constraint. Columns marked as NOT NULL must always hold a value. An attempt to insert or update rows with NULL values for these columns will cause an error. And NULL, on the other hand, means that NULL values are okay. For this table, I think you can see why. The delivery date is not known until delivery, so a NULL is permitted. The TotalPrice column is defined using an expression. This is called a computed column. Also, this column as defined here is not stored in the database. It is computed every time it is selected. You can force the expression result to be stored by adding a keyword PERSISTED, which I've commented out for this example. And the data type for a completed

column is inferred from the expression. I mentioned that I wanted to use BobsData as the filegroup to hold the data for the order tracking table. Also, I've added a table option for DATA_COMPRESSION I recommend you compress most tables and have a good reason if you choose not to. While it does cost CPU cycles to compress and decompress the data, it saves on I/O and the CPU cycles needed to handle that extra I/O. The tradeoff is almost always worth it. Here I've specified PAGE level compression. ROW level compression is also available. The details, limitations, and implementation notes are beyond the scope of this course, and you should consult the official documentation for more information. I simply recommend you use it for most new work. And note that before SQL Server 2016, data compression was only available in the enterprise edition. If I run this script, the messages tell me that the table was successfully created. In the Service tab, you can see that it is. Also, you can see that SQL Server assigned a data type to the computed column, in this case that is numeric. If you've run the script just as I have, you've created your very first table, at least in this course. Still, something is missing. I promised we'd put a key constraint on the table. Let's do that in a separate script. The CreatePrimaryKey script looks like this. First, it enters the context of the target database, BobsShoes. Then using the ALTER TABLE command, a constraint is added. PK_OrderTracking_OrderId is the name of the constraint. It is defined as a PRIMARY KEY constraint on the OrderId column. I like to use a convention for constraint names where the first two characters are the type of the constraint, so PK for PRIMARY KEY, followed by the name of the table, followed by the columns in the constraint. Note that like table names, constraint and index names must be unique in the database schema. With this constraint in place, SQL Server will stop any attempt to overwrite the OrderId column with a duplicate value. This also ensures that the table is a proper relation since at least one column is unique for every row in the table. Sometimes there's a little confusion around key constraints and indexes. Let's fix that right here. A key constraint is implemented by SQL Server by creating a matching or backing index. This makes checking the constraint efficient. Also, I could've put an ordinary index on the same column in this table, but it would not have been a constraint. Constraints and indexes are not the same thing, but they can

support each other. A backing index is always built to support a key constraint. There are more options that can be specified for columns than I've shown here. Some I'll cover later in the modules on normalization and constraints. One, I think worth covering at this point, is collation. Let's look at that.

Using Collations

Quoting from the official SQL Server documentation, a collation specifies the bit patterns that represent each character in the data set. Collations also determine the rules that sort and compare data. You can specify collations at the instance, database, column, and expression level. Let's dig in a little deeper. SQL Server stores character data as either Unicode or non-Unicode. These map to the data types nchar and varchar and char/varchar respectively. Now all character types have some collation. If not specified at the column level, it uses the database collation. If not specified at the database level, it is inherited from the instance, and the instance collation is defined during setup. You can also specify collation on an expression, for example, when doing a comparison. Collations provide sorting rules, case sensitivity, and accent sensitivity properties. For non-Unicode types like char and varchar, collation also dictates the code page to be used and the set of characters available. Now let's go back to Data Studio and explore the collations in the BobsShoes database.

Demo 2: Using Collations

Here I have a few queries that pull data from the system to show us more about the collations in effect. The first query shows the collation configured on the instance. The SERVERPROPERTY function will return that. Running this query, you can see that the collation name is SQL_Latin1_General_CP1_CI_AS. The last two acronyms tell us that the collation is case-insensitive and accent-sensitive. The second query shows the collation in effect for the BobsShoes database. The DATABASEPROPERTYEX function gives us that. It needs the name of the database, but I cheated a bit and used the DB_NAME function to get it. Running that, you can see that it matches the instance

collation, which makes sense since the collation is inherited, and I did not override that at database creation time. The third query shows all the collations assigned to every character column in the OrderTracking table. The results show that all those columns have the same collation inherited from the database, which inherited its collation from the instance. The fourth query expands upon the abbreviations in the name, and we can see that this collation is case-insensitive, accent-sensitive, kanatype-insensitive, and width-insensitive for Unicode data, and that it is Sort Order 52 on Code

Page 1252 for non-Unicode data. Now we can interpret the name. From the right-hand side, we have AS for accent-sensitive, CI for case-insensitive, CP1 for Code Page 1, which is really Code Page 1252 of the default Windows Code Page. The rest of the name is just a short form name to make it descriptive and unique. These four queries will tell you pretty much all you need to know about what collations are in effect. The fifth query returns SQL Server collations that don't have Latin in the name. I have 46 of those. Now suppose you knew your customers were all in Sweden. Perhaps you'd want to use a different collation. You could change the collation like this. Here I change the collation to be compatible with Scandinavian countries. Note that this is Code Page 850, case-insensitive, and accent-sensitive. Of course, this can also be specified on column definitions in the CREATE TABLE statement.

Summary

In this module, you learned the basics of creating tables. You now know that you need to think about naming objects and saw a few common conventions. And after thinking about the table we need to track orders for Bob's Shoes, I reviewed the major data types available. Then you saw how to code the CREATE TABLE statement for this purpose and put a primary key on it. I then spent a little time talking about collations and how you can use them to control sorting order and comparisons for character data. Well, there are more data types available than the ones we explored in this module. Before we move on, you might want to check out the official documentation, which is easy to search for, and it could be found at this bit.ly link at the time of writing. Now, are we done with Bob's Shoes?

Hardly! In fact, as the next module will show, there are some problems with the design that can be fixed by applying normalization. See you there.

Improving Table Design Through Normalization

Introduction

Hello. Welcome back to the course, Designing and Implementing Tables and Views in SQL Server. My name is Gerald Britton. In the previous module, we created a table to hold order data for Bob's Shoes. We covered a lot. Still, I have the feeling that there are some issues with this table that you'll see in a moment. Fortunately, there are standard methods to fix these. That approach is called normalization. Here I'm not talking about the common, everyday idea of being normal, whatever that is. In the world of relational database systems, normalization is a rigorous mathematical concept that when applied correctly can reduce redundancy and improve integrity. So in this module, I'm going to be talking about normalization. Okay, you knew that. So what will we cover? First off, I'll talk about the motivation for normalization, and I'll do that by revisiting the table design from the previous module and note some problems with it. Then I'll give a formal definition of normalization and briefly talk about its history. The bulk of this module will be talking about the various normal forms available. As I describe each one, I'll change the design for Bob's Shoes to match. As you'll soon see, applying normalization can reduce duplication but also introduce a way to enforce referential integrity. I'll implement that using FOREIGN KEY constraints. Now let's revisit our table design so you can see a problem or two that I want to address.

Identifying Problems with Unnormalized Data

For the first demo, let's revisit the order tracking table from the previous module. Here is the CREATE TABLE statement I used. Let's add some data to it. This script will add three rows for a new order placed by Arthur Dent. There are three rows since Arthur has ordered three different types of shoes in different sizes. Probably Arthur has a small family, and everyone needs a new pair of shoes. Well, what could be wrong here? The first and most obvious thing is that there is a lot of repeated

data. The order date, requested delivery date, customer's name and address are all repeated. If Arthur moved to another town, we'd have to update these rows. That's not too bad, you'd think, but imagine that this table has millions of active orders, and some of them need updating. Suddenly it's not trivial anymore. But beyond the extra work required at update time, there's a bigger question. Why are we storing duplicate information in the first place? There's a good programming principle that I try to stick to no matter what the language. Don't Repeat Yourself or DRY for short. This data is definitely not dry. There is lots of repetition. Can we do anything about it? Normalization to the rescue. Here's a great definition of normalization. Normalization is the process of organizing a database to reduce redundancy and improve data integrity. Simple, succinct, and to the point. You saw how easy it was to have redundant data with the design we're working on. Normalization will help us with that. It will also help us improve data integrity since it will eliminate the possibilities for data to become inconsistent. Let's dig a little deeper into that problem and learn a little bit about the history of normalization along the way.

Setting Objectives for Normalization

E. F. Codd, the genius behind relational database theory, wrote a set of simple objectives for data normalization. The first objective is to eliminate anomalies. In the demo that follows, we'll show how anomalies could arise with the design for Bob's Shoes' order tracking system. The second objective is to reduce the need for restructuring tables as new requirements or data are added. Since such work might affect working application programs, a properly normalized database can reduce the need for application maintenance. The third objective is to make the relational model provide more information to users. This recognizes that the structure of information can be as important as the information itself. The fourth objective is to make the tables in the database less sensitive to statistics from queries, especially when those statistics are liable to change. You don't want to have to change your design because new data points with a different frequency distribution begin appearing. Now let's look at the first problem, anomalies, using the OrderTracking table from BobsShoes database.

Demo 1 - Discovering Anomalies with Unnormalized Data

A few moments ago, we added these three rows to the OrderTracking table. Now let's add a couple more. There. Now we have five rows in the table representing two orders placed by Mr. Dent. Now imagine that you get a call one day, and you find out that the Dents have moved to a new place, Magarathea. Say you only know about the first order, so you write this query. Then you check your work like this and discover to your horror that there is another order by Arthur that has his old address. You have an update anomaly. Of course, you could fix this particular one by removing the date check in the UPDATE statement. But if another order taker is entering yet another order for the Dent family and hasn't got to the memo about the address change yet, you will be in the same situation. Normalization can eliminate this problem. The next day when you come into work, you are asked to add a new customer to the database because they will be making lots of orders. What will you do? You start to write an INSERT statement and immediately see that you are stuck. there is no way to enter a new customer without an order. You have an insert anomaly. Normalization can help there too. Friday comes along, and you're asked to remove Arthur's order for his size 10D. The first order has shipped, so you run this query and check your work. Then you realize you deleted a row for an order that hasn't shipped yet. You have a delete anomaly. And as you probably guessed, normalization can help with this.

Understanding First Normal Form

The first normal form, often abbreviated 1NF, has three simple rules. First of all, there must be only one value per table cell where a cell is the intersection of a row and a column. That means that Bob's OrderTracking table can't have multiple shoe sizes per row or multiple styles. I'll show you an example of a violation in the demo. The second rule is that there must be one table per set of related data. Well, this definition raises the question, What is related data? While thinking of the OrderTracking table, let me ask you, Is a customer's address related to the number of pairs of shoes in the order? I think you can see that the answer is No. We'll need to break up the OrderTracking

table into other tables to obey this rule. The third rule is that each row must be unique. This rule is usually attained by introducing a primary key, which enforces uniqueness. A key can be simple, that is, just one column, or composite, two or more columns. A primary key must be unique and not null. Now let's look at these rules in the demo and put the order tracking system into first normal form.

Demo 2 - Transforming the Design to 1NF

Back in Data Studio, I have a query to insert a new row into the OrderTracking table. Now you may be wondering why I list every column name in the INSERT statement. This is a best practice designed to avoid errors and help the code be self-documenting. If the data inserted in the values clause does not match the types of the columns they correspond to, the INSERT operation will fail, and it should. Be sure to follow this format in your own code. This INSERT statement will run without error, but it will violate the first rule or the first normal form that there should be only one value per cell. Here I have one row that says Arthur wants one pair each of Oxfords and Wingtips. So in this row, we have two values in one cell. To fix it, I need to change the INSERT. Well, I'll comment this line and uncomment the next two. There. Rule one is not violated anymore, or is it? The customer address just has a single value, but do addresses really look like that? Usually a real-world address has at least five parts, a the street name and number, a city, a state or province, a postal code, and a country. And in some countries, even that might not be enough. British addresses are notorious for their ability to be quite complicated. Also, what happens when Bob expands his business to Mars? He'll probably need to add a planet name to the address. Now we could fix this by adding more columns to the OrderTracking table, but it's really a better idea to create a separate Customers table. While I'm at it, I'll separate out the order into two new tables and the stock information into another table. These four new tables will also satisfy rule two since related data will be in separate tables. Now here is a script to create the four new tables. There's a Customers table with the information we just discussed, a Stock table to keep track of the inventory, an Orders table to anchor all orders, and an OrderItems table for the items in each order. Note that the Orders table has a reference to the

Customers table via its CustId column, and that the OrderItems table refers to the Orders table with the OrderId column. Naming columns like this adheres to the principle of conformity. All this means is that wherever a column appears that has the same meaning, it should have the same name and type. Since CustId refers to a customer by their CustId, the name and type are the same. Also, all the tables except the Stock table have an IDENTITY column. An IDENTITY column is guaranteed to be unique for the table, which will help satisfy rule three. To fully satisfy rule three, each table has a PRIMARY KEY defined. A PRIMARY KEY is a type of constraint and simply means no duplicates. The Stock table has a composite primary key since both the SKU and size are needed for uniqueness. To create the primary key, I use the CONSTRAINT keyword, give it a name, and list the columns to be part of the composite key. In the previous module, I introduced the first type of constraint, the NULL constraint. A PRIMARY KEY is the second constraint you will need. In general, constraints are used by SQL Server to preserve data and referential integrity by prohibiting operations that will violate the constraints. Well, what about the names for the other primary keys? In the Servers tab of SQL Operation Studio, you can see them. The other tables have a system generated name. However, that name alone doesn't tell you very much. It is better to explicitly name your primary keys. So let's do that. I cheated a bit and already coded the constraint definitions, so I only have to comment the original unnamed primary keys and uncomment the named one. As with the Stock table, these are defined as constraints. Now the tables all have properly named primary keys. Also, look at the indexes. SQL Server has created an index for each of the primary keys. These indexes are marked as unique and clustered. The unique property satisfies rule three of the first normal form. The indexes created for PRIMARY KEY constraints are sometimes called backing indexes. They are not strictly required for rule three, but make it faster for SQL Server to check for duplicates since the alternative would be to read the entire table every time you had to insert a new row just to be sure there are no duplicates. The clustered property, if you haven't seen it before, says that the table data is ordered by the clustering key. Since table data can only be ordered one way, there can only be one clustered index. There is another type of index called nonclustered that does not impose any order to

the table data. We'll touch on nonclustered indexes a bit in the next module when we talk about other types of constraints. With these new tables, our order system is now in the first normal form or 1NF. I've also partially fixed the insert anomaly. I can insert new customers without affecting anything else. Unfortunately, I can also insert orders for nonexistent customers. We'll fix that shortly. Now let me populate the tables with some data. First, I'll add two customers. Then I'll add some stock. Now I can create some orders. Finally, I'll add some items to the orders. Now, let's look at what we have. Everything is populated correctly. However, I used a couple of educated guesses. I guessed that Arthur would be CustId 1 and Trillian CustId 2. I also guessed the OrderIds. These were educated guesses since I was starting with empty tables. In a real application, you'll need to find these IDs by looking them up with other queries. But let's move on for now. Can we improve on this? Yes we can. Let's look at the second normal form, 2NF.

Understanding Second Normal Form

Second normal form or 2NF builds upon the first, so the first rule is unsurprising. The database must be in first normal form or 1NF, and the second rule states that only single column primary keys are allowed. Well, actually the requirement is stated like this, no non-key attributes should be dependent on any proper subset of the key. Although it's possible to satisfy this rule with a composite key, if there are no composite keys, then the only proper subset is the empty set. That implies a single column primary key, which is the standard approach to this problem. And this will mean a change to the Stock table we just built since its composite key is comprised of two columns, the SKU and the Size. But if I change that, I will also have to change the OrderItems table, which refers to the Stock table by those same two column. Let's see that in the next demo.

Demo 3 - Transforming the Design to 2NF

In this demo, I'll change our tables into second normal form, 2NF. The Customers table is fine as it is. I'll leave it alone for now. The Stock table, though, needs a new column, which I'll call StockId to be the single column PRIMARY KEY replacement for the former two-key column. This kind of key is called a surrogate key since it is a surrogate for the business key, which is the composite key made up of the SKU and Size columns. The Orders table is fine, but the OrderItems table needs to change. Here I've replaced the former SKU and Size columns with a reference to the new StockId column in the Stock table. As before I can populate these tables and view the results. Once again, I guessed at the StockIds and got them right. However, shouldn't this system stop me somehow if I try to reference a nonexistent StockId or CustId or OrderId? Why, yes, it should. To bring this kind of muscle to the system, we need a new kind of constraint called a FOREIGN KEY constraint. It says that a column refers to a key in a different or foreign table. Let me show you how to do that. In the Orders table definition, I will add a constraint to the CustId. It looks like this. It may seem a little long, but there are a few parts to it. First, I need a name. I add it to the convention I used for primary keys. I start with FK and the table name and column of the referring table, then add the table name and column name of the reference to table, all separated by underscores. Then I tell SQL Server that this is a FOREIGNKEY constraint that references the Customers table, column CustId. So it really all makes sense. Now I need to do the same thing in the OrderItems table, but here there are two foreign keys, one for the Orders table and one for the Stock table. As before I can populate the tables and show their contents. Now what happens if I try to insert an order item for a nonexistent order? Let's try it. As hoped for, SQL Server stops me and tells me that I'm trying to violate a FOREIGN KEY constraint. Well, that's good. It should stop me. Foreign keys are used by SQL Server to preserve referential integrity in the database. Now BobsShoes database is in second normal form or 2NF. Still, there is more to do. Let's look at the third normal form, 3NF.

Understanding Third Normal Form

Third normal form or 3NF builds a bond to 2NF. So the first rule is pretty much what you'd expect. The database must be in second normal form or 2NF. The second rule states that there can be no transitive functional dependencies. Well, that's math talking. What it means is this. Column values should only depend upon the key. This also implies that for any table in 3NF, an update to one column should not cause an update to another column unless that other column is a key. A memorable way to describe 3NF is captured in this quote from Bill Kent, who wrote a guide to normal forms back in 1983. He said, "Every non-key must provide a fact about the key, the whole key, and nothing but the key. " Any column in the table that is not part of the table key is a non-key. These are usually called attributes in relational language. So there should be no column that is not dependent on the key. As it turns out, I sneakily added such a column to the Customers table. Let's look at that and fix it in the next demo.

Demo 3 - Transforming the Design to 3NF

Here's the Customers table in 2NF. The problem is the Salutation column. Suppose Trillian, a very oldfashioned lady, gets married to Arthur Dent. Well, being old-fashioned, she will change her name to Trillion Dent. Also, her salutation will change from Miss to Mrs. That change exposes the transitive functional dependency since the salutation is dependent upon the name but is not a key column. To fix that, I'll add a Salutations table. But before I do, I want to point out the DROP statement. Since my database is now constrained by foreign keys, I can't just drop tables in any order. I need to drop them from the last reference table up to the first referencing table. Now this is a small database, so I know which is which. In a larger database, you would need to know in advance or investigate the dependencies first. That's not too hard, but it's a little off topic, so I won't go into it here. So now I can add the Salutations table and change the Customers table to refer to it, with the appropriate foreign key definition of course. Now let's re-create the tables by running the DROP and CREATE statements. Good. Now let's populate them. First, the Salutations table, then the Customers

table with references to the Salutations table, then the rest of the tables. Okay, our database is now in 3NF. We've eliminated the insert, update, and delete anomalies for all practical purposes, or did we? Something is still bothering me about the Customers table. It's the addresses. The addresses are not dependent on the customer ID, not like the customer name is. Plus, I'm sure you can easily think of examples where many people or businesses share many of the same addresses. For example, people who live in an apartment building or businesses located in skyscrapers. Normalizing address information is a common task in database design. There are different approaches to it. Here's a simple one. Separate the city, state, province, country, and postal codes into a separate table. These change infrequently if at all, so it makes sense to keep them together. Then change the Customers table adding a new column that will be the FOREIGN KEY reference to the new City, State, Country table. You know, I'm going to let you take a crack at that. Starting from the tables used for this demo and available in the download section for this course, see if you can create and populate a new table holding the city information and modify the Customers table to use it.

Considering Other Normal Forms

Most database designs don't go beyond 3NF. Even so, you need to understand the other normal forms. So let's take a look at them. The Boyce Codd normal form is a slightly stronger version of 3NF developed in 1975 by Raymond F. Boyce and Edgar F. Codd. It stipulates that any dependencies in a 3NF compliant table must either be trivial, for example, where shoe width is a part of the shoe size column in the Stock table I've been working on if I also had a separate width column, that is, or that the dependent column depends on a super key. Well, what is a super key? A super key is a key that uniquely identifies a row but is wider than necessary. For example, in the Stock table, the SKU, Size, and Description form a super key, but the description is not needed for uniqueness. However, if a new column, say, Color, was added, the Stock table would not conform to BCNF since color is dependent on the shoe style but not the size. Not only that, but real-world considerations come into play. For example, men's tuxedo patent leather shoes are usually black. The Stock table

would need to be split into new tables for compliance. Note, however, that it is rare for a 3NF table to not also satisfy BCNF. Fourth normal form is concerned with multivalue dependencies. This extends and generalizes 3NF from a single column dependency to multicolumn dependencies. For example, in an Address table, columns for City and State are dependent on Postal Code and vice versa. These are the kind of multivalue dependencies the 4NF addresses. We can also say that a database in 4NF has two-part join dependencies, which simply means that we can reconstruct any un-normalized table using joins onto tables. This normal form generalizes upon 4NF to impart join dependencies. Only rarely is a 4NF table not in 5NF. These are situations in which a complex, real-world constraint controlling combinations of values in the 4NF table is not implied by the table structure, and the application program maintaining that table must implement the logic by itself. 5NF takes care of this. Sixth normal form seeks to reduce tables to irreducible parts. It can be important when dealing with temporal variables or other interval data. For example, if we want to enhance the Customers table to include date ranges when customer names and addresses are valid, the other normal forms are insufficient. Fortunately, starting with SQL Server 2016, a new kind of table called a Temporal table has been added for just this kind of situation. This kind of requirement is common in data warehousing situations were a Kimball database design is used. There are other normal forms in development that are not generally accepted yet. This remains an active area for research.

Summary

In this module, we look at normalization and worked on normalizing the database for Bob's Shoes order system. The main ideas behind normalizing a database are to remove redundancy, which promotes data integrity, and reduce the chance that anomalies may occur as the data changes. I worked through the first three normal forms and transformed the order system along the way to conform to each one. I introduced you to primary keys and foreign keys as constraints to offload the work of preserving and protecting normalization away from application programs and onto the database system. Then I gave you a short overview of some of the higher normal forms. It's important

to understand that normalization is still an active area of research. It is an area to watch though if you want to be a database designer. To help you remember what we're doing when we normalize a database, I'll quote Bill Kent again, "Every non-key must provide a fact about the key, the whole key, and nothing but the key. " Well, nothing will improve your normalization skills like practice. That's why I gave you a little exercise a few minutes ago. See if you can solve that before we move on to the next module, Ensuring Data Integrity with Constraints.

Ensuring Data Integrity with Constraints

Introduction and Overview

Hi. Welcome back to the course, Designing and Implementing Tables and Views in SQL Server. My name is Gerald Britton. Throughout this course, we've been refining the design for a database to hold an order processing system for Bob's shoe company. I started off with a naïve design with everything in one table. That meant a lot of repetition of things like names, addresses, and basic order information. And you also learned about the NULL constraint. In the previous module on normalization, we transfer on the design into third normal form or 3NF. To do that, I introduced two other constraints, the PRIMARY and FOREIGN KEY constraints. In this module, I'm going to explore these constraints in more detail and introduce three more constraints, the UNIQUE, CHECK, and DEFAULT constraints. Here's what's ahead. First, I'll revisit the NULL constraint, when to use it, and a little bit about an ongoing controversy about the nulls. The NOT NULL constraint should be paired with a default option, so that will come next. I'll revisit primary keys and go a little deeper into two ways of defining them and how to select good ones. What happens with FOREIGN KEY constraints when you update rows or delete them? I'll show you how to set up your foreign keys to handle that in a process known as cascading updates. In addition to primary and foreign keys, there is a third type of key constraint you need to know about, the UNIQUE constraint. And I'll finish up this module by talking about CHECK constraints. As I've been doing all along, I'll continue to modify the design of Bob's order system adding appropriate constraints as I go.

Using NULL and DEFAULT Constraints

Part of E. F. Codd's original work on relational databases included a special marker for the absence of a value. We call that marker null, and that is also the keyword used in SQL Server. A value may be absent because it is unavailable or because it is inapplicable. An example for Bob's Shoes might be newborn baby booties, which are typically one side, so size is inapplicable. In some

cases, the value may become available later like the delivery date of an order. Null seems like a good match for this kind of attribute. However, having nulls complicates logic. SQL is famous for its three-value logic. It's simplest to remember that when SQL tests a condition and one of the values is null, the result is unknown, neither true nor false. However, even though that is easy to remember, it gets complicated with joins, aggregates, foreign keys, and other things. This has been the source of many bugs in SQL programming, not unlike Tony Hoare's billion-dollar mistake, the nulls found in other popular programming languages. The implementation of null in SQL has led to much criticism and calls for change. Even Codd, in his second version of his relational database opus, saw the flaw and advocated not for eliminating null but for adding more null values with different meanings. Others suggested different approaches. But looking at the situation today, it's easy to see that none of these proposals were implemented in any major RDBMS. It seems that null is here to stay. Well, what about the COLUMN constraint? The NOT NULL constraint implies that three-value logic will not be used for a column with this constraint because it can never be null, and that seems attractive. Also, this is the normal default when creating tables. However, the default can be changed at the database level or the session level. That means don't rely on the default. Be explicit about column nullability. So should you always use NOT NULL? Well, that implies that you will supply a value for every column in every row you insert, even if you don't know the correct value or it is not available yet. That is why there is a NULL constraint to explicitly state that a column is nullable. Of course, that means you have to worry about the implementation of three-value logic in the database and in your code. There is no hard and fast rule. Discuss it with your team. Choose your approach and be explicit. Well, there's another type of constraint that can help us. Meet the DEFAULT constraint. A DEFAULT constraint is used to provide a default value for a column. The default value will be added to all new records if no other value is specified. This can help when a column has a NOT NULL constraint. If a DEFAULT constraint is also specified, and if you don't know the value of that column when a new row is inserted, the default will be used instead. Defaults can be constants like strings or numbers and can also be function calls, which can be quite useful. You'll see examples of these in the next demo.

Demo 1 - NULL and DEFAULT Constraints

Recall the Order table from BobsShoes? Here it is again. Normally, when inserting rows into this table, everything must be specified, like this. Well, not quite everything. I don't need to specify a value for the OrderID column since that is an identity column which SQL Server auto-populates. Also, the OrderDeliveryDate is unknown when the order is placed, and that column is nullable to reflect that, so I don't need to give that a value either. We can't have an order without a customer, and CustID has a foreign key constraint to the Customers table, so I'll always need that. However, what about the OrderDate and Expedited columns? These are good candidates for defaults. Here's a command to add a default constraint to the Orders table for the OrderDate column. Let's walk through it. It begins with the ALTER TABLE command and specifies the table to be changed. Then there is an ADD CONSTRAINT subcommand. This command takes an optional constraint name. I highly recommend that you give proper names to all your constraints, including default constraints. If you don't, SQL Server will assign one with a random number on the end. I name them similar to the way I name key constraints. Start with DF, then add the table and column name and a brief hint as to what the default will be, a call to the Getdate function in this case. The next section indicates what the type of the constraint is, hence the word DEFAULT. That is followed by the default value, a call to the GETDATE function in this case. Finally, we identify the call in the constraint will be applied to. That's what FOR OrderDate means. I also like to indent commands like this, showing the subcommands indented from their parents. I just like things easy to read. Now let's also add a default constraint for the Expedited flag. The command looks much the same, except for the default value, 0, indicating false, and the column name, of course. With these new default constraints in place, I can omit values for those columns in an INSERT statement, which runs just fine. I can view the last order created, like this. You can see that the OrderDate and Expedited flag have been given their default values. Of course, you can add default constraints when you create the table. Here's a revised definition for the Orders table with those two default constraints baked in. You may be thinking that the simple table definition we started with is getting complicated, and you are correct. What we're

doing here though is pushing work down to SQL Server. The result is less application code and more consistent results. Now what happens if you want to change a default constraint? Maybe the default order date should be the current date plus one day. You might think that this would do it. However, that red squiggly in Data Studio is a clue that something is wrong. In fact, you cannot alter a constraint. You have to drop it and recreated it, like this. And that is the rule for all constraints, not just default constraints. Now, before we move on to other constraint types, see if you can add reasonable defaults to the other tables in BobsShoes order system. I think there are a few opportunities. Pause this video, roll up your sleeves, and take a crack at it.

Implementing the PRIMARY KEY Constraint

Let's talk a bit more about primary keys. In the last module, I used primary keys as constraints on the tables in Bob's Shoes order system to ensure uniqueness. There are a few more things to think about when designing primary keys that we need to cover. SQL Server implements the PRIMARY KEY constraint with a backing index. And therein lie a few choices. The first choice is an important one. Since the primary key is backed by an index, what kind of index should that be? There are two choices--clustered and nonclustered. Looking at the documentation from Microsoft, you can see that clustered indexes sort and store data rows in the table based on their key values. These are the columns included in the index definition. These keys are stored in a special structure called a B-tree that enables SQL Server to find the row or rows associated with the key values quickly and efficiently. There can be only one clustered index per table, however, because the data rows themselves can be stored in only one order. If a table has no clustered index, it is called a heap. Data rows are stored wherever they fit in no particular order. This is why we say that a table is either a clustered index or a heap. It has to be one or the other. Now if your primary key is an identity column on a clustered index, like I've been using for Bob's Shoes order system, this means that new rows, which get new everincreasing identity values, will always be inserted at the end of the data and that the data is always in order of the ID. Well, that could be a good thing. Since SQL Server maintains a

clustered index in sorted order, it means less I/O when inserting new rows and when reading the table in the order of the identity column. On the other hand, if your application mainly reads from a table in a different order other than that of the identity column, this can mean more jumping around the disk to get the rows you want. For example, if you're producing a report of customers, chances are you want to keep that report in the order of the customers' names, not their IDs. So before you just take the default and use a clustered index for your primary key, take a look at the alternative.

Using Index Types and the UNIQUE Constraint

Nonclustered indexes have a structure separate from the data rows. A nonclustered index contains the nonclustered index key values, and each key value entry has a pointer to the data row that contains the key-value. That data row may be part of a clustered index or a heap. Like clustered indexes, the nonclustered index structure is stored as a B-tree for efficient retrieval. Nonclustered indexes do not affect data rows when changes happen to the index. Only the index structure is affected, and usually that is a small fraction of the size of the data rows. And nonclustered indexes might also include some of the data columns. This option can reduce I/O for columns that are frequently accessed using the nonclustered index. For BobsShoes Customers table, it might make sense to put the customer ID, which is an identity, on a nonclustered index and the customer’s name on a clustered index if most sequential access was in customer name order, that is. So far in this course, I've talked about PRIMARY KEY constraints and FOREIGN KEY constraints. Before I jump into the next demo, though, I want to show you yet another key constraint, the UNIQUE constraint. A UNIQUE constraint makes sure that there are no duplicate values of a column or columns independently of the primary key. One difference with primary keys is that the UNIQUE constraints allow for the value null. However, since this constraint enforces uniqueness, there can be only one null value per index column. UNIQUE constraints are ideal for business keys in tables where the primary key is a surrogate key such as an integer column with the IDENTITY property. A UNIQUE constraint can also be referenced by a foreign key, a property which we will use in the demo. And like

primary keys, UNIQUE constraints are backed by an index. That means you need to decide if that should be a clustered or nonclustered index. Okay, time to put these four concepts, PRIMARY and UNIQUE constraints and clustered and nonclustered backing indexes, together.

Demo 2 - Implementing PRIMARY KEY and UNIQUE Constraints

In this demo, I'll mix up the PRIMARY and UNIQUE constraints with the two index types, clustered and nonclustered. You'll remember the little Salutations table from the previous module. It has just two columns, an ID and the salutation itself. Here I've modified the definition we had by adding a UNIQUE constraint. Notice that it looks like a PRIMARY KEY constraint. And you should give it a name. I'm using the prefix UQ here to identify my constraint as a UNIQUE constraint, then the keyword UNIQUE followed by the type of index. If the type of index is not specified, a UNIQUE constraint defaults to a nonclustered index and a primary key to a clustered index unless there is already a clustered index, in which case a primary key will be backed by a nonclustered index. Note that I can create PRIMARY KEY and UNIQUE constraints without naming them. The system will then generate names for them. However, as before, I recommend that you name your constraints and indexes explicitly. It makes things easier for your teammates and even you six months down the road. Actually, I don't need to specify clustered and nonclustered in this table definition since those are the defaults for those constraints. However, I want to obey the mantra that explicit is better than implicit. You should to. In this second version, I've switched up the index types keeping everything else the same. You may be worried about putting a UNIQUE constraint on the IDENTITY column since UNIQUE constraints allow nulls. However, this also has the NOT NULL constraint so that property is still enforced. Also, SQL Server will never generate a null for a new identity value. In the Customers table definition that follows, a FOREIGN KEY reference does not care whether the reference is the PRIMARY KEY or UNIQUE constraint. Either will do. I've also modified the Stock table. If you recall from the previous module, we started out with a primary key on the SKU and Size columns, then moved it to the surrogate StockId column during normalization. that primary key also

enforced the uniqueness of the SKU/Size combo, which is, in fact, the business key. So I've added a unique index on that as a new table constraint. It must be done this way since two columns are involved. If I run the whole script, the tables are redefined with their new constraints. Then if I run a section to populate the tables, it also runs fine. Note that I can collapse the longer statements in Data Studio to be able to view more of the script at once. Now why don't you take a crack at adding a UNIQUE nonclustered constraint to the OrderItems table. Bob has said that the same stock items should only appear once per order. How would you do that? Pause this video and try that out.

More About Foreign Key Constraints

In the last module, I set up foreign key relationships between several tables. Still, there is much more to learn. Here I want to cover some of the missing pieces. A foreign key works by building and enforcing a link between two tables. This link controls the data that can be stored in the foreign key table. The link is controlled by referencing a primary or unique key in a base table from a referencing table with the same columns as the key in the base table. In the Bob's Shoes order system, the OrderItems table references the Orders table by including the OrderId, and the OrderItems table refers to the Stock table by another foreign key. Foreign key definitions on those columns enforce the links. We'll see that in the demo. Foreign keys help preserve referential integrity. For example, in the OrderItems table I've been working with, I cannot add a new row referencing a nonexistent OrderId. Also, I cannot delete or update the key in the base table since it is bound by the FOREIGN KEY constraint, but this can be a problem in some situations. For example, suppose Bob's Shoes stopped carrying brown sandals in size 17. No problem, you say. Just delete that row from the Stock table. Well, suppose there is an existing order for just that shoe in just that size. There are a few options. Issue an error message and stop the deletion leaving an order for a discontinued product, delete the

OrderItems that match that Stock item, or perhaps replace the FOREIGN KEY reference in the OrderItems table with a null. The rules for handling this situation and others like it are known as cascading referential integrity. In the demo, you'll see various ways of setting this up.

Demo 3 - Using FOREIGN KEY Constraints

Here are the Stock and OrderItems table definitions we've been working with. Let's add that big sandal I mentioned. Okay, now querying the table, you can see that the new stock item ID is 4. I'll add that to an existing order. Now suppose I hear that the company is discontinuing that sandal in that size. So I'll try to delete it from the Stock table, but I can't. I get an error that this would break referential integrity since the OrderItems table has a FOREIGN KEY that refers to it. By the way, you can see the value in explicit naming here. If the name of the FOREIGN KEY constraint were a system-generated ID, it would be hard to figure out the problem. So, I get an error and report it to my management. They tell me that I should just delete any OrderItems for this shoe. Now I could explicitly delete those rows, but then I'd have to remember that for next time. Or I might be on vacation and someone else will have to go through it all over again. What can I do? I can specify a CASCADE option on the foreign key. Here I set the action to CASCADE. That means that if the key reference is deleted from the Stock table, SQL Server should also delete any row containing that foreign key in the OrderItems table. The DELETE operation is cascaded to the referencing table. So, I'll re-create the table and add the order back in. There it is. Now I'll try that DELETE again. It worked! And what about that order item? Gone! SQL Server did its job and deleted the referencing row when the referenced key was deleted. Referential integrity is preserved, though I might have an angry customer or two with large feet. Here's another example using the cascade option. I confess that this is completely contrived and not related to Bob's Shoes. I'm also breaking best practices about naming things. I just want to get to the heart of the matter. Here I create two simple tables with a foreign key relationship from table 2 to table 1. I've name these tables using throwaway names. They will not be part of any production system. Note the foreign key option, ON UPDATE SET NULL. This means that

if the key in the reference table changes, the key in the referencing table should be set to null. Next, I populate both tables and show what is in table 2. Now I want to change status B to status C. Here's an UPDATE statement to do that. Will it work? Let's find out. It worked. But what about that foreign key? It was set to null as requested. So now you've seen the most commonly used options for foreign key cascading changes. There are a few more, so let me review them all. The CASCADE option means to update any referencing tables with the changes made to the referenced table. NO ACTION means do not allow the delete or update, which means throw an error and leave things as they are. This is the default setting. SET NULL means set the foreign key values to null if the corresponding row in the base table is updated or deleted. For this constraint to execute, the foreign key columns must be nullable. This can be useful, though, if the foreign key refers to a table containing optional data. For example, suppose our order system had optional delivery instructions. SET NULL might make sense in that case. SET DEFAULT as the name implies sets the foreign key values to their default values when the corresponding row of the base table is updated or deleted. If no default is defined and the column is nullable, the value is set to null. One difference between primary keys and foreign keys is that with foreign keys, the backing index is not automatically created. However, creating such an index is recommended in many situations. See the Pluralsight course on indexing for a deep dive into that subject.

Introducing CHECK Constraints

So far, I've covered the NULL, PRIMARY KEY, UNIQUE, FOREIGN KEY, and DEFAULT constraints. Time for the last type, CHECK constraints. A CHECK constraint is a way of declaring limits and validations on data inserted to or updated in a table. Since the CHECK constraint is part of the table definition, it is automatically performed by SQL Server. It is an example of declarative programming, which you can set and forget. This saves application coding for data validation and ensures consistent, ironclad limits on data that is persisted to any table. Checking limits in code is an example of imperative programming. It doesn't matter if you do that in T-SQL, C#, Java, Python,

JavaScript, or whatever your favorite language is. The risk is that two programmers or even you want on two different days might write the code a little differently, and away goes consistency. CHECK constraints can be defined at the column and table levels. And you can have as many as you need or want. Internally all CHECK constraints are table constraints, but SQL Server accepts simplified syntax when CHECK constraints are defined at the column level. The basic parts of a CHECK constraint are its name and the condition to be checked. As with other constraints, you can let SQL Server generate the name, but it is a better practice to name them explicitly. You'll see an example of why that can be important in the next demo. The condition must evaluate to a Boolean expression, true or false. The expression can be any valid T-SQL expression including comparisons, membership tests using IN, function calls, and anything else you can dream up as long as it evaluates to true or false. That is very powerful. One type of expression not supported by SQL Server or by the majority of commercial databases is a query, that is, your check condition cannot contain a SELECT statement even though that is included in the ANSI SQL standard. However, you could call a function that does contain a SELECT statement. So, this limitation is not really very limiting. Although, there is a catch, as you'll see. There are more caveats and special cases that go with CHECK constraints, though you may never hit them on the job. Consult the official documentation for details. Okay, time for a demo.

Demo 4 - Using CHECK Constraints

Here I've put a few CHECK constraints on our tables, starting with the Salutations table. A salutation is useless if it is blank. So, I've got a CHECK constraint to prevent that. Note the name I used. It explicitly states the rule being checked. Now let me try to violate it. See the error message. It includes the name of the constraint being violated. Now let me do that again using a system-generated name. Just how useful is that error message? You'd have to dig in to the table definition to find out what's really wrong. Now let me try a table constraint. Bob just told me that the stock item description must be different from the SKU. Here's one way I could do that. I need to use a table constraint since two columns are involved. And if I try to violate that constraint, SQL Server stops me. Next, let's pretend that Bob is really picky about his customers. He only accepts customers from the

US, the UK, and Canada. To control that, I can use this CHECK constraint, which tests membership using an IN clause. For the last example, I'll pretend that in order to validate an order, I have to do some fancy date checks. To do that, I've created a scalar function that returns a 1 or a 0 to represent true or false. In this case, true means the dates pass the test. My new CHECK constraint calls the function and checks the result. Let me try to violate it. Foiled again. So CHECK constraints work with functions too. A scalar function can do a lot, including querying other tables. A word or two of caution is in order though. The first caution is about performance. If a scalar function used in a CHECK constraint uses queries that involve large tables or complex joins, don't be surprised if the time to insert or update a row increases. That doesn't mean don't use functions or functions with long queries. After all, if you did the same work in your application logic, it could run just as long. The second caution is this. If you use a function in a CHECK constraint, and you later change the function so that it returns different results, SQL Server will not automatically reject the constraint. You need to do that manually. There's an ALTER TABLE command for that that tells SQL Server to explicitly CHECK a constraint. Here's an example. The challenge may be that you have a helper function used in CHECK constraints in many table definitions. If that helper function changes, you'll need to find all the places it is referenced to recheck the constraints and verify that the change to the function does not break anything. Now there are lots of opportunities to add CHECK constraints to the tables in Bob's Shoes order system. This would be a great time to pause this video and write some of your own. Some things you might want to check. Can the number of items ordered be less than 1? Can a delivery date precede the OrderDate? Can you write a CHECK constraint to verify shoe sizes for the Stock table. Or how about a constraint to verify country names using current world ISO standards? And can a price or discount be negative? Well, give those a shot, and I'll see you later.

Options for Defining CHECK Constraints

To complete this look at constraints, let's review the major statements you will need. You can add constraints when you create a table. They can be specified at either the column level or the table level. Remember to use a table constraint if two or more columns are involved. You can have multiple constraints per column or table in either location. The only exceptions are FOREIGN KEY constraints, which cannot be used with temporary tables or table variables. You can add new constraints at any time using the ALTER TABLE ADD CONSTRAINT command. Both column and table constraints can be added this way. By default, SQL Server will check the table against the newly added constraint. The optional parameter with no check will add the constraint without checking the table. The default is to check that there are no constraint violations and issue an error message if any are found. If you no longer need a constraint, you can remove it using the ALTER TABLE DROP CONSTRAINT command. You also need this to change a constraint since there is no ALTER CONSTRAINT command. Changing constraints always means dropping and re-creating them. By the way, if you drop a key constraint backed by a clustered index, the table becomes a heap. If you need to temporarily disable a constraint, use the ALTER TABLE NOCHECK CONSTRAINT. You might do this, for example, when bulk inserting data known to be good, and the constraint slows down the INSERT operation enough to be a problem. Note that you can only disable FOREIGN KEY and CHECK constraints, not other constraint types. You can re-enable or disable a constraint with the ALTER TABLE CHECK CONSTRAINT command. If you want to also reject the constraint, add the WITH CHECK option to the command. For example, you might use WITH CHECK after modifying a function used in a check constraint or to verify FOREIGN KEY constraints. The ALTER commands all need a constraint name, but you can also use the keyword ALL to perform the action to all constraints at once.

Summary

In this module, we look at the six types of constraints available out of the box in SQL Server. I first reviewed the NULL and NOT NULL constraints, when to use them, and a little about the controversies regarding THEM. I typically use the NULL constraint only on columns where the data may not be available until a later time and no suitable default exists. The DEFAULT constraint is perfect for nullable columns where there is a suitable default. Note that all tables should have a PRIMARY KEY constraint. Otherwise, your database is not in third normal form, considered the baseline for good database design. I also showed you the UNIQUE constraint and how that can be used to ensure uniqueness on a business key if the primary key is a surrogate key. But, of course, there are many other uses. The section on FOREIGN KEY constraints showed different methods to handle FOREIGN KEY references when corresponding parent rows are deleted or updated. Cascade and update are perhaps the most used options. Finally, I showed you the CHECK constraint and how you can use it to enhance data integrity while reducing application code. Well, for this course, we finished with table definitions. However, I have not covered all table types and options. In fact, some of them deserve courses on their own. What I have covered, however, is enough for the bulk of what a typical database developer will need for table and database design. In the next module, I'm going to start talking about a related topic, Views, which are projections of existing tables. See you there!

Designing View to Meet Business Requirements

Introducing Views

Hello again. Welcome back to the course, Designing and Implementing Tables and Views in SQL Server. My name is Gerald Britton. In the previous three modules, I showed you the basics of designing tables, normalizing them up to the third normal form, and applying constraints to enforce data and referential integrity. At this point, we have a good set of tables to operate Bob's Shoes order system. One of the effects of normalizing tables is that it takes one or more joints to reconstruct the original, un-normalized View. Customer data goes to the Customers table, CityState data to the City table, inventory to the Stock table, and the orders themselves to the Orders and OrderItems tables. Correctly writing joins every time to get the right data the right way is a bit of an art. Give two developers the same task of joining these tables together to produce a report or view it in a browser, and you'll likely get two different solutions. That is at least a maintenance problem, at worst one of the joints may work most of the time but fail with some edge cases. Wouldn't it be nice if you could somehow take a best-of-breed query to join up these tables and encapsulate it so that everyone could use it, and you only have one thing to maintain. Well, that's exactly what views are for. In this module, you'll learn the basics of creating views to encapsulate queries and begin to discover other great uses for views.

Overview and Motivation

In the upcoming videos, you'll learn the motivations for views and the problems they address, the limitations of views, what they can and cannot do, three types of views--basic, partition, and indexed. You'll learn how to protect views against table modifications. Without such protection, a view may fail or return erroneous results, if an underlying table is changed or dropped. You'll learn how to persist view results using indexes. Sometimes a view is costly to run. In such cases, performance can

be enhanced by adding an index, which saves the view results. I'll also talk about updatable views. Views can be treated, for the most part, like tables including DML operations such as INSERT, DELETE, and UPDATE with certain restrictions. Now let's kick this off by looking at the first item on the list-motivation. The fundamental theorem of use is that they are used to encapsulate queries. A senior developer can write, test, and optimize a query for general use. That query can then be encapsulated in a view, which can be used instead of the bare query. This improves code reuse and reduces maintenance. A view can customize results from a table or set of tables to focus, simplify, and customize the perception each user has of the database. For example, in Bob's Shoes order system, there's no point in showing the delivery date for a new order since the delivery hasn't been scheduled yet. By the same token, the warehouse shipping the order may not care when the order was placed and only want to see the requested delivery date. Views can add a layer of security to the database by acting as a proxy to the underlying data while hiding some of it. You can enforce this by granting users access to the view but not the underlying base tables. Views are also a way of providing backward compatibility. Suppose I need to add a column to the Stock table to show the popularity of an item. Applications reading that table may be affected by the new column. Or perhaps I need to change a data type. Such a change would undoubtedly break existing applications, but views offer a way to mitigate these effects until all applications are updated. You can think of views as virtual tables. Like tables, they can be queried, indexed, and even updated, though some restrictions apply to indexing and updating views as you'll see. A user of a view need not know that they are querying a view and not a table since it mostly looks and acts just like a table.

Reviewing the Bobs Shoes Order System Design

Let's review the schema of the table we've got so far. All customers are in the Customers table. It has foreign key relationships with the Salutations and CityState tables. On the other side is the Orders table with one row per order. It has a foreign key relationship with the Customers table to enforce that relationship. The OrderItems table references the Orders table and the Stock table. The table definitions matching the schema are available in the downloadable items for this module. Now let's see about creating a view.

Demo 1 - Creating a View to Produced a List of Customers

For the first demo, I'll create a view to return a list of customers with their salutations, names, and addresses. This means that I have to join three tables, the Customers, CityState, and Salutations tables. Such a query might look like this. Before we go further, I want to highlight some best practices in this query. First, alias the tables being queried. I've aliased the Customers table as cust, the CityState table as city, and the Salutations table as sal. Second, use ANSI join syntax. That means always using the keyword JOIN to join tables, never a comma as in some older syntax. Third, always use two-part names for column references consisting of the table alias and the column name. Sticking to these rules will make your life and that of those maintaining your code easier. If I execute this query, the view is created. Then I can query the view as easily as a table, like this. Notice that the columns all have their original names reflecting the tables they came from. However, this is not necessary and may not be desirable. You may recall that in computer science, there is the notion of information hiding. David Parnas came up with the idea of information hiding back in 1972. Then it was in the context of object oriented program design. The principle is that of segregating the design decisions that may change or are likely to change. It applies here since the tables underlying a view may change their schema or column names, but an application program, which may be just another piece of T-SQL code, can use the view as an interface to the base tables. Let's hide the base tables in this view. To do that, I'll simply alias the columns from the original tables. This ALTER statement

does that. Now when I select from the view, the schemas of the base tables are hidden. I've achieved the desired segregation that information hiding offers.

Demo 2 - Using WITH SCHEMABINDING

There's a problem with the view I just defined. It's not obvious at first, so let me show you. Here I have a script to illustrate the problem. First, I create a test table with just two columns, an integer and a float. I populate that table with some sample values. Next, I create a view on that test table. I can easily query the view and get the table contents. Now I'll drop the table. SQL Server does not complain. But what if I query the view again? Boom! It throws an exception unsurprising. But that's a problem. Any application program including other T-SQL code will now break if it tries to query the view. How about something more subtle? I'll redefine the table but with a twist. I've switched the meaning and datatypes of the two columns. The view seems to work again, but does it work as expected? It does not. The first column was supposed to be an integer, and the second a float, not the other way around. Again, application code using this view will break. What can I do? Well, let's go back to the top of this script. This time I'll add an option to the view, WITH SCHEMABINDING. When you use the option WITH SCHEMABINDING in a view, three rules apply. First, the tables the view references cannot be changed without modifying or dropping the view first. The tables can neither be altered nor dropped. This protects application code against such attempts. Second, any SELECT statement in the view must use two-part names, that is, the schema name plus the object name, for any tables' functions and other views referenced. And this rule also implies the third rule, all referenced objects must be in the same database. In other words, three- or four-part names are not permitted. As a general rule, use WITH SCHEMABINDING whenever you can. Other developers and your future self will thank you. Now back to the demo. This view definition fails at first. I've literally said bind this view to a schema, but I've specified no schema. To fix it, I'll just add the default schema, DBO, to that table reference. Let's run the same tests as before. Querying the view works. What about dropping the table? I'm prevented from doing that, and that's a good thing. Let's see if I can

change the table with this ALTER command. I can't! Schema binding has blocked my attempt. That means that applications using this view are now guaranteed that it will not return inconsistent results due to base table changes. Now I'll just clean up those test objects. Back in our CustomerList view, I'll add WITH SCHEMABINDING to it to lock it down and verify that it still works.

Working with Updateable Views

like tables, views can be updated, subject to certain restrictions. That means that you can insert, delete, and update rows in A table through view. but in order for this to work, SQL Server has to know what table to update. And that leads to the first restriction. Any modifications, including UPDATE, INSERT, and DELETE statements, must reference columns from only one base table. That means that if your view joins two or more tables, you can update only one of those tables through the view. The second restriction is that the columns being modified must directly reference the data in the base table so the columns cannot be derived, which excludes aggregate functions like sum or average, computed columns and columns from set operators like union and intercept. The third rule says that the columns being modified must not be affected by GROUP BY, HAVING, DISTINCT, PIVOT, or

UNPIVOT clauses. This rule and the previous one are consequences of the first rule. Think about it. How could SQL know which row of a base table to update? If data is grouped, there could be no matching rows or many of them. Also, your view cannot use TOP or an OFFSET clause together with the WITH CHECK OPTION clause. You'll see the WITH CHECK OPTION clause in the next demo. It forces all data modification statements executed against the view to follow the conditions in the SELECT statement by making sure that the data remains visible through the view after the modification is committed. Now if the view were to contain a TOP or OFFSET clause, that would imply that some rows may be hidden after the data is modified. So TOP and OFFSET are not allowed when also using the WITH CHECK OPTION. You'll see these rules in action in the following demo.

Demo 3 - Updating Tables using Views

This is the CustomerList view I've been working on in this module. It joins three tables. This view is updatable. Let's see that. This query returns the current result of that view. Now suppose Trillian and Arthur get married and she takes his surname. I can update the Customers table through the view like this. Now I'm doing it in a transaction so that I can discard the changes. Selecting from the Customers table, not the view, shows that, indeed, the base table was updated. I'll roll back those changes because I just remembered that Trillion also wants her salutation changed to Mrs. So I'll just add that to the UPDATE statement. What should happen? If you guessed that this would fail, you are correct as you can see. I tried to update both the Customers table and the Salutation table through the view. That's not allowed by the first rule. I've got another view here. This one returns a summary of current orders showing the CustName and TotalQuantity ordered. Now I just heard that Trillion's order should be expedited, so I try this UPDATE, but it fails because the query contains a derived column in violation of rule two. Let's try another view, one showing just the TotalItems per order ID. Can I update this one? No, I cannot since it violates rule three. Here's another view designed to only return customers whose names begin with the letter A. Notice that this view has the WITH CHECK OPTION specified. Selecting from this view returns just one row. Now let me try to change Arthur's name to Ford Prefect. Can't do that. I violated the WITH CHECK OPTION property, so my UPDATE is not allowed. This is a useful option on updatable views. You would need a good reason not to use it. As an exercise, pause the video here and see if you can create a view that cannot be updated because it violates rule four, that is, it contains a TOP or OFFSET clause and also specifies WITH CHECK OPTION.

Summary

In this module, you learned the basics of building views and that they can be thought of as virtual tables. For most use cases, they can be queried like tables, though there are a few limitations. You also saw the difference between views bound to the base table schemas and those that are not. Whenever possible, use the WITH SCHEMABINDING clause to prevent unexpected results in applications caused by base table changes. I talked about updatable views and the rules that must be followed. In the demos, I tried to violate those rules to show how SQL Server stops that from happening. You also learned about the WITH CHECK OPTION, which preserves the view results when updating or disallows the update. Hopefully you tried this in the mini-exercise at the end of the last demo. Now that I've covered the basics of views, let's look at one of the advanced types called an indexed view.

Implementing Indexed Views

Introducing Indexed Views

Hello. Welcome back to the course, Designing and Implementing Tables and Views in SQL Server. My name is Gerald Britton. Throughout our exploration of SQL Server views, I've been saying that they are similar to tables, but virtual. But if views are similar to tables, that means that they can be indexed. And as with tables, putting one or more indexes on a view can speed up your queries. In this module, I'll be creating indexed views for Bob's Shoes order system. There won't be a general discussion of indexing or how to maintain them or troubleshoot them. Pluralsight.com has other courses that do a great job of that. What you will learn in this module, though, are the requirements, recommendations, considerations, and, of course, the syntax for defining indexed views. In this module, the very first thing I'll do is describe what an indexed view is and how such a view is similar to an ordinary table and where it differs. Next, I'll discuss the requirements and restrictions placed on indexed views. There are quite a few. Most of the requirements stem from the fact that a view must be deterministic. You'll see just what that means. Then I'll look at some of the key recommendations for building indexed views, especially regarding date conversions. There are a few other considerations for indexed views that you need to be aware of to avoid surprises later on. Of course, throughout this module, you'll learn about the DDL syntax used to create, alter, and drop indexes on views. So what is a view? An indexed view is a persisted object stored in the database in the same way that table indexes are stored. The key word here is persisted. Another word sometimes used is materialized. That means that the index is written to disk. So, while the underlying view is still a virtual table, any index on it is no longer virtual. It is stored in physical form just like an ordinary table index. Now the query optimizer may use indexed views to speed up the query execution. The view does not have to be referenced in the query for the optimizer to consider it for substitution, and that last part is worth repeating. The view does not have to be referenced in the query for the optimizer to consider that view for a substitution. That's right! You can get performance gains from an indexed view without

even referencing it or using it by name on purpose. SQL Server knows that it is there and will use it if it thinks it will speed things up. When you learned about tables, you learned that clustered indexes define the order in which database pages are stored so that the table becomes a clustered index as opposed to a heap. I also showed that nonclustered indexes are separate objects that point to table pages whether the table is a clustered index or a heap. Indexed views are only slightly different in that they are never stored as heaps. That means there must always be a clustered index on a view, and that's a great segue into the general requirements. So let's look at those.

Requirements for Indexed Views

The first requirement for an indexed view is that the first index must be a unique clustered index. After the unique clustered index has been created, you can create one or more non-clustered indexes. Creating a unique clustered index on a view improves query performance because the view is stored in the database in the same way a table with a clustered index is stored. Indexed views require that certain SET options be in effect at creation time. There are several that I'll show you on the next slide. The view upon which you want to create an index must be deterministic. That's the reason for the SET options you'll see in a moment. The determinism requirement also implies some other things that I'll get into. Any view that you create an index on must have been created using the WITH SCHEMABINDING option. Also, any functions referenced by the view must have, likewise, been created using this option. There is a list of T-SQL elements that may not be used in the SELECT statement in the view definition upon which you wish to place an index. The list is long enough that a full discussion of the reasons for each item might just make your head spin, but I'll touch on some of the important ones. And note that if a view contains a GROUP BY clause, the key of the unique clustered index can reference only the columns specified in that clause. To make sure that indexed views can be maintained correctly and return consistent results, they require fixed values for several SET options. This is another way of saying that the view must be deterministic. And this table shows those options. There are seven SET options with the required settings given in

the second column labeled Required. Note that the required values are the same as the Server Default settings. However, you cannot generally count on the defaults being in effect. The rule, Explicit is better than implicit, applies here. And note that the defaults for OLEDB/ODBC and DB-library are different from the requirements and the server defaults. Also, if you set ANSI warnings to ON, that action implicitly sets ARITHABORT on as well. When you set them to OFF, an arithmetic overflow or divide by 0 does not cause an error, and null is returned instead. Well, that won't do for indexed values. That is why both of these must be on for an indexed view. It's easy to come up with examples for the others as well. Take a moment and work through these options to convince yourself that incorrect settings can lead to indeterminate results. The definition of an indexed view must be deterministic. That means that all expressions, including those in the WHERE and GROUP BY clauses and the ON clauses of joins must always return the same result when evaluated with the same argument values. An example of a deterministic function is DATEADD since it always returns the same result with the same inputs. If you know some functional programming, you can call DATEADD a pure function. On the other hand, the GETDATE function is not deterministic and also not pure since it can return a different result on every call. See what other examples you can find of nondeterministic functions. One of the properties of every column is the IsDeterministic property. You can query this with the COLUMNPROPERTY function as you'll see in a moment. Now floating-point data is a special problem since the exact result of an expression with floating-point numbers may depend on the processor or microcode versions in use. Such expressions cannot be in the key columns of an indexed view. Deterministic expressions that do not contain float expressions are called precise, and that is what you need for key columns and for WHERE, GROUP BY, and the ON clauses of indexed views. The COLUMNPROPERTY function will also show you if a computed column is precise. Now let's look at some of these things in the first demo.

Demo 1 - Determining Determinism

Here I have a script that tries to create an indexed view where at least one column is not deterministic and precise. First, I drop any existing view of the same name. Then I create the view WITH SCHEMABINDING as required. The view has two computed columns. The first is just A concatenation of an OrderId and an OrderItemId. But the second uses a floating-point number in its computation. Now having created this view, I can query the COLUMNPROPERTIES. Note that the first column, the concatenation of Order and ItemIds, is both deterministic and precise. But the second column, while deterministic, is not precise since it involves a floating-point number. Now let me try to create an index on this view. It fails. Note the error message. Column 2 is the problem. I think it's a good idea to pause this video for a moment and try to change the view definition to fix that. What would you do?

YMIVR - Yet More Indexed View Requirements

There are a number of other requirements placed on indexed views, so many that it would be a little tedious to go over each one, but let me highlight a few of the forbidden T-SQL elements: COUNT, ROWSETs, OUTER JOINS, derived tables, self-joins, sub-queries, DISTINCT, TOP, ORDER BY, UNION, EXCEPT, INTERSECT, MIN, MAX, PIVOT, UNPIVOT, and many more. See the official documentation for full details. At the time of writing, this bit.ly link opened up the page. Or you can simply search for Create Indexed Views in SQL Server. One other problem area concerns date literals. Look at the expression at the top. What is that date? Well, it depends on the locale setting. Some locales read this as the 12th of January, and others as the 1st of December. The ISO format, however, is always read as year, month, day, so January 12, 2020, in this example. It is deterministic. If you use date literals in your indexed views, the recommendation is to explicitly convert them to the type you want. The CAST and CONVERT functions have format styles that are deterministic. Use those. Now let's build an indexed view for Bob's Shoes. On to the next demo.

Demo 2 - Indexing the Customer List View

In the last module, I created a view to retrieve data from the Customers, Salutations, and City/State tables for easy use by Bob's business users. Here's the definition I ended up with. I've added one more column, the CustomerID. I'm going to need this since to create any index on a view, there must be a unique clustered index. Customer names are not unique, even if you include all the other attributes in the view. But the CustomerID is unique, since that is generated by SQL Server. Now that I have a new view, I'll put the first index on it. Here I will create the unique clustered index. I can select from the view as before, and now SQL Server has the option of using the new index in its execution plans. I'm working with a very small data set, however, so the new index may not be chosen. Notice the one option I've commented out, EXPAND VIEWS. This tells SQL Server to ignore any index on the view and expand the view into queries on the underlying tables. If I uncomment this line and run the query, the execution plan shows this in action. All the tables referenced in the view are queried directly. In other words, the view was expanded as the name of the option implies.

Demo 3 - Adding a Nonclustered Index and Views with Aggregates

Now let me also put a non-clustered index on this view, this time using the Name and PostalCode columns. There, the query works fine, and here I can also use the option EXPAND VIEWS if I want to. Here's another view I built in the last module, the OrderSummary view. Let's index that. Oops! I got an error. It seems I need to use the COUNT_BIG function here. In fact, the full rule is that if there is a GROUP BY, there must also be a COUNT_BIG. so let's put that in. Second try. Oops! Another error. The view contains an expression on the result of an aggregate function or grouping column. In this case, the problem is the in-line IF expression on the IsExpedited column. I'll remove that. Note that since an in-line IF is syntactic sugar for a CASE expression, using CASE here would not be any better. Now the index is successfully created. Since the view contains an aggregated column, the sum of the quantities, the index then persists that aggregate. For querying, this means that SQL Server no longer has to process the base tables to get those values. They are part of the

view's clustered index and can give a nice performance boost. This is not without a cost however, since if Orders or OrderItems are inserted or updated or deleted, then part of the index view will also have to be updated. Like so many things in database design, you need to weigh the cost against the benefits. For example, if you know that the view will be queried many more times than the base tables are updated, the index is probably worth it. The best approach is to get a baseline of current performance, then decide if the savings of having an index on a view is worth the cost of updating it when the base tables change.

Summary

In this module, you learned about indexed views. Since views in many respects can be thought of as virtual tables, the idea of indexing them is not unexpected. What can be unexpected, though, is the long list of requirements and restrictions placed on indexed views. When you create an indexed view, it is persisted to permanent storage just like an index on a table. This can lead to an increase in performance since SQL Server now has another option for satisfying a query. And depending on the edition of SQL Server you are running, the database engine will look at indexed views even if not specified in a query. Unlike table indexes, however, the first such index must be a unique clustered index. There is no such thing as an indexed view stored as a heap. Also, you cannot put a primary key on an indexed view. A primary key is a constraint, and that belongs on a base table. Second and subsequent indexes are non-clustered since there can be only one clustered index per object. A view that is indexed may contain certain aggregations, sum, for example. And, finally, as with all indexes, indexed views must be maintained as the base tables change. This can be costly, and that cost needs to be weighed against any perceived performance gains. Now there is one more view type to discuss in this course, partitioned views. See you in the next module.

Implementing Partitioned Views

Introducing Partitioned Views

Hello. Welcome back to the course, Designing and Implementing Tables and Views in SQL Server. My name is Gerald Britton. During this course, we've been building a solution to Bob's Shoes' expanding business. Now it's time to think about the long-term. Specifically, what happens with old orders several years into the new system? Normally you don't want to just throw that data away. At the same time, keeping years' old data in the current table will inevitably make the system run slower for some operations, especially maintenance operations like backup and CHECKDB. Also keep in mind that storage can be expensive. I'm not talking about drives you can buy at your local computer store or online. This is about server class storage to build a business on. Many enterprises use storage area networks or SANs to hold their persistent data. That's a good solution that scales well, but it can be expensive. On the other hand, storage does not have to be homogeneous. You can store current data on high-speed, solid-state drives and older data on cheaper spinning media. Still, you want to have the data available and queryable when the auditor comes knocking. Partition views offer a solution to this kind of problem. In this module, you'll learn how to create partition views and the requirements the tables must have to be part of a partitioned view.

Outlining a Partitioned View

Generally, a partitioned view is defined like this. It begins like any other CREATE VIEW statement, and you SELECT the data from the first table. That is followed by a UNION ALL operator and a SELECT statement for the data from the second table. If there is a third table, there is another UNION ALL and a SELECT statement for that data. This can be continued if you have additional tables with no built-in limit to the number of member tables in the partitioned view. The requirement for UNION ALL for partitioned views and the proscription against the UNION operator for indexed views implies that partitioned views cannot be indexed. However, there are some other requirements

and conditions. Let's look at those. The SELECT statements in a partitioned view should contain all columns in the underlying base tables. And columns in the same position in each SELECT list should be of the same type, not just types that can be implicitly converted, but the same actual types. Though not explicitly stated, generally you want to ensure that the columns also have the same semantic contents, such as dates, names, and quantities. At least one column in the same position across all the SELECT statements should have a CHECK constraint. That constraint has to be defined so that only one base table in the view can satisfy the constraint. That is, the member tables cannot have any overlapping intervals with respect to the constraint. This column is called the partitioning column and can have different names in each of the tables if required. In practice, though, keep column names the same across the member tables if at all possible, to avoid confusion. Such column names are also called conformant. And, finally, a column can appear only once in each SELECT list.

Reviewing the Requirements and Restrictions

There are a few important requirements for partitioning columns. First, the column needs to be part of the primary key of the table. This helps SQL Server ensure that any queries against the partition view only have to scan one table in the view if the partitioning column is specified in the query. Of course, if the partitioning column is not specified in a query to the view, all tables will be involved. Second, the partitioning column cannot be computed or have the IDENTITY property or have a default value, nor may it be a timestamp column, also known as a row version column. Though I haven't covered this type in this course, the official documentation contains a full discussion with examples. Third, there can be only one partitioning constraint defined on this column. Otherwise, the definition is considered to be ambiguous. And, fourth, there are no restrictions on the updateability of the partitioning column, at least not from the perspective of the partitioned view. Naturally any constraint on the column still applies to inserts and updates. The tables participating in a partitioned view are called member tables. They may be on the same instance as the partitioned view or on

different instances including remote servers referenced through four-part names or open data source or open row set table names. If one or more of the tables is remote, the view is called a distributed partitioned view and has even more restrictions that you can find in the official docs. A member table can appear only one time in the set of tables combined with the UNION ALL statement. And no member table may have an index on a computed column whether persisted or not. All member tables should have similar primary keys. Practically, this means that the corresponding columns in each table should be of the same type and in the same order and that there should be the same number of columns in the primary keys of each member table. This means that the only things that can differ are the column names. Finally, member tables must all have the same ANSI padding settings. This controls the way the column stores values shorter than the defined size of the column and the way the column stores values that have trailing blanks in char, varchar, binary, and varbinary data. The recommendation is that this option always be set to On. Don't change it without properly understanding all the implications, not just those for partitioned views. Now with all those restrictions and caveats under your belt, let's build some partitioned views.

Demo 1 - Building a Partitioned View for the Orders Table

In this demo, I'll start to address Bob's requirement of modifying the Orders table to keep current and older orders separated yet accessible. First, I'll add a partitioning column with the appropriate CHECK constrain. Then I'll create and run a partitioned view so you can see how this works in practice. Back in Data Studio, I'll start off by re-creating the Orders table. The first change is that the OrderID, which was the primary key, now has no constraint, though it is still an IDENTITY column. The second change is the addition of an OrderYear column. This will be the new partitioning column. As required, it has a constraint, in this case limiting it to orders from the year 2019. The other columns are the same, but I have a new PRIMARY KEY constraint. As required by partitioned views, the primary key now includes the partitioning column. the OrderItems table has also changed. I need to add the OrderYear column here too. This change is also reflected in the FOREIGN KEY reference,

which now needs to reference both columns. In order to have a real partitioned view, I need another table that looks like the Orders table. I created one that looks the same, although the names of the objects have changed adding 2018 on the end. Also, the CHECK constraint on the partitioning column now limits data to the year 2018. With these things in place, I can now create the partitioned view. Notice that it follows the outline I started with at the beginning of this module. There are two SELECT statements connected with a UNION ALL set operator. All columns are selected from each table. You can easily imagine how this might grow as new business years are added. Considering that some regulations require seven years of data, there could in time be seven SELECT statements here. I'll add some data so that I have something to query. Now I can query the partitioned view. First, let me query the whole thing. All four orders are returned. Now let me limit the query to just the year 2018. Only two rows are returned as expected. Something interesting, though, is how SQL Server handles these queries. Here are two execution plans produced using the new partitioned view. The first plan where the query specifies the partitioned view without conditions concatenates the two tables together. The second plan, which limits the results to those orders from 2018, only looks at one table. The other table has been pruned away. This pruning is responsible for the performance advantage offered by partitioned views. To prove to yourself that this works, change the query to return only the year 2019, and observe the results.

Designing and Updating Partitioned Views

You might be thinking that setting up a partitioned view is a little complicated. Well, you're right. If you want to use partitioned views, you should design your tables with partitioning in mind. That means thinking about the partitioning scheme you want to use. It could be time or date oriented like I used. That's a popular choice. But it could also be by order ID range if you like, or anything else that makes business sense. I think you can also see that as Bob's business grows year by year, this view from the demo will change. New tables will be added for new years. This could be done by renaming the table for one year and creating a new one for the next. Then adjusting the view. Or you

could create a number of tables in advance to reduce the yearly effort. If you take that approach, though, you will probably be creating stored procedures to insert into the right table based on the order date. Though encapsulating logic like that in a stored procedure makes lots of sense for many reasons, it is more code to test and maintain. Is there a simpler way? Though I haven't discussed them in this course, partitioned tables offer many of the same features without creating multiple tables. On the other hand, you can't partition a table across servers the same way you can with views. Also, maintenance of partitioned tables is really no simpler than that for partitioned views, though the two are quite different. But what about updating partitioned views? You can update partitioned views with INSERT, UPDATE, and DELETE statements, but several conditions apply. An INSERT statement must supply values for all the columns in the view even if the underlying tables have columns with defaults specified. If you are inserting or updating data, you cannot use the default keyword for a column value, even if the column has a default value defined in the corresponding member table. It may seem redundant to say this, but any values supplied for the partitioning column must satisfy one of the constraints. And since the constraints need to identify disjoint sets, such a value cannot satisfy more than one constraint. When updating a partitioned view, you cannot update an identity or a timestamp column. There are a few other minor conditions that you may actually never encounter involving triggers and bulk inserts. See the official documentation for all the details.

Summary

In this module, you learned some of the motivations for partitioned views and how to build them. You also saw that there are many requirements and restrictions placed on partitioned views. I talked about the partitioning column constraint and how it is used by SQL Server to determine which of the underlying tables to look at to satisfy a query. I also showed you how SQL Server uses the partitioning columns to prune tables from the execution plan. In the demo, I built the partitioned view for Bob's Shoes order table partitioned by year. For that demo, I added a new column to act as the partitioning column based on year. Now can you think of another way this could be done without

adding a new column? Hint: Think about the OrderID column. Could anything be done with that? Can you change it so that the primary key is once again a single column? I noted that building partitioned views can be a little complicated and will likely require ongoing maintenance depending on the business usage. Lastly, I talked about additional restrictions for modifying partitioned views. And though I didn't go into this advanced topic, partitioned views can be distributed among several servers, quite a powerful concept. However, as you might expect, there is a set of additional restrictions placed on distributed partitioned views. This brings us to the end of the major topics to be covered in this course. In the summary module, I'll recap what we covered and talk a bit about what we haven't covered, especially several advanced table types.

Summary

Introduction

Congratulations! You made it through the course, Designing and Implementing Tables and Views in SQL Server. The purpose of this course was to give you the essential tools you need to succeed as a database designer and developer. If you have mastered the material here, you are well on your way. In this final module, I want to summarize the topics that I've covered and point out some directions for future study. If you haven't already guessed it, I only just scratched the surface on designing tables. There is a lot left to explore. Plus, with every release of SQL Server, more features are added to fit the needs from small to large to world-spanning, big data projects. Before I get ahead of myself, though, here is what we covered.

Designing Tables

After a brief review of relational databases and the seminal contributions of E. F. Codd and others, I launched into a discussion of how tables should be designed in SQL Server. Using a working example of an order processing system, I began with creating a simple, many would say simplistic, table design. That led to a discussion of what data types are suitable for use. Because Transact-SQL is a strongly typed language, it is imperative to choose the right data types early in the design process. Changing types sometime later can be frustrating, costly, and cause application code to break. Avoid that by making good data type choices at the beginning. Though we looked at some of the most commonly used data types, there are others available for special purposes. Oh, and did I mention, you can create your own. I'll come back to that in a few minutes. After building a single table order system, I noted that it was inefficient and prone to inconsistencies. That led to a discussion of normalization. With your help, if you worked out the exercises, we transformed our database into third normal form or 3NF. This is probably the most common normalization you will find in the wild, though there are other forms starting with Boyce Codd normal form or BCNF, sometimes called

normal form 3.5. Then I briefly discussed other forms up to 6NF. Even that is not the end. There are other forms for special purposes and some still under discussion. Once the database was 3NF normalized, I talked about the six types of constraints available out of the box including PRIMARY, UNIQUE, and FOREIGN KEYS, CHECK constraints, DEFAULT constraints, and NULL/NOT NULL constraints. I left one out, the CONNECTION constraint that came in with SQL Server 2017 for graph databases. No doubt more will be added as SQL Server continues to transform to an all-encompassing big data platform. From my experience, constraints, especially CHECK constraints, are underutilized. They can prove invaluable during a development cycle as a way of verifying assumptions about your data and can help keep your data squeaky clean in production systems. Use them.

SQL Server Tables View Designing Implementing

After we designed, normalized, and constrained the tables for the working example, I discussed views. First, I showed how simple it is to create a view from an existing query and discussed the importance of the WITH SCHEMABINDING option. Next, I talked about the requirements and restrictions for defining views that can be updated. As you learned, there are many things you can do in a view that will stop you from updating it. The key takeaway is that SQL Server must be able to map an INSERT, UPDATE, or DELETE operation to columns of a single table in the view. Modifying multiple tables in the view is not permitted. Indexed views come with performance-enhancing possibilities by persisting the view to disk like any index on a base table. Once again, there are several requirements and restrictions. The last type of view we looked at is a partitioned view. This is a powerful view that can bring together several tables for easy querying. Again, there are many requirements and restrictions, but also great opportunities. Among other things, a partitioned view can actually spin several SQL Server instances, something not possible with its table-based counterpart, partitioned table.

Considering Other Table Types

There are several other table types not covered in this course. Let's look at some of them. Tables can be partitions to produce a high-performance table-based analog to partitioned view. They are common in very large databases, for example, in data warehouse applications. You can partition a table by dates and other values the same way you can partition a view. Maintaining partitioned tables is a special art that requires a deep look to understand and apply properly. Columnstore tables turn data storage by 90 degrees. Actually, I should use the official name, columnstore indexes, since that is what they are. However, considering that a table is either a heap or a clustered index, once you put a clustered columnstore index on a heap, it is no longer a heap. The thing about columnstore is that data is stored by column rather than by row. A SQL Server AK page for a columnstore index will hold data for just one column. This allows for much greater compression than standard data compression and is ideal for data warehouses and real-time operational analytics purpose. Graph tables are new in SQL Server 2017. As the name implies, they allow you to realize a graph in SQL Server. Previously that required very careful table design and very clever programming. Memory-optimized tables help improve the performance of OLTP applications through efficient memory-optimized data access. These tables are resident in memory and can also be persisted to disk. A temporal table is a special kind of system version table with built-in support for returning data stored in the table at any point in time. External tables allow you to read data from Hadoop using SQL Server and Transact-SQL. These arrived with SQL Server 2016 and PolyBase and are indicative of the future direction of the system. These are the major table types that are out of the scope of this course, but I fully expect this list to grow.

Looking at Other Data Types

If there are a lot of table types left to learn, there are at least as many data types that haven't been covered in this course. Let's look at some of them. Float and real datatypes are available for storing imprecise numeric values and measurements. Do not use them for financial data, however, where precision is required. The binary and varbinary types can literally store anything you throw at them. The first has a fixed length. The second is variable. Some common uses are to store images like JPEGs or PNGs, audio files like MP3s and hashed passwords. These types are stored as a series of bytes or byte arrays, free from collation and code page considerations. You can easily convert between other types and binary types with a few exceptions. Rowversion is a data type that exposes automatically generated, unique binary numbers useful for versioning table rows. This type is also known as timestamp, but that is a bit confusing since it is only an incrementing number and not a date or a time. The hierarchyid data type is a variable-length, system data type. Use it to represent position in a hierarchy. Note that a column of type hierarchyid does not automatically represent a tree. It is up to the application to generate and assign values in such a way that the desired relationship between rows is reflected in the values. The uniqueidentifier type stores a GUID or globally unique identifier value. They are guaranteed unique when set with the new ID and new sequential ID functions. Use the XML data type to store XML data. It can be untyped or typed by specifying an XML schema collection, which is a special object containing XML schema definitions or XSDs. There is extensive support for XML data in SQL Server and is a mini topic all on its own. A spatial type's geometry and geography are used to store planar, Euclidean coordinates and round earth real-world coordinates along with a set of methods to manipulate them. User-defined types can hold anything you can imagine that can be implemented in a .NET language. They can be compound consisting of multiple data types and storage structures and have custom behaviors. For example, you could use a UDT to hold all the quantum properties of a particle with all those spooky quantum behaviors. Now although I'm writing this course in 2018, if you are reading it a few years later,

chances are there will be more types added. As usual, consult the official documentation and other Pluralsight courses to get in-depth information on them.

Finishing Strong!

Well, we've covered a lot in this course. Yet there is so much more to learn. Pluralsight has other great courses that can help you to continue along your learning path. I hope you enjoyed this course on Designing Tables and Views in SQL Server. My name is Gerald Britton, and I hope to see you again soon on Pluralsight.com.