Course Overview

Course Overview

Hi everyone. My name is Tamara Pattinson, and welcome to my course, Combining and Filtering Data with T-SQL. I've been involved with database management and programming for over 20 years, and I'm a former Associate Dean of Computer Science. So I enjoy teaching, and I have a passion for data, which is why I'm so excited to bring this course to you and help you begin your journey learning how to use the Transact Structured Query language. Some of the major topics we're going to cover are how to set up the SQL Server environment, work with Management Studio, creating databases using scripts and backup files, formatting strings, working with characters, dates, and numeric values, querying data from multiple sources, and understanding primary and foreign keys, how to filter and apply logic to control the execution of your queries, aggregation, set operators, and how to use common table expressions and intermediary tables. I'm also going to introduce you to user-defined functions that can take the work out of repetitive tasks. And because queries are only as good as they are accurate, this course content will always focus on data integrity and best practices. All of the demonstration queries have been included in the exercise files so that you can follow along, gaining hands-on experience as we progress through the course, I'm going to provide you with a real-world example of how queries work in a web-based table. By the end of this course, you'll have the knowledge you need to write queries for reporting, software applications, and data analytics. From here, you're going to be prepared to dive into the more advanced T-SQL techniques of creating views, stored procedures, and user-defined functions. I'm not expecting you to have any prior knowledge of programming or the structured query language. So if you're just starting out, this is a great place to begin. I hope you'll join me on this journey learning T-SQL at Pluralsight.

Setting up the SQL Server Environment

Version Check

Introduction

Hi, I'm Tamara. Welcome to Combining and Filtering Data with T-SQL. This course has been updated for the latest versions of SQL Server Express and SQL Server Management Studio. My goal is to teach you how to effectively apply the T-SQL functions, logic, and expressions that are used to manipulate and analyze data. Designed to take you beyond the basics, the focus is to provide you with the skills you're going to need to work with data whether you're writing queries for reporting, software application development, or data analytics. Knowing how to write queries is a good start, but quality and accuracy of the results is equally important. So this course places a lot of emphasis on how to test for data integrity issues and how to apply techniques used for validating queries. In each module, I'll demo the topic we're working on, and then we'll build upon these queries as the course progresses. The end result will be queries combining everything we've learned into comprehensive validated reports. And I want to encourage you to download the exercise files and follow along with the demos that I do in the videos because writing queries is the best way to learn the language, and then experimenting with the queries you write will really provide you with an understanding of how T-SQL works. By the end, you'll be able to write your own queries based on your own requirements, validate the results, and identify data integrity issues. Before we start writing queries though, we'll walk through setting up the SQL Server environment and talk about the tools available in Management Studio that will increase your productivity and get you on the way to becoming a power user in Management Studio. So, let's get started.

Setting up the SQL Server Environment

In this course, I'll be using the free version of SQL Server known as SQL Server Express and SQL Server Management Studio, which is the graphical user interface tool used to interact with database objects. And it's in Management Studio where you write T-SQL queries. Express offers all of the functionality for querying the data that the other editions do, and their differences won't impact what we do here. There's information detailing the differences between these versions out on the Microsoft website, and I recommend you look these over for future reference. I've also included a copy of Microsoft SQL Server 2019 edition's data sheet in the exercise files, which also provides a chart of the

different version features. First, we'll walk through installing SQL Server Express and then SQL Server Management Studio. After the environment is set up, we'll explore Management Studio and create the databases used in the course. SQL Server Express can be downloaded from the Microsoft site. So if you go out and do a search for SQL Server Express download on microsoft.com, it will be one of the first results. I'm going to be working on a Windows operating system, but you can also find installs for Linux, as well as a container image with Docker. Once you've downloaded the installer, you'll have a few options for installation types. I'm going to install using the basic installation type and to the default location. When the installation finishes, there's an option for installing the Management Studio tool at the bottom of the dialog. But as with SQL Server Express, you can either search on microsoft.com, or you can follow the link at the bottom of the SQL Server download page under Tools. There are other tools available for SQL Server, and I encourage you to review these, but we won't be using any of these other tools in the course. Download the SQL Server Management Studio installer and follow the prompts. With SQL Server Express and Management Studio installed, let's explore some of the capabilities of Management Studio and set up the two databases.

Creating SQL Server Databases

With SQL Server Express and Management Studio installed, the last step is to create and populate the two databases that we'll use as examples. And we're going to create the two databases in different ways using the most common approaches, first, manually creating the database and executing a script to create the objects and populate with data and, second, creating a database from a backup file. And we'll be using Management Studio to create these databases. When you first install Management Studio, the easiest way to open it is from the Start menu and by typing Management Studio. The first time you run Management Studio, the Connect to Server dialog opens. If it doesn't open or you closed it, you can open it by selecting Object Explorer, Connect, Database Engine, or by selecting the Connect icon. The Connect to Server dialog box gives you options for connecting. First is the server type. And for Server type, by default, this is going to be Database Engine. The other options are for Azure and Reporting Services. We'll be using the Database Engine server type. And the next is the server name. The server name is the instance of SQL Server that you installed. And if you have more than one instance or version of SQL Server, you'll have other choices available, so select the SQL Server instance that we installed. For authentication, I'll be using Windows authentication, and this is

the default when you install SQL Server. We won't be going into troubleshooting if the connection fails, but there's some great information out there that can walk you through some of the steps you can take to fix any issues that you may run into. To review troubleshooting techniques, Microsoft has a lot of information on how to troubleshoot connections, and this is the link to their site. Once you connect, the Object Explorer tree view will display the database objects in the server instance, including the databases of the SQL Server database engine, and this is where we start creating the databases. First is the Northwind database, which we'll create manually and then populate it using a script file. The script is included in the exercise files, which you can download from the Exercise tab on the main course page. In Object Explorer, right-click Databases and select New Database. In the new database dialog, enter the database name, which, in this case, is Northwind. Now, if you decide to name it something else, something you need to keep in mind is that the script we're going to use to populate and create the objects references Northwind. So if you decide to name this something else, then you'll need to go through the script files and change all of the references to Northwind to the database name that you're using. Create the database by accepting all the default values and click OK. Now, if you go back to the Object Explorer and refresh, you'll see the Northwind database is there, but at this point, it's empty. So the last step is to populate and create the objects with the script file. Find the file that you downloaded from the exercise file named NorthwindObjectsAndData, and all T-SQL script files have an extension of SQL. There are several ways to execute a script, and we'll go into this more in depth in the next clip where we explore the Management Studio environment. You can either execute by selecting Execute from the toolbar or the keyboard shortcut of Ctrl+E or by pressing F5. Once the script executes, the message window will appear, and the status bar at the bottom displays some useful information at a glance. If the message on the query status shows success, you're good to go. If it doesn't, you'll need to do some troubleshooting. Refresh the database in the Object Explorer again, and you'll be able to see the tables have been created. This next database we're going to create using a backup file. I chose to use this method as an example because it's a common approach, especially if you're going to be working on a team project with existing databases. In that case, you're going to get a copy of the database. And then usually on restore, you'll be restoring an existing database on your computer, but restore will also create the database from a backup file if it doesn't already exist. Again, on the Object Explorer, right-click on Databases and select Restore Database from the menu. In the dialog, select the device option and then click the button on the right,

which opens a File Open dialog window where you'll be able to locate the backup file that was also included in the exercise files. Select WideWorldImporters-Standard. The window will show the backups being restored. And at that point, select OK to start the restore. Refresh the Object Explorer again. And if you did it correctly, the database WideWorldImporters, along with all of its objects and data, will be in the list. So far then, we have Express and Management Studio installed, and now the two databases that'll be used in the course are created, so let's get started exploring Management Studio and begin writing our T-SQL queries.

Using SQL Server Management Studio

SQL Server Management Studio is an integrated environment for managing any SQL infrastructure. It's used to configure, manage, administer, and develop all components of not only SQL Server, but Azure SQL Database and Azure Synapse Analytics as well. It's a powerful utility, and we'll be using a number of the rich features. I'll show you some of the more commonly used features and focus on those that we'll be working within this course. How you can view table relationships using the database diagrams and working in the query text editors, how to add comments in the keyboard shortcuts. We'll look at the Object Explorer and customizing the environment. I'll show you how to view query execution plans, which show you how your query is going to execute and the time it will take to get it done. Any bottlenecks that you may have will show here. What we cover will be helpful as we progress through the course. It will get you proficient with the environment and how to quickly spot errors as you're writing your queries. Querying data in a relational database is all about pulling information from related tables. Management Studio offers a visual database diagram tool for analyzing these relationships. It's a great place to start when you're becoming familiar with the database. Existing database diagrams can be found in the first database tree node. And if there aren't any, you can create one by right-clicking on the folder and then selecting New Database Diagram from the menu. From there, you can add any of the tables or all of the tables and view the relationships. This is an introduction to the diagram to let you know it's there and that it's available for you to use. In our discussion on primary and foreign keys, we'll create some of these together and go through the process step by step. When I begin working on an existing database, it's the first place I go, and you're going to find that it's an invaluable tool, especially if you're joining a team and there's an existing database and you need to bring yourself up to speed quickly. These diagrams aren't just for viewing

though, and you can create, edit, and delete objects in here as well. So, when you're first starting out, experiment with a local copy. Let's open a query window and add in a sample query. New query windows can be opened by selecting New Query from the toolbar or by using the keyboard shortcut Ctrl+N. Each query maintains its own connection instance, and the connection status and properties are displayed on the status bar at the bottom of the query window. By default, the query connection will be set to the database activated in the Object Explorer. And if you open a new query window without one of these databases activated, then it defaults to the master database. The database used in the connection will be displayed in the drop-down on the toolbar, as well as in the status window. You can change the database from the drop-down or in the query itself, which is a best practice, and we'll be doing that with all of the queries we write in the course. For the sample query, we'll use the editor tool, which allows you to select the tables and fields for the query using a graphical user interface. Open the query editor using either the keyboard shortcut Ctrl+Shift+Q or right-click anywhere in the query window and select Design Query in Editor from the menu. Just as with the database diagram, an Add Table dialog will appear. I know there's a relationship between Suppliers and Products, so for this example, I'll select those two tables. The relationship in the designer window is represented by that same visual view we saw in the diagram indicating that a relationship exists between tables. The bottom window on this designer displays the query framework, which right now is just referencing the two tables that have been added because there have been no fields selected. From the Products table, I'll bring in ProductName and then CompanyName from Suppliers. As the fields are added, the bottom window generates the query based on the selections. Selecting OK will output this query to the query window. I don't use the editor as a rule, but it's a good way to get the example started, and I want you to know it's available. If you're new to writing queries, go in here and experiment. All of this query syntax is covered in the course, and right now we're only going to focus on the query environment and the text editor. The text editor has many of the same features found in other text editors, and so they're going to be familiar. Ctrl+A selects everything, Ctrl+C to copy, Ctrl+X for cut and copy. And then there are my personal favorites, Ctrl+Z to undo and Ctrl+Y to redo. You can add comments to your queries in two ways, the multiline comment, which begins with a /* and then terminates with a */, and then the line item comment, which is two dashes. There are also two keyboard shortcuts, which will add and remove these dashes, and I use them all the time as I'm writing queries and testing my results. Either place your cursor at the first of the line or select the text

you would like to comment out. And then with the keyboard, use Ctrl+K+C to comment and Ctrl+K+U to uncomment. Using these shortcuts adds or removes the dashes, but it won't do anything to the multiline comment, and it won't do anything to the text. It's only adding and removing dashes two at a time. It also only works with dashes at the beginning of the line, and I like to use this kind of comment if I want to add a note about what this particular part of the query is doing, and I'll add the comment at the end of the line. In this way, it isn't impacted with any of the keyboard shortcuts. There are several ways to execute a query. You can either select Execute from the toolbar, use the keyboard shortcut Ctrl+E, or you can execute using F5. If the query executes without errors, the results will be displayed in the Results pane, and the status window will show two new statuses, the query execution success or fail and the number of records returned from the query. You'll have two tabs. The first tab displays the results, and the second tab returns messages about the query progress, how many records were returned, and if there were any errors, it will display the error messages along with the lines that the errors occurred on. I'll change Suppliers to Supplier and run again. This time when the query is executed, there's an error. No results are returned, but there is an error message. The message and the line where the error occurred will be displayed in this message window. Double-clicking on that message will place the cursor on the line where the error occurred. If there's an error in the query syntax, you'll have a visual cue of a red squiggly line, and hovering over the line will give you a message hint as to what the error is. Management Studio has dozens of options for customizing your workspace, everything from font size and color to what happens when you open Management Studio. One of the first I set for Management Studio is adding in the line numbers. To set these options, go to the main menu, select Tools and Options. There's a simple text search at the upper left of this window, and it will return the results where any of the words you entered are part of the description of that option. I'll type in line for line numbers, and it's found under Text Editor, All Languages, and General. What I like about line numbers is that you can see where you're wanting to go and use another shortcut, which is Ctrl+G. This works whether the numbers are visible or not, but the line numbering gives you a visual. And when you're working with an extra long query with over 100 lines of code, this is a great guide. This is one of the final queries from the course, and you can see how line numbers in this would become a nice visual tool because in this case it's over 100 lines long. When you execute a query, the results can be returned to a grid. like we've done here, or a text window, or you can save it to a report file. When you're executing queries, you have an option to view the execution plan, and this

returns the information in a third tab and displays the number of records impacted and returned and any errors that may have occurred. If you have hover the individual sections in the execution plan, it provides a lot of detail that you're not going to see in the two standard windows. It also displays the amount of time as a percentage that each part of the query took to execute and how long the processor took to complete. This tour around the environment has provided you with some tips and instruction on what is most widely used when writing queries. I recommend that you become familiar with all of its capabilities because this is the development environment you're going to be using when querying data.

Summary and Resources

With SQL Server 2019 Express edition installed, as well as SQL Server Management Studio, the database is created. And a first look around Management Studio, our environment is set up, and now we're ready to move on to writing queries. This is a list of references and links that I mentioned in the module. These links will take you to the Microsoft site and provide you with more information if you need to troubleshoot or want to view versioning. Let's begin writing our queries in the next module, Shaping Data in a Query.

Shaping Data in a Query

Introduction

Hello. I'm Tamara Pattinson. Welcome back to Combining and Filtering Data with T-SQL. In the last module, we set up the SQL Server environment and created the two databases that will be used throughout this course. The goal in this module is to get you comfortable writing queries that incorporate data manipulation by implementing T-SQL functions, expressions, and logic. We'll start by overviewing the overall SELECT statement syntax, and then we'll look at concatenation, casting, and converting values to different data types. We're going to be working with strings and characters and begin using variables that we'll use to make our query suitable for ad-hoc reporting, and I'll introduce you to user-defined functions, which can help you simplify repetitive tasks. You'll learn how to format

and calculate dates and numeric values using practical exercises to walk through some of the most common tasks and then working with formatting of those values. We're going to start using the SQL CASE expression, which is a form of logic you can use in T-SQL. We'll also start working with nulls and learning how to handle these in queries. And throughout this entire course, you'll learn how to implement querying techniques that will assure the accuracy of not only the results being returned by your queries, but the integrity of the underlying data because queries are only as good as they are accurate. Let's get started.

The SELECT Query Syntax

Throughout this course, you'll hear me refer to the syntax of a query. And syntax, by definition, is the rules that state how and in which order words and symbols are used together in a computer language. A SELECT query can contain the following elements. SELECT, which is always required, designates the fields to be returned, FROM is the table or tables where the data is located, and GROUP BY is used when aggregation is applied, for instance a count of employees grouped by their locations. HAVING, WHERE, and MATCH. HAVING is used to filter the results when the GROUP BY clause is included. Otherwise, you would use either WHERE or MATCH. These filters are applied to limit the results, for instance a count of employees grouped by their locations having a status of active. ORDER BY sorts the results and can include multiple fields. An example would be a query to return a count of active employees grouped by their locations and then ordered by both office name and then employ name.

Concatenating Values in T-SQL

So exactly what is concatenation? Concatenation is the action of linking things together in a series. And in the case of Structured Query Language, we're talking about combining characters, string values that can come from either values in the database, text you've written out, or a combination of the two. With T-SQL, you have three methods for concatenating. The string method was all that was available prior to SQL Server 2012. And although it's not the recommended approach, it's important that you know how it works because you're going to see a great deal of it in older work. And if you happen to inherit queries, you'll need to know how it works and some of its limitations. Two additional

methods were introduced with SQL Server 2012, CONCAT and CONCAT_WS. Both of these latter methods solve some of the challenges inherent with the original string concatenation method, and each serve a different purpose. Let's look at these three concatenation methods, their syntaxes, and the rules for using them. This first query is implementing all three of the methods, and they return the same result, the first and last names from the Northwind employees table as a single value. The CONCAT method works in much the same way as the string concatenation in that the resulting value will be literal. And if you want additional spaces or text between the values, you have to add it. CONCAT_WS, on the other hand, will add a character in between each value that you only have to specify once, but the limitation here is that there's just a single parameter. And if you want different characters between each of the string values, you'll need to use CONCAT. The string concatenation method has two distinct limitations, first is the way nulls are handled or not handled, null being the absence of a value and not a value itself. And by default, the string method will return null if any of the concatenated fields are null. You can see in this example that the string concatenation method returned null across the board, even though only the first value was null, where both CONCAT and CONCAT_WS methods removed these challenges by converting null to an empty string. If you run across this problem where the string concatenation method is being used, you can handle the null challenge using a system setting. The system setting is CONCAT_NULL_YIELDS_NULL. And if you set it to OFF, then the null value is ignored. A word of caution here. The system setting will stay off until you reset it to on, so that's something you just want to keep in mind. The second limitation with the string method is that you have to explicitly convert any non-string values first to keep the query from returning an error. So with all the challenges that come inherit to the string concatenation method, why then would you need to know how to work around these limitations and not just update to one of the newer methods? Because if you ever find yourself in a situation where you need to support existing work that uses the older method, if it isn't going to be a quick fix and you have time limitations, you can use this system setting and come back to it later when you have time for housekeeping and updates. You may have noticed that there aren't any column names when you concatenate, so you need to add alias names for these columns. This is a good habit to have because column names are required with temp tables. Throughout the course, we'll be using aliases and talking about them more, so this is just a first look. So in summation, concatenation means linking string values together. There are three methods available. Prior to SQL Server 2012, string concatenation was the only method. Still

used and common to see an older queries, but has challenges for null and non=string values. CONCAT and CONCAT_WS resolved these challenges by evaluating the field and converting to string values. Alias column names are a best practice, and they're required when using temp tables, and we'll see more of this in the module Simplifying with Intermediate Tables. Now that we've covered concatenation and written queries using all three of the methods, let's move on to working with casting and converting data type functionality available in T-SQL.

Casting and Converting Data Types

Knowing how to CAST and CONVERT are important skills you will need to master because converting data types is used in every aspect of data manipulation. We've talked about how concatenation requires character values and how CONCAT and CONCAT_WS methods CONVERT non-character values as part of their functionality where the string concatenation method doesn't provide it. The reason is that the original string concatenation method uses the plus operator, which is also used in mathematical computations. The processor attempts to CONVERT character values to numeric values when there are mixed data types in the concatenation statement, and this returns and unable to convert data type error. Converting data types is used to format dates and times, as well as in mathematical calculations. Null is the absence of value. And if you're working with calculations where there's a possibility that some of the values could be null, in order for the mathematical calculations to be accurate, you would use CAST. When we review joins, I'll demonstrate joining tables on fields with different data types using CAST. Also, when you're preparing data that will be used in reporting, you'll run across instances where you're asked to return values in different data types. This isn't an ideal situation, but it does happen. There are four functions available for converting data types, CAST and TRY_CAST and then CONVERT and TRY_CONVERT. TRY_CAST and TRY_CONVERT have the same syntax as their counterparts and offer error-handling capabilities. The CAST and TRY_CAST functions have two parameters. First is the field or value to be altered, and second is the data type you want to alter it to. If it's an allowed conversion type, the value returned will be the value altered to the new data type. There's an option to return a certain length if the data type being converted supports length specifications. The TRY_CAST function will return null instead of an error if the conversion fails. CONVERT and TRY_CONVERT offer the capability to change the data type in length, but they also have a third optional parameter, which is used to apply additional formatting. And it's this option that

makes CONVERT the powerhouse. As with TRY_CAST, TRY_CONVERT has the same syntax as its counterpart and will return null if the conversion fails. However, both TRY_CAST and TRY_CONVERT have limitations to their error-handling capabilities. The reason for these limited capabilities is that some data type conversions are explicitly not permitted. This is the SQL data type conversion chart provided by Microsoft, and it outlines the data type conversions. You can locate this chart on the Microsoft site, but I've also included a copy in the exercise files. In this chart, you'll find explicit conversions, and it's these conversions that cannot be done under any circumstance and where TRY_CAST and TRY_CONVERT will still return an error if the conversion type is explicitly not allowed. The terms CAST and CONVERT are used interchangeably, but there are distinct differences between the two. Where CAST as the ANSI standard and supported by all relational databases, with the exception of some of the older versions, CONVERT is not standard, but it is supported by some of the more popular platforms. Where CAST is the ANSI standard and universally supported, the best practice is to use CAST whenever possible and only use CONVERT when you're sure the platform supports it. Let's write some queries using these functions. The GETDATE function returns the current database system timestamp as a date time value, and I'll use this GETDATE function in these examples. This query uses GETDATE to return the current date as a variable character data type. Without adding the optional length parameter, which datetime data types do allow, the result is the abbreviated month, day, and year and the current time. This is the same query, but the option of length has been changed to 11, which means only the first 11 characters, including spaces, are returned, which resulted in the time value not being included in the result set. In the case of datetime, we can be fairly certain that a datetime would support a length of 11, returning just a day as it did here. However, CONVERT has that third optional parameter. And using CONVERT with this optional parameter returns the day in different formats and not just the archaic approach of hoping 11 characters will work. CONVERT offers dozens of predefined formatting options that you can take advantage of, and we'll dive deep into these later when we explore formatting dates and numerical values. Some data type conversions won't return an error because they are allowed, but they will return invalid results. This next query is using CAST and CONVERT to alter the current date to a data type of money. No error is returned because the data type conversion is allowed, but you need to be careful when selecting the conversion data types because some of the conversions return invalid results. Where conversions are not allowed, such as changing text to numeric values, the TRY_CAST and TRY_CONVERT methods

will return the null we talked about earlier. These methods don't return errors if the values can't be converted as is the case here where they just return null. I use TRY_CONVERT and TRY_CAST in all of my queries to handle any unexpected errors. You may find that a query will work as expected when it's written, but the data types of the fields in the table can be changed. If this happens and it's an incompatible data type, you could begin to see errors if the conversion type isn't allowed. Using TRY_CAST and TRY_CONVERT will give you at least a level of error handling if something like this does happen. This last query demonstrates what happens when you attempt to CONVERT a data type to an explicitly not permitted data type. Date times are on the Microsoft chart listed under the explicitly not allowed conversion to a data type of money. And this is the example of where even TRY_CAST and TRY_CONVERT won't be able to handle the errors. So, we've discussed the similarities and differences between CAST and CONVERT, how to determine which of these is a best practice to use, and we've examined queries demonstrating each of these functions, talking about the limitations of error handling, as well as some of the conversions and how they may return unexpected results. We're going to be exploring a lot more on the formatting functions when we talk about formatting dates and numerical values. But now, let's start on formatting strings.

Formatting Strings

I'm going to be starting our formatting strings discussion by demonstrating how to complete a common task, and that is how to apply proper casing to unformatted values and concatenate them into a single field separated by a comma. I'll show you how to achieve this using TRIM, SUBSTRING, and LENGTH functions, and then I'll apply UPPER and LOWER to return the values with proper casing. With the names formatted, I'll apply the concatenation to return the results in a single field and add an alias as we did with concatenation. As we progress through the course, we'll fine-tune this query, adding additional functionality to handle some more of the formatting challenges. I've created a copy of the Employees table and named it EMPLOYEES uppercase, altering them to a mixture of upper and lowercase characters and some extra spacing as well. So to give you an idea of where we're going, this is the starting point. And then our end result by the time we're through will return the first and last name with proper casing concatenated last name, first into a single column named FormattedName. There's a copy of this script in the download files if you'd like to follow along. But what I've done here is put each step onto this same script, and then I'll comment it out as we go

through and then uncomment the next step so that you can see them in order. Step 1 is to remove the trailing and leading spaces from the first and last name fields using TRIM. Then, we'll need to isolate the individual characters so that we can apply upper casing to the first character and lower casing to all the remaining characters using SUBSTRING. With the characters isolated, we'll apply UPPER to the first character and then LOWER to the rest. With the first and last name fields formatted and in proper casing, we'll use CONCAT to put them together in a single field with Lastname, Firstname. So let's start with step 1 to remove the leading and trailing spaces using TRIM. There are actually four versions of the TRIM function. There's TRIM, which simply removes all of the trailing and leading spaces from both sides of the expression. There's LTRIM, which removes only the leading spaces on the left. And then there's RTRIM that removes all the trailing spaces. And then there's another optional version of TRIM, which allows you to remove characters, as well as spaces, and I'll demonstrate how to use that version in the next clip when we start dealing with characters. This query is returning the first and last name as is and then, in the same SELECT statement, using TRIM to remove any trailing or leading spaces. TRIM takes a single parameter and only works with string values, which FirstName and LastName fields are. This has to be the first step in the process, and you'll understand why after the second step where we'll apply SUBSTRING to isolate the individual characters so that the UPPER and LOWER functions can be applied. SUBSTRING is used to return a part of an expression, which can be a character, binary text, or image. The star is an integer value and designates what part of the string you want to start at, 1 being the first character, then the length is a positive integer, and it specifies how many characters of the expression you'd like returned. If the length parameter is longer than the string expression, then the characters from the expression are returned and not anything additional. The return type is based off of the data type of the expression. It can be either varchar, nvarchar, or varbinary. Because the first and last name fields are varchar, then our return type will be varchar as well. Let's look at some examples of SUBSTRING and then apply it to our query. I've added SUBSTRING to the query and encapsulated the First and LastName fields inside a TRIM function so that any trailing or leading spaces will be removed first. When I'm running queries like this, I like to open a new query window and use it as a scratch pad. I'm going to be using a hardcoded value, the word hello, rather than anything from a table so that we can concentrate on the SUBSTRING method. Hello is the expression, the starting position is 1, and I only want to return that first character, so I'll set the length at 1 as well. Executing the query returns just the letter h, which is

what we would expect. But a lot of the name fields have leading and trailing spaces. And space is a character, so if we don't use TRIM, then the first character returned will actually be a space. We want the first letter, so the expression needs to be encapsulated inside a TRIM function to get rid of any leading or trailing spaces. Adding TRIM now returns that first letter h. This is what we'll be doing when we isolate the first characters of the first and last names so that we can apply uppercasing. In this select query, the SUBSTRING is starting on the second character, which will return all, but the first character of the expression. The length is still set to 5, which now returns all but that first character of the word hello. You can add any value for LENGTH, and so this one is using 5,000, which is much longer than the expression, and it's going to return the right value. The LENGTH function returns an integer value representing the LENGTH of the expression. Instead of adding a hardcoded value like the 5,000, I can use this LENGTH function as the third length parameter. And by doing so, we're assured that all of the characters will be returned. Going back to our query, I've encapsulated Last and FirstName fields inside TRIM and then used the SUBSTRING to return just the first character and a length of 1. So executing the query is going to return just that first character that we're going to be applying the UPPER function to to make sure that all of the first characters are capitalized. And so this next query does the same thing, but it starts at the second character. It uses the LENGTH function and returns the last name and the first name, beginning at the second character, which is getting us close to the proper casing. We're ready to move on to the third step and apply the UPPER and LOWER functions. UPPER and LOWER take a single string parameter and then return a string value as either all upper or all lower. This query is combining the functions from the last two queries that we did, returning all of the SUBSTRING values. I'll apply UPPER to the first character and then LOWER to the SUBSTRING that's returning the remaining characters, and I'll repeat this step for both the first and last names. The formatting is complete, and all that's left is to apply the concatenation and return the final result. I'm applying the CONCAT method we worked with earlier when we looked at concatenation, wrapping the formatted values inside the CONCAT method and combining back the first and last names, which are now in proper casing. We're only working with two fields, and you can imagine how this repetitive process could quickly become tedious and very prone to errors. In the next section, we'll continue exploring different formatting techniques with characters, and I'll introduce you to the temporary table, as well as an introduction to user-defined functions, which are saved queries

where you can add a lot of this functionality and then just call it instead of having to write this out every time.

Working with Characters

Where a string is a group of letters, numbers, or symbols, a character is a single letter, number, or symbol. And there are times when you will need to isolate a single character located in a string. I'll start by introducing you to the user-defined function and how you can take a complicated query, move the repetitive tasks into a custom function that can then be reused and called from any query. Then, taking what we've learned about string manipulation, I'll show you how to implement functions that isolate and interact with individual characters. We'll use the PRINT command to check our work and validate the query results. And I'll also introduce you to WHILE and IF, which is covered in detail in the module Filtering and Controlling Flow. Here's the query we wrote, returning the employees' first name and last names formatted with proper casing. This entire query is only formatting two fields, and you can imagine how complicated it could quickly become if you were formatting, say, 10 fields in the same query. This query is returning the same result set, but it's calling a user-defined function I created named Proper, which factors out the functionality we used to format the name fields. It takes all the work we did using LOWER, UPPER, SUBSTRING, LENGTH, and TRIM, and then returns the formatted value for us. User-defined functions in detail are outside the scope of this course. However, just as with stored procedures, it's important that you know what's available and then look into them later on. There are other courses in this path that do cover these functions completely. These functions are found in the database node , programmability, functions, and then either table or scalar. Scalar returns a single value, and this is where the proper function is located. Let's step through this query inside the function comparing it to the query we wrote. The function takes a single parameter, and this is the field we want returned with proper casing. With our query, the variable is going to use that LastName and FirstName field. Then, inside the function, two other variables are declared. The first will hold the value returned once the query has processed, and then the second is going to be set to that first letter. Step 1 is to set the entire string value to LOWER, which isn't something that's possible in a SELECT query. Step 2, using SUBSTRING to isolate the first character, and then UPPER is applied to set it to uppercasing. Then, using CONCATENTATION, SUBSTRING, and LENGTH, the entire string is combined. The variable formatted value is set to this value and then returned to the

query. Taking all of the functionality, adding it to your user-defined function, and then calling that function is what allows us then to just place the variable, either First or LastName here, as a parameter inside Proper and then returning the strings with proper formatting. Sometimes though, you'll have strings containing characters that should not be included, and you'll need to isolate these characters, remove them, and then format. I've created another copy of the Employees table, naming it EmployeesExtraCharacters and added some very unfortunate formatting, not only a combination of upper and lower values in spacing, but also added in a mixture of alphanumeric and symbol characters. We're going to take these fields, apply the functionality we've looked at so far, and then adding additional functionality, we'll return them again in proper casing. First, we'll use TRIM. We've worked with TRIM to remove trailing and leading spaces, LTRIM to remove anything from the left, right to remove anything from the right. Then, there's this fourth TRIM variation that allows you to remove the spaces and characters. This TRIM variation takes two parameters. First is the group of characters you are removing, followed by the keyword FROM, and then the expression or field to be evaluated. I've created a variable and set it to a string that includes leading and trailing spaces, asterisks, periods, and ampersands. Using this TRIM variation, adding in all of the characters to be removed from the beginning and ending of the value, removes these unwanted characters. So, this is the first step using that TRIM function to remove trailing and leading spaces, as well as any unwanted characters. But this only removes characters and spaces from either side. We'll need to use other functions to remove the unwanted characters located within the string, and we're going to look at three functions that can be implemented to remove these ampersands. First is REPLACE. REPLACE takes three parameters. First is the value or field expression to be evaluated, second is the pattern to be searched for, and third is the value that you want to replace this pattern with. This is a great function to use and one that you're going to use often if you need to replace a specific value contained within a string. If you want to do more than that, you'll need to isolate the values and then deal with them using SUBSTRING and CONCAT, and we'll be using just that in the final query. Second is CHARINDEX. CHARINDEX returns the position of a character within a value. The function returns a BIGINT if the expression to search has an nchar, varbinary, or varchar data type. Otherwise, it returns an INT data type. The syntax has two required parameters, and then it has a third optional parameter, and it's this third optional parameter we're going to be using to get that final proper casing in our query as well. First is the expressionToFind, second the expressionToSearch, and third is this optional parameter,

which is used to find where inside the expression to start the search. And then third is PATINDEX. Very similar to CHARINDEX, it also returns a BIGINT if the expression is of the data type varchar or nvarchar. Otherwise, it returns an INT. PATINDEX requires two parameters. First is a pattern to be searched for, and then second is that expression or field to be searched. The difference between CHARINDEX and PATINDEX is that PATINDEX can use regex expressions or a group of expressions to be searched and also search for characters not be included in the list by adding in a caret at the start of the pattern. Let's put all of these together and return a list of values with proper casing. This value contains an array of characters, and we want to eliminate anything that isn't either an alpha, hyphen, or single quote. We're leaving those in because names can be hyphenated, and they can contain single quotes. First, we're going to use TRIM because even though we want hyphens included in the string, we don't want them at the beginning or end. Then again, using WHILE as we did with the user-defined function proper names, along with PATINDEX and an expression that will remove anything not alpha, not a hyphen, and not a single quote. Let's walk through this in detail because there's a lot going on here. I'm using another variable with a mixture of alphanumeric values, hyphens, and spaces. This parameter is going to look for any character within this string. Because the pattern has included a caret, PATINDEX is going to look for any character that isn't alpha, hyphen, space, or single quote. With single quotes because the single quote is also used to reference variable characters, they have to be handled a little differently. You have to refer to these single quotes using two single quotes and then enclose the single quotes inside single quotes, which means that to represent a single quote requires four single quotes. As long as PATINDEX finds anything outside this parameter, it will equal a number greater than 0, indicating the position within the string that the character was found. Then, it sets the value of the variable equal to the position of the character, last using SUBSTRING to isolate the first section beginning at 1. Subtracting 1 from the position will exclude the character and then another SUBSTRING, which increases the position by 1, skips that character. The WHILE loop will continue to execute as long as any of the characters found are not included within the parameters that have been set, and then the final value is returned. So far, so good. All of the unwanted characters have been removed. Then, applying UPPER, LOWER, LENGTH, and CONCAT, the value is returned in proper casing, sort of. Because of the hyphens and single quotes, we need to add proper casing to not only the first character, but the first character following either the single quote, hyphen, or space. And this is where we're going to need to use CHARINDEX

because we need to take advantage of that third optional parameter. Using PATINDEX would just create an infinite loop because PATINDEX would always find these characters, and we don't want them removed. We're going to iterate through each character in the string individually because we need to apply upper casing to the first character after a hyphen, space, or single quote and then using if logic. After you've reviewed the module Filtering and Controlling Flow, I recommend coming back to this query and reviewing this functionality again. The purpose here is just to show you how these functions can work to manipulate string values. If the single quote is found, then the variable holding the position is set to the position of that single quote. Otherwise, if it contains a hyphen, then the variable of the position is set to that hyphen location. If it doesn't find a single quote or a hyphen, it's going to check for a space. The position of the variable is used to isolate that next character that should be capitalized, which means we don't want the exact position of the character. We want the next position, which represents the position of the letter to be capitalized using SUBSTRING and UPPER. And then CONCAT combines these together with that letter capitalized. Then, we'll increase the starting position used in that third optional parameter of CHARINDEX by 1 to move on to the next character until the entire string has been searched. With this query now, we can place it inside another user-defined function and call the query anywhere within queries. Now just by calling this user-defined function proper with characters, adding the LastName and FirstName fields, all of those unwanted characters are removed, and proper casing is applied. This is a lot of information, and the best thing you can do is to experiment with them and using PRINT command review the results. There are so many scenarios you'll be using these functions with, and I've introduced you to the tools. And that's the most important part, knowing what tools are available. Another important part of manipulating data is working with numbers and dates. So let's move on to the functions, which allow you to manipulate datetime and numeric values.

Formatting Dates and Numbers

No matter what aspect you'll be using T-SQL for, dates and numbers will be a constant part of data analytics and reporting. I'm going to be demonstrating some of the most commonly used functionality and then point you to additional resources. GETDATE is used to retrieve the current date, so we'll start with that because it will be used in most of the examples. Then, we'll work with CASE statement using ISNUMERIC and ISDATE and DATEDIFF. Building upon that, using DATEPART, DATEADD, and

DATENAME, I'll show you how to write queries with formatting that will answer questions such as how long did it take in order to be shipped, and was it shipped within the company's acceptable range, and how long has an employee worked for the company? And then using products, we'll format the current price of the product into currency. This is one of the final reports from the course, and we've worked with much of the functionality already, such as variables, concatenation, and converting. So we'll build upon that and add date numeric functions that are also used in the query. Once you've completed the course, you'll understand how to create a report like this one because we're going to cover all of the functionality used in the query. So let's get started by first exploring the CASE statement and the ISNUMERIC and ISDATE functions. There are two variations of CASE. First is the simple CASE, which evaluates a single condition, translating into something like either/or. When it's true, it will produce the value located in the first section. Otherwise, without any additional logic, it will use the second part of the statement to return a value. There are two required parts. First, you need to close the CASE statement using END. And second, the field must be assigned an alias. We've used aliases before, but they weren't required. However, with CASE statements, they are. The second variation of CASE is a search CASE expression. It also requires the keyword END and an alias name. This CASE variation can evaluate up to 10 conditions. But if you find yourself reaching that point, then you should consider some of the other logic you can use, and we'll be talking about those later on in the course. Unlike the simple CASE variation that works as either/or, this CASE expression will enter the first WHEN block if the expression being used to evaluate returns true, even if the other expressions evaluated to true. Let's look at some query examples. CASE statements must be contained within a query. So beginning with SELECT and in CASE WHEN, an expression is evaluated. Here, I'm using 1 equals 5, and so the ELSE block will execute because 1 does not equal 5, and then the query will return 1 does not equal 5. The else part of CASE WHEN is the catchall, and ELSE can be used with this simple variation of CASE, as well as the search variation we'll look at next. And then to close CASE, you use the keyword END and an alias field name, which completes the CASE statement. This query is using the CASE search expression, checking to see if 1 is greater than 0, which it is, so it evaluates to true. And 1 is greater than 0 is returned in the result set. But notice here that I have another WHEN checking to see if 1 is equal to 1. That is true, but with CASE, the first evaluation to return true wins. And so once it reaches a true value, then it exits the CASE statement. Many of the queries we write are using GETDATE for examples, and GETDATE returns the current database

system timestamp. GETDATE is used in reporting, but generally it's formatted to month, day, year. FORMAT can take three parameters. The first two are required, and the third is the culture argument. If that third optional parameter isn't provided, the language of the current session is used. These two queries are examples of the available cultures, and they aren't just used for formatting dates. Here, using GETDATE as the first parameter, which is the expression to be evaluated, and then adding the month, day, year pattern enclosed in quotes because it's requiring an nvarchar data type, leaving out the optional culture parameter returns the current date as either an nvarchar data type if the expression to be evaluated is not null. Otherwise, it returns null. FORMAT supports over 20 different data types, and we're only looking at date in numeric. Search for FORMAT on the Microsoft website to review a complete list of the data types supported and some additional formatting options. CONVERT also provides formatting functionality. And when we talked about casting and converting, I showed you an example using CONVERT. And as with formatting, a search on the Microsoft website for CONVERT will provide you with a list of possible built-in formatting options. And I've provided you with a few other examples of converting and formatting in the exercise download. The ISDATE and ISNUMERIC functions take a single parameter and return an integer value, either 1 if evaluating for true or 0 if evaluating for false. This query is using ISDATE to see if the hire date in the Employees table is a date value and then a CASE statement returning Hire date is available if it's a date; otherwise, returning the wording No hire date. And I've also added in the concatenation and that user-defined function we wrote to return the employee's name formatted as last name, first name. This query is using ISDATE, CONCATENANCE, and DATEDIFF to return the employees and how long they've worked for the company. It's using the search CASE variation to evaluate for three possible outcomes. First, if there isn't a date, second using DATEDIFF if the employee has worked for the company less than a year, and third using ELSE, which means that neither the first or the second evaluations were true. It will return the year difference between today's date and the employees hire date, then another simple CASE statement, also checking if the hire date is a date. If it is, then formatting the hire date. Otherwise, returning the wording Unknown. Let's examine the DATEDIFF function. DATEDIFF has three parameters. First is the interval, such as day, month, or year, second is the from date, and third is the to date. Using GETDATE, these three queries are returning the number of days, years in nanoseconds until January 1st, 2060. This query is using the Orders table, bringing back formatted shipping and order dates and then the DATEDIFF function to determine the amount of time between

the order date and the ship date. This is the same query, but using two CASE statements. First, if an order date isn't there, it prints no order date. Otherwise, it shows the formatted order date. And because CONVERT returns a string value, there's no additional conversion that's needed. The second CASE is checking for the ship date. If it's null, it's returning 0. Otherwise, the date difference in days between the order date and the ship date. Because DATEDIFF is returning an integer, I'm using a 0 so that I don't have to convert. Northwind likes to have all of their orders shipped within 14 days, and they need a way to determine what date it must be shipped by to meet that goal. Also, if the 14th day falls on a Saturday or Sunday, they want the due date moved to the following Monday, and this is going to require two additional functions, DATEADD and then DATENAME. DATEADD also takes three parameters, just as with DATEDIFF. The first parameter is the interval type, second is the amount to be added, and then third is the expression to be used. And it must be either a date data type or a value that can be converted into a date data type. The value returned is the datetime format, so you'll most likely want to format that as well. DATENAME returns a character string representing the DATEPART from the first parameter. The second parameter is the expression to be evaluated. This query is using DATEADD to increase the order date by 14 days, and then it's using DATENAME to determine if the weekday is either a Saturday or a Sunday. If Saturday, it increases the order due date by 2. And if Sunday, it increases it by 1. Let's end with taking a look at formatting currency and a user-defined function. This user-defined function returns the calculated order total. The standard mathematical order of operation rules also apply to T-SQL queries. It's using the orderId as the variable and SUM, which is covered in aggregation. The asterisk is the notation for multiplication. So first, it's multiplying the unit price by the quantity and then subtracting any discounts. Then, the value is returned to the query. These three queries are all using that function, first without any formatting and ordered by the total order amount in descending order with the largest order listed first; second, with formatting. But notice that the ordering isn't right . When you're using formatting in an ORDER BY, the results returned aren't going to be accurate. To get formatting, you add the formatting to the field in the SELECT statement, but use the numeric equivalent in ORDER BY. That's it for date numeric values. There are so many variations to these functions and so many applications for these that you should consider this a good start, but I encourage you to continue reviewing all of the functionality. Use these examples and these links for further studies.

Review and Resources

That's it for Shaping Data in a Query. We've covered a lot of material in a short amount of time, and I recommend you go over this material, write out your own queries, and experiment with a combination of these different functions. Using these functions will become second nature to you, and we'll continue using all of these as we progress through the course. There are 24 examples that you can use as a reference. All of them begin with M3 and are included in the exercise file download. In this next module, we'll begin writing queries pulling in data from multiple tables. Using the various join types available, we'll continue talking about data integrity and how to validate that the number of records returned are accurate. So please join me in this next module, Combining Data from Multiple Sources using Joins.

Querying Data from Multiple Tables Using JOINS

Introduction

Hi, I'm Tamara. Welcome back to Combining and Filtering Data with T-SQL. Up to this point, we've been using a single table to write queries using the T-SQL functions, expressions, and logic. In this module, we'll explore primary and foreign keys and how these keys are used to create relationships between tables. We'll build upon what we've learned so far and begin combining data from multiple tables using each of the join types and build upon what we've learned in the previous modules to test for data integrity and query validation. I'll start by demonstrating how to create database diagrams, which is Management Studio's graphical user interface where you can view and create relationships between tables. From there, I'll demonstrate combining data from multiple tables. And because queries are only as good as they are accurate, quality counts, and I'll show you the techniques I use to perform data integrity and query validation checks.

Understanding Primary and Foreign Keys

Before you begin writing queries using joins, it's important to understand how relational databases work and how the tables in a database are connected or related to one another. Management Studio provides a graphical user interface to view relationships through the use of diagrams that visually display the table relationships. This diagram shows all of the tables in the Northwind database and how they're related to one another. If a relationship exists between tables, there will be a connecting line between them. Let's create a diagram with just a few tables. The first folder in the database's tree view is named Diagrams. Right-click on that folder and select New Database Diagram from the menu, and a blank diagram along with an Add Table dialog will come up. If the Add Table window is closed, you can reopen it by right-clicking on the database diagram and selecting Add Table from the menu. The first join type demonstration, inner joins, will use the Orders and Order Details table from Northwind, so I'll select Orders and Order Details from the Add Table dialog. This connecting line means that there is a connection between these tables, and on either side of the connecting line are icons. The key next to the Orders, OrderId field means that each record is a unique record and, in this case, represents a single order, and that same order will not be added more than once to this table. OrderId is a primary key because it's a unique value, and no two orders will have the same ID. But a single order can contain many products that were purchased in that order, and the Orders, OrderId primary key in Orders is included in Order Details to connect the purchased items associated with that particular order. The infinity icon is next to the OrderId field in Order Details because there's no limit to the number of products that can be added to a single order. And when a primary key is used as a reference in another table, it's called a foreign key. This type of a relationship is an example of a one-to-many relationship, one order, many products included in that order. You can view the properties of a relationship by right-clicking on the connector and then selecting Properties from the menu or by selecting F4. The Properties window has a line item called Tables and Columns Specifications, and expanding this will give you the details of that relationship. You can also view table and field properties by clicking on them. There are a lot of useful things you can do in a database diagram, including editing and creating relationships, creating and editing tables and fields. But two of the most useful things I've found are the ability to add related tables and to arrange tables in a diagram. Right-click on a table and then select Add Related Tables from the menu, and all of the tables from the database with a relationship to that table will be added in the diagram. When you're getting familiar with a new database, this is really helpful. These tend to pile up on each other, and you can drag them around, or

Ctrl+A works like it does in many other applications and will select all of the objects. Once they're selected, if you right-click on any of the selected tables and then select a range selection from the menu. Ctrl+- will zoom out and Ctrl++ will zoom in. I'll close the diagram and save it as Order - Order Details so we can come back to it later and add more tables as we look at joins. Let's explore inner joins using the Order and Order tables.

Using the Inner Join

Inner joins return records from two or more tables where the field being used in the join exists in both tables. It's the default type of join, and when just the keyword JOIN is used in the query, an inner join is implied. This is the diagram we created for Orders and Order Details, and this is the query that will return all of the related records from these two tables by joining them on that OrderId field, returning all records and all fields from both tables where OrderId exists in both tables, and it's doing this by using this statement JOIN ON Orders OrderId = Order Details OrderId, and I'm using the alias o for orders and od for Order Details. Keep in mind though that even though an ordered item won't exist without a related order, an order can be created before items are added. If you're looking to find all orders, even with orders that don't have related items, that's going to require a different kind of join. It's going to need either a left or right join, and we'll be looking at those next. So, use inner join only if you're wanting to work with records that exist in both tables. Let's bring back something more meaningful in the query. From the Orders table, OrderDate, ShippedDate, and ShipAddress and, from Order Details, Unitprice, Quantity, and Discount. This error, Ambiguous column name OrderId, means the column name OrderId exists in both tables. It's ambiguous because the processor is unable to determine which OrderId field is being referenced. Any fields referenced in a query that are named the same thing in both tables must include a table reference to specify which fields should be used, even if they aren't primary or foreign keys. If these fields are not specifically being referenced in the query, then you don't need to do anything. That's why this first query using just the wildcard asterisk didn't return an error. The alias name assigned is recognized by the processor. In fact, the processor no longer recognizes the table named Orders because o has been assigned as the alias name. But adding the alias name reference to OrderId now tells the processor which OrderId to use. This is the same query with the formatting for the dates and calculations we did in the last module, and this query is working, but it's not very informative and I'd like to see the product name returned as well, and product name is

in the Products table. Going back to the diagram that we put together in Understanding Primary and Foreign Keys, I'll add in the Products table. Order Details and Products are connected through the Products primary key ProductId and the ProductId foreign key in Order Details. So, before we can get the product name, the Products table will need to be included in the query using this ProductId in the join. Joining products in Order Details on the ProductId now, the product name can be added into the query. This query is only returning records where ProductId is in both Order Details and Products. But what about products that haven't been purchased? It's possible to have products that have never been included in an order. I'll show you some better ways later on to check for validation once we cover nested SELECTs in WHERE clauses and COUNT from aggregation. But for now, these two queries will work. They're returning the ProductId from their respective tables. ProductId is unique in Products because it's the primary key, but Order Details can have many of the same product. So by adding the keyword DISTINCT, then the results will only return each distinct ProductId. The query returns 78 products in the Products table, but using the distinct count for product IDs in the Order Details returns 77, so one of the products has never been ordered. This isn't wrong if you're wanting to work with just the orders and the Order Details. However, if you wanted to answer the question how many of each product has been ordered, then one of the products wouldn't be included in the results. So to return all products, including products that haven't been purchased, we'll need a different type of join, a join that returns all of the records from a table, even if there are no corresponding records in the join table, and these type of joins are called left and right joins. Let's rewrite this query to return all products even if they've never been purchased and include Order Details if they've been purchased.

Left and Right Joins

Where an INNER JOIN returns only records that have matches in the field being used in the join from both tables, the LEFT JOIN returns all of the records from the first table referenced and matching records from the table being joined. But no matter what, all of the records from the first table, the table on the left, will be returned. This is the query we left off with joining Orders and Order Details where we added in products, but one product had never been ordered, and so it wasn't included in the results because there wasn't a ProductId associated in Order Details. Let's rewrite this query using LEFT JOIN so that all of the products can be brought in. Because products is the primary focus for this query and I'm wanting to analyze product data and not just order data, I want to include any data from

Order Details in orders if it exists. But even if it doesn't, I still want to bring back all of the products. I'll put the Products table first because it's my main focus. Then, I'll add in Order Details and use a LEFT JOIN because that's the table that has the relationship to products, and that relationship is through the ProductId primary key in Products and the ProductId foreign key in Order Details. Then, I'll add back in Orders using a LEFT JOIN as well. This LEFT JOIN is going to return all of the products. This product, Chocolate Covered Pickles, has never been ordered. And I know this because I've ordered by the OrderId field in Order Details, and null is returned in the fields where the corresponding data doesn't exist in join tables. But, look what happens if I change this to our RIGHT JOIN. Now the product is missing, and that's because on a RIGHT JOIN, I'm saying bring back everything on the right, Order Details, and include anything from the left table if it exists. So where left returns all of the records from the table on the left, right returns all of the records from the table on the right and any corresponding data from the table on the left. I could write this using a RIGHT JOIN and making Order Details the first table, but that doesn't show me right off that Products is the main focus for the query. And it's better to start with the table you most want to focus on and then add in the other tables containing any extra data you want brought back in subsequent joins. Time for a validation check. I know there are 78 products because a simple SELECT query from Products returned this many. I'm not concerned at this point with Order Details, but I can test this query by just adding DISTINCT to the ProductName and keeping this join the way it is. I want to leave the join this way because what I'm testing is the join. I'll comment out the other fields, execute the query again, and I know all of the products are being returned. We'll come back to this query and look at some of the other validation tools available later when we work with aggregation and filtering. But for now, let's look at full joins.

Full OUTER Joins

The FULL OUTER JOIN returns all records from all the join tables, and this type of join is really useful with data analytics. This is the query we ended with in LEFT and RIGHT JOINS, and it's returning all products, even those not purchased along with the data for Order Details if the product has been purchased. Let's go back to the Orders, Order Details diagram. I'll add in the Products table because we last looked at products along with the order data if it existed, but there's another case that could conceivably occur. There could be an order started with no items added. An order doesn't require a product to be added, which is when it would also have a record in Order Details. And we know that a

product won't be in Order Details if it hasn't been purchased. So, what about the situation where you want to return all products, purchased or not, and all orders, even those without items added? For this, you're going to need a FULL OUTER JOIN. This query returns all records from Products, Orders, and Order Details, and I'm using the alias table name and asterisk to return everything. There are nulls in the result set, but some of these could be legitimate in fields that allow null. However, we know the OrderId and ProductId are required. This is the same query, but it's returning just three ID fields from each of the tables. Let's review what the result set is telling us. In this first record, ProductId from Products, is the only field returned. It's null in Order Details, which means it's never been included in an order. And because there is no order, both OrderId fields from Orders and Order Details are null, which we would expect. And this represents our product that's never been ordered. This next record though only has an orders OrderId. There's also no Order Details record associated with the order, and that means that the order does exist, and one was started, but no items have been added. This type of query generally isn't used for end user reporting and, in the as is condition it is now, doesn't provide any information apart from what the state of the products and orders are, so its benefit isn't analyzing the data, but it can be more useful in a formatted query. This query is using case logic to return something more meaningful. It's returning text to quickly identify what the field being null means. It's taking a few steps to get there, including case logic, some casting and formatting. We'll continue with this query when we use COUNT to return the number of products ordered and the number of items in an order when we look at aggregation. But now, let's look at a common join type, the SELF JOIN. We'll talk about when you need to use it, and then we'll write some queries together.

Self Join

SELF JOIN is where two instances of the same table are joined together in a query. The best way to understand SELF JOINs is to understand the reason for SELF JOINs. Let's walk through three different scenarios of where this would come into play. We've reviewed the relationship lines in the diagrams before where one table is related to another table. This is the Employees table from Northwind database, and it has a relationship linking back to itself. In this case, the EmployeeID field is linking to a field in the table itself named ReportsTo, and this ReportsTo field contains the EmployeeID of the employee's manager. A manager is an employee, and their employee information won't change when they're a manager. So, you don't need an entirely different table in the database

just for managers when the only reason to do so would be to identify a manager. This concept is called database normalization, and it's the process of reducing data redundancy where the same data shouldn't be kept in multiple tables. All that's required here is a single field that can store the manager's EmployeeID. Here's another example. This is the Northwind Customers table, and it contains a SELF JOIN where the primary key is Customer ID, linking it back to the ParentCompanyID. Here again, all of the information for a company would remain the same whether it was a parent company or not. So, you wouldn't need a completely different table just to keep the parent company information in it when a ParentCompanyID could be used to return all of the information from the same table. So, this is an example of linking a hierarchy of companies or people, reps, something like that into the same table. The key here is that all of the information would remain the same whether they're a parent company in this table or a manager in Employees. This is the People table from World Wide Importers and another example of why you would want to join a table back to itself. This field LastEditedBy is just updated with the ID of the person that last updated the record. So, even though the information is being kept for a different reason, again, the common thread is that you don't need an entirely new table just to find the person who updated the record. Let's write a query using the Employees table to return all of the employees' information along with their manager's name. I'll start out with a simple SELECT query because I want to see how many employees there are in the table so that while we're in the process of completing the JOIN, I don't inadvertently leave any out. Because we want to return all employees and there are 10 records, this is what we're going to be looking for. We need 10 employee records to be returned. The ReportsTo field can be null, and knowing what we know about INNER JOINs, if I were to join Employees to another instance of Employees and use just a JOIN, I'm not going to return all of the records because one of the ReportsTo fields is null. So, I know I'm going to need a LEFT JOIN because I want to bring back all the employees from the first instance of employees, which is on the left side of the JOIN statement. We also have some errors here, and that's because from this point on, we have to use alias field and table names because we're referencing two instances of the same table, which contain identical field names. When you're naming these instances, try and use something more descriptive than e1 and e2. I'm going to use emps for the first instance and mgrs for managers as the second instance. Then, the fields I'm going to link on are emps.ReportsTo and mgrs.EmployeeID. This time when I execute the query, I have all 10 employee records returned, so I know I'm good to that point. I'll keep the asterisk, but I'm going to start

putting in the fields above it to begin formatting the query. The first field I want to bring in is EmployeeID, but the EmployeeID of the manager, not the employee, which means I've got to use the manager's EmployeeID. I also want the employee's EmployeeID, so I'll put that in next. Then, I'll execute to make sure I still have all 10 records. So far, so good. EmployeeID is returned in the result set as the label for the first two fields, and that's not what I want. So, I'm going to add an alias name to the manager EmployeeID and rename it ManagerID. Run this again, and the alias names represent what we want returned in the result set. This is a formatted report where I'm bringing in the employee ID, the employee name, and then the manager they report to. Using a CASE WHEN statement, if ReportsTo is null, then the result set will return no manager assigned. Now, this is a simple query, and I've done that on purpose because I want the focus to be on this JOIN type and not on the formatted report itself. Here's another example using the World Wide Importers People table. Again, I've started out by bringing all of the records in, and I've done a JOIN on people, giving it an alias name of People and a LEFT JOIN of People giving it the name of editedBy. And I'm joining it on people.PersonID = editedBy.PersonID. Now, really all I want here is to return all the information from people, so I'll use people.* wildcard to bring in all of the fields from people. And I only want the editedBy.FullName, and I've created the alias of LastEditedBy. So, here's two examples, one to bring back the employee's manager in the result set and another to bring back information on who last edited a record. Let's write a query using the Customers table using a SELF JOIN to bring back all of the customer's record and also to include the parent company name. Here's the skeleton query where I'm using the asterisk and I'm selecting FROM Customers, giving it an alias of c and then doing a LEFT JOIN to another instance of Customers with an alias of pc for parent company. But now adding the alias name c in front of the asterisk means that I only want the information returned from the first instance, which will represent our customers. And then right before this, I'm going to add in pc for the alias and then the company name, and the result returns all of the customer records and fields, along with the parent company name.

Review and Resources

That's it for this module. We've looked at primary and foreign keys and the different types of joins. Joins aren't hard, but they are where you can get yourself into trouble if they're not done right. It can leave records out or bring them in multiple times. Remember to check your results and review the

record numbers. Later on in the course, I'll show you how to use the COUNT method, which is another way to check the number of records in a data table and make sure you aren't leaving any out or bringing too many in. Practice writing joins, change the join types, and look at the data returned. Use the database diagrams as visual aids before working with any database you aren't familiar with. And remember, databases aren't created the same. And so, first things first, know the structure and relationship layout of the database you're working with. Joins are one of the most important parts of a query when it comes to the accuracy of your queries and data integrity. This is a list of the corresponding query examples used in this module, and they're located in the exercise file download. In the next module, we're going to look at filtering the record sets and applying logic to control what the processor does given a set of circumstances you assign. I hope you'll join me as we look at filtering and controlling the flow using logic, the expression and comparison operators, and the logical operators available in T-SQL.

Filtering and Controlling Flow

Introduction

Hello, I'm Tamara. Welcome back to Combining and Filtering Data with T-SQL. My goal in this course is to take you beyond the basics of the SELECT query and show you how to use the T-SQL functions, expressions, and logic you'll need to incorporate whether you're writing queries for reporting, application development, or data analytics. In this module, you'll learn how to apply filters using WHERE to further refine the query results. We'll start with an overview of the syntax and then explore the various expressions, comparison operators, and how to add multiple filters using the AND and OR parameters. Then, picking up where we left off with some of the other queries we've written to this point, we'll add these filters to return only the records matching the parameters set in our queries. We'll talk about nested SELECT queries that can be added inside the WHERE clause and query optimization concerns you need to be aware of when using nested SELECT queries. And along the way, we'll discuss the important role these filtering techniques play when analyzing data and validating query results. After completing this module and combining with everything else we've learned in the

course, you'll know how to write the query that would be used to populate this web page table, and you'll also understand how the filtering is achieved.

Expressions and Comparison Operators

We'll begin by reviewing the WHERE clause syntax and then using some of the queries we've written. I'll add in filtering using quantifiers. And then, I'll show you how to write nested SELECTs, which are SELECT queries inside WHERE clauses. Last, I'll take the orders and customers queries we wrote using a LEFT JOIN, finding orders with no associated products and customers who hadn't placed orders and rewrite them using a filter to bring back just those orders without order items and customers who have never placed orders. This simple query is returning all orders from the Orders table and ordering by OrderID. 832 records are returned, which represents that total number of orders placed with Northwind. Using WHERE to filter and the equals comparison operator, we can filter to bring back just one record. The WHERE clause goes directly below the SELECT statement, and I'll use the OrderID 10260. And executing again returns just that one record. Changing the equals to not equals returns all records, but that record. Because this field is an integer data type, I can use the comparison qualifiers, returning any orders that have an OrderID above 10260. And then greater than or equal to will include that order, as well as any that have an OrderID higher than 10260. Less than returns any with an ID below 10260, and less than or equal to returns that order and any other orders with an OrderID less than 10260. This is the query we wrote using a LEFT JOIN and then ORDER BY to find orders that had no associated products. We can filter this by using a WHERE and the IS NULL to return only orders without associated products, no longer relying on ORDER BY and then searching manually for null. Doing the opposite, we can return orders with only associated products by changing the IS NULL to IS NOT NULL. Using IS NULL and IS NOT NULL is a best practice, and you should stay away from implementing the equals and not equal to comparison operators because they can return unexpected results, and that's because of a system setting. This system setting is anti-nulls, and it specifies an ISO-compliant behavior of the equals and not equal to comparison operators when they are used with null values in SQL Server. ISO is the International Organization for Standardization. And if you're not familiar with this organization and their driver for standardizing, I recommend you look into their mission because ISO standards are very much a part of not only best practices, but your everyday life. These simple queries will provide an example of this setting and how it impacts

evaluating for null. With ANSI_NULLS OFF, equals and equal to work. When they're on, they won't work. And using IS and IS NOT NULL will work, even with ANSI_NULLS ON. ANSI_NULLS doesn't just impact an entire field, but using anything for an evaluation where that value could be null will return null. If you need to filter where null may be involved, you can use the comparison operator, as well as a logical operator, and we'll be looking at the logical operators in the next clip. This is another query we wrote, again using a LEFT JOIN. And we knew, because of the LEFT JOIN, that they had never placed orders because the OrderID field was null. Again, we can use WHERE and NULL to filter customers that haven't placed orders using WHERE OrderID IS NULL. But this can also be written another way, and that's by adding a SELECT statement as part of the filter, and this is called a nested query. The value you use in a SELECT statement does not have to be in the SELECT part of the main query. It can be any field from the tables in the main SELECT query. There are two ways to do this. Just as IS NOT NULL and IS NULL, you can use an IN or IS NOT IN for filtering. This SELECT statement is just returning the CustomerID from Orders. Then, the values returned in this query results are used by the processor to filter based on either IN or NOT IN. Nested SELECTs can be heavy on processing, and this is because it's querying the database multiple times. Let's run these two queries side by side and review the execution plan. With the nested SELECT statement, the processor had to complete multiple steps and accounted for 59% of the query cost compared to 41% of the query cost using IS NULL. This may not seem like a lot, but we're only working with 832 records. Imagine what the processing overhead would be when you're working with hundreds of thousands of records. Whenever possible, the best practice is to use another approach, such as IS NULL or IS NOT NULL as in this example. Or later on, we'll look at additional ways such as with common table expressions and temporary tables. But nested SELECT queries are commonly used. And if you find yourself supporting existing queries, you may want to review to see if these can't be rewritten to optimize performance. When filtering with strings, the value used in the expression requires a single quote. This example is using WHERE to return records where the last name is Davolio. Using not equal returns all records, but Nancy Davolio. Greater than, when used with a string, returns the result of anything alphabetically that would go after Davolio. Greater than or equal to returns Davolio, as well as any records that would be alphabetically listed afterwards. Less than returns all records alphabetically above, and less than or equals to returns all records from A to Davolio. You don't have to put in exact values. This example is using LIKE, which we'll discuss in detail in the next clip, and it uses a wildcard,

which is the percent sign. This example is returning any records where the last name begins with D. Working with datetime values can get interesting. I'm going to update Nancy Davolio's hire date using GETDATE, but GETDATE inserts date and time values. Now running a query using GETDATE in the filter doesn't return this record, and that's because it's looking for values that match exactly. I can run SELECT GETDATE multiple times, and each time I'll get a different value because of those time value. It's a common task to need to bring back records with the current date value. And to get accurate results, you need to use the CAST function, casting both the field comparing to, as well as the expression. CONVERT works too, but because CAST is the ANSI standard, using CAST is the best practice. Looking at this web page, you can now envision what the query must look like that populated this table. Notice the 832 records. This is the query that was used to populate the table. If I search on 10268, then the order with that ID is what's displayed in the table. Searching on around the horn returns 13 records for the customer around the horn. And then searching just on oms will return any items in the table where the company name contains oms and it's using that LIKE logical operator. In a true use case example though, these values aren't hardcoded, and they'd be returned as a request from the user interface, and then the filtering would be done using that variable. This variable searchVal is a varchar, and you can use a varchar on an integer data type because the WHERE clause will convert for the comparison. Once you've completed the module Limiting the Results with SQL Functions, you'll also know how the number of records to show at a single time and the pagination works. But now, let's dive into the details of using LIKE, as well as other logical operators available to filter query results.

Applying T-SQL Logical Operators

In this section, you'll learn how to add logical operators used in conjunction with expression operators to further define the result sets returned by a query. Logical operators test for the truth of the conditions, and they return a boolean data type with a value of either true, false, or unknown. AND returns true if both boolean expressions are true. OR returns true if any of the boolean expressions are true. LIKE uses the percent sign as the wildcard, and it returns true if the operand matches a pattern. NOT LIKE is the reverse of LIKE and returns records that do not match any of the conditions BETWEEN returns true if the operand is within a range of a condition. NOT can also be used with BETWEEN. Just as NOT LIKE, it returns anything not within the parameter. Let's look at some

examples. Using the Employees table, we wrote a query where the results were filtered to return any records where the last name was Davolio. This query is using the is equal to with the logical operator AND, which means that the results will contain employees with the title of Sales Representative and the city of London. Both conditions must evaluate to true before the records will be included. This is returning three records that match the criteria, the title of Sales Representative and city of London; whereas, this query is using the OR logical operator, returning any records where the title equal Sales Representative or the employee city is London and the result set contains seven records that meet this criteria. The first record is returned because the city is London, even though the title is not Sales Representative. The second has the title Sales Representative, but is not in London. With OR, only one of the conditions needs to be true. Logical operators are often combined together to further refine the results, and the combination can include any of the logical operators. This query has expanded the filtering using both AND and OR. All conditions still must be met before the record will be included in the result set. The records must either have the title of Sales Representative and the city equals London or not a Sales Representative where the city is Tacoma. And four records match the criteria. Going back to the queries we looked at earlier, finding employees by last name, this query is filtering using not equal to in conjunction with the AND logical operator, and it will return any employee records where the last name is not equal to Davolio and is not equal to FULLER and is not equal to Leverling. Out of the 10 employees, the results will return all but these three records. You need to be careful on combining logical operators. Let's look at this example. Using not equal to with the OR logical operator is returning all 10 employees even with the filter, and this is because using OR means that only one of these conditions needs to be true. When the value is Davolio, the not equal to is true. But using OR instead of AND, it's included because the other two expressions, not equal to FULLER, not equal to Leverling, will be true. And then the same will happen when the processor is evaluating for either FULLER or Leverling. Because OR only needs one of the conditions to evaluate, this query would always return all of the records. There's another way to write these queries, which are considered to be a best practice. That's using either IN or NOT IN. IN is a shorthand equivalent to using WHERE is equal to because you can include the values to be evaluated inside parentheses. NOT IN is the equivalent of WHERE IS NOT equal to. Again, using the parentheses, you would add the values to be used in the filtering inside the parentheses. You don't have to write hardcoded values, and these typically use a nested SELECT, which would return a record set that would then be used to evaluate

for which records should be returned. By now, you're familiar with these first two queries. Selecting OrderID from Orders returns 832 orders placed with Northwind. Using SELECT DISTINCT returns 830 orders. This query is using NOT IN to return only orders that have an OrderID in Order Details using a SELECT statement to return the OrderIDs and then filtering using NOT IN. And then the opposite, OrderIDs that are in Order Details, returns 830 for a total of the 832 orders. This query is returning all records where the last name alphabetically is between Davolio and Leverling, including the two values because of the greater than equal and less than equal expression operators. This same result can be returned using BETWEEN, which evaluates based on a range of values. In this case, any values between Davolio and Leverling, including Davolio and Leverling, results in the same record set. BETWEEN requires the smallest value to be first. Switching Davolio and Leverling will result in no records. NOT BETWEEN returns anything outside the range, also requiring that the smallest value be first. Using DATEPART, DATAADD, GETDATE, and two variables, this query is filtering the Employees table, returning a record set containing any employees hired between these years, including these two year variables. If the years are reversed with the highest value first, the result would always be an empty record set because BETWEEN always requires that the smallest value be first. We can also use the logical operator LIKE and its counterpart NOT LIKE, which uses an asterisk as the wildcard. These examples are returning products using LIKE and NOT LIKE with the percent sign as a wildcard, and this query is returning any product with a product name beginning with C and then the opposite using NOT LIKE to return any product not beginning with C. The wildcard can be used anywhere in the variable. Adding it in front of ing returns any records where the product name ends in ing. And again, the opposite using NOT LIKE returns everything that doesn't end in ing. Using the percent wildcard on both the beginning and ending of the value results in a record set containing anything here with the letter G, and this will also include any records beginning or ending with the letter G. Using NOT LIKE returns any products that do not contain the letter G in the product name. Going back to the table on the web page, we can see how LIKE is being used. Adding ome to the search returns any customers with a company named containing ome. And this is the query we could write using the Company table to achieve the same results. We've looked at the expressions and the logical operators, and now it's time to start applying logic used to control the flow of processing queries.

Controlling Flow with T-SQL Logic

You think every day with logic. If it's cold or windy, wear a coat. If there's a good movie playing, go to the movie, otherwise read a book. And keep washing dishes while there's still dirty dishes. With T-SQL, you can use logic constructs that work in the same manner to control the flow of execution. I'll be demonstrating how to apply this logic to control the processing flow and refine the results, drilling down to a granular level, beginning with the variations of IF and THEN, writing queries that will iterate through a section using WHILE statements. I'll also show you how to use GOTO, which is used to set the point at which the execution will be redirected. Let's start with the simplest form of a logical construct, and that's IF. IF tests for true. When the expression used is true, then the query inside the beginning in block will execute. This query is using two variables, num1 is 5, and num2 is 10, and then the statement evaluates these two variables. And if num1 is less than num2, the section inside the BEGIN and END blocks will be executed. Let's write some queries using IF. This first query is like the example from the slide, using two variables num1 and num2 and then setting them equal to 5 and 10. The IF statement is evaluating using the less than comparison operator. And if true, which is the case here because 5 is less than 10, a message is being printed to the message window. Anything outside the BEGIN and END block is executed regardless of the logical if construct. BEGIN and END aren't required if there's only a single line that should be executed. However, it's definitely a best practice, and you can see how it would easily cause bugs if left out. Including the BEGIN and END block also greatly increases the readability. Executing the query returns the first message TRUE, 5 is less than 10, and the message not included in the block outside the if statement. This query is using the same variables, checking for num1 to be greater than num2 and also using the comparison operator OR, which means that either num1 must be greater than num2 or num1 must be equal to num2. Neither of these expressions evaluate to TRUE, and so the statement inside the block won't be executed. Changing the greater than to less than and executing does return this message because with OR only one of the expressions is needed to evaluate to TRUE. But there's a better way to write this, and that's using the less than or equal to comparison, and this is a best practice. Any time you can simplify a query, returning the same results also increases the readability and reduces the margin of error. Using IF allows for only one scenario, but the logic construct if/else allows for additional control of the flow. The block inside the IF will execute when TRUE, and we've looked at that. But by using ELSE, the block inside ELSE will execute when IF evaluates to FALSE. This allows you to control the flow by providing a way to execute something only when it's true. And then rather than just proceeding with

the next statement in the query, when ELSE is followed by IF, you're assured that one of these blocks will execute. Next is the IF IF/ELSE ELSE block, and this allows for testing for multiple cases and then executing the block that returns true. With this construct, the first true statement will execute, and then the flow moves outside the construct going to the next statement in the query. Even if one of the following sections evaluates to TRUE, it's important to remember that the first expression evaluating to TRUE will execute. So just be aware that you could have unexpected results and review the logic. You'll also find situations where you aren't necessarily going to want the query to continue executing right outside the logical construct. And when this is the CASE, you can use a GOTO statement to further alter the flow of what happens at a certain point. Here we're using the GOTO label, which can be located anywhere within a query. This is evaluating for num1 and num2, checking to see if they are numeric values and that num2 is greater than 0. If these conditions are met, the num1 is divided by num2 using this GOTO command and then defining a label. Here, the label name is CalcValue. So if the IF portion evaluates to TRUE, then the processor will skip down to the CalcValue label. It's also going to execute PrintMessage because these labels don't stop the execution, and this is something else you want to keep in mind. This is the same query, but I've added another GOTO right below CalcValue, which then is going to skip that PrintMessage. There's one last variation of the IF statement, and that's IIF. Unlike the other IF constructs, IIF can be used inside SELECT statements. And essentially, it's another way of writing the CASE statement we've been using in the course. It's a shorthand version of CASE, and it's actually converted to a CASE statement by the processor when it executes. The difference is that where CASE allows for the use of CASE WHEN, which is like ELSE ELSE/IF, IIF is like the ELSE IF and allows for either TRUE or FALSE without the nested ELSE IFs. The IIF statement syntax requires the expression to be evaluated followed by what should happened when true and then what should happen when false. These two queries are returning the exact same results, the first using IIF and then the second using CASE. So if you'd like to use a shortcut, use IIF because it doesn't require that you name the field. But when you need to evaluate more than one scenario, use CASE. Just as with IF, the WHILE construct uses a condition which must evaluate to true. As long as the condition is true, the processor iterates or repeats through the section of the query that's included inside the BEGIN and END block. We have an integer data type variable num with an initial value set to 0. This WHILE construct is evaluating num using less than. And as long as num is less than 10, the processor will continue to repeat through the BEGIN and END block. Once the value

of num is 10, it will exit the block. When there isn't anything to change the expression being used to evaluate, the processor won't exit the WHILE construct, which means an infinite loop has being created, and it will just continue to run. This expression 1 = 1 will always be true and will continue running until the time limit set in Management Studio is reached. The amount of time that lapses before timing out is a setting in SQL Server, and that's going beyond the scope of this course because it's an administration setting. However, you need to be aware of infinite loops and really test your queries before placing in production. BREAK is used to exit a while loop before the expression evaluates to false. And with BREAK, the processor will exit the WHILE construct. Let's look at CONTINUE, which, unlike BREAK, doesn't exit the WHILE construct, but what it does is jumps back up to the start of the BEGIN block. And you can also nest constructs inside constructs. Still using 1 = 1 as the condition, however, inside the WHILE statement, I've added an IF/ELSE construct where IF is checking for num be less than 5. The original value is 1 and then, at this point, num = 2, so IF evaluates to true because num is less than 5. Now the BEGIN block inside the IF construct will execute. First, it increments num by 1, so at this point it equals 3, prints the value of num, prints continue, and then the CONTINUE statement is executed. Because CONTINUE is executed, the processor will jump back to setting num equal to num + 1. It prints the value of num, and num still is less than 5, so it will enter the BEGIN block, increase num by 1, print the value, print continue, and then CONTINUE. So now we're back to SET num = num + 1. This time though, num isn't less than 5, which means the block inside ELSE will execute, num is increased by 1 again, and then BREAK is called, which means the processor will exit the entire WHILE construct. All of these examples are included in the download files, and I'd encourage you to go in and experiment with these values. Use the PRINT command to review what's happening during the different stages of processing. One of the most common places WHILE is used is with cursors, which we're going to be covering in detail in the module Simplifying with Intermediary Tables and Common Table Expressions. But for now, let's go back to our web page and learn how the number of records to be shown in the table and the pagination work using offset and fetch and then how to return the top number of records using numeric values and percentages.

Review and Resources

And this concludes our discussion of filtering queries and using logic to control the flow of execution. You know how to use expressions and comparison operators and how to filter out for null, what nested SELECT queries are, and some of the optimization concerns. We're going to come back to this when we work with common table expressions and temp tables and how those can be used instead of these nested SELECT queries, using filtering to analyze for data integrity and how to identify records that do not have corresponding data in a related table. You've learned about T-SQL logic operators and how to combine them to create compounded logical statements and how to control the flow of execution using T-SQL logic and commands. We've taken our first look at how filters work with tables, as well as ordering and some of the challenges you'll face in different software environments. This is a list of the corresponding query examples used in this module and located in the exercise files. In this next module, we'll take up where we left off with the web page table and begin working with pagination, which is how the paging works, and learn how to bring back a limited amount of records from a query. Please join me in this next module, Limiting the Results with T-SQL Functions.

Limiting the Results with T-SQL Functions

Introduction

Hi, I'm Tamara. Welcome back to Combining and Filtering Data with T-SQL. My goal in this course is to take you beyond the basics of querying data and show you how to effectively apply functions, logic, and expressions used to manipulate and analyze data with a focus on data validation and best practices. In the last module, we learned how to control the flow of execution in a query using logic and how to filter the data. We looked at how a table on a web page displayed data and how filtering is applied. We're going to begin by revisiting our web page, learning how the number of records are returned using the FETCH and OFFSET functions, and I'm going to be talking about the challenges that you'll come up against regarding sorting on cross platforms. After learning how to use OFFSET and FETCH, I'll show you how TOP n and top % are used to drill down in the data, answering questions such as who are our top five customers and what are the top five best and worst-selling

products? I'm going to introduce you to the COUNT aggregate function because it's one of the most commonly used functions when analyzing for top results. Later on in the course, we're going to dive deep into this function along with the other aggregates in detail. I'm also going to be demonstrating how to use variables in our queries where you only need to change their values to return variations of the result set. So let's get started with this module, Limiting the Results with T-SQL Functions.

Offset and Fetch

This is our web page with the table populated from the Northwind customers and their orders, and we've learned how the filtering in order works. The number of records shown at a time, as well as pagination is accomplished using the OFFSET and FETCH T-SQL functions. This is the raw query that was used to populate the table with all of the 832 orders being returned. It's using a JOIN instead of LEFT JOIN because we're only interested in customers that have placed orders. The table on the web page is set to show 10 records at a time, and paging through the table contents called pagination then returns the next set of records with a number of records being returned, determined by the number of records selected in the drop-down. To achieve this result, you'll use the OFFSET and FETCH num functions. OFFSET determines the beginning point, and this is the number of records to be skipped. Setting it to 0 means that it will start with the first record. This syntax is OFFSET, the number of rows to be skipped, followed by the keyword row, then the FETCH NEXT, which limits the number of records to be returned. With OFFSET set to 0 and FETCH NEXT at 10, the results returned from the query are the first 10 records from the query results. Selecting the second page on the table and keeping the number to show at a time at 10, skips the first 10 records and shows records number 11 through 20. I want to take a minute to talk about ordering. Ordering isn't approached the same on all platforms. How SQL orders the results isn't necessarily going to be how another application or, in this case, a table on a web page orders the results, and this has to do with the way they treat alpha, character, and numeric values. I'm mentioning this to you because you'll run across this usually reported as a bug where they want to know what the difference is. After you validated that you're querying is working as it should, investigate the difference in sorting. It could be that you need to return the results in a different data type before the work is expected. Both the table and the query ordering numerically work because the numeric values are being handled in the same manner, but I'll order by customer in both the table and the query. And then if I select the fourth page on the table and

then change the query to skip the first 30 records, the results aren't the same, and that's because the table is processing the order by when a single quote is in the field differently than the processor. Looking at the full record set, you can see that when ordering by company name, the company B's Beverages is the 101st record. But in the table, it's the 41st. Let's look at how result sets can be limited to a certain top number and a certain top percentage of the total query results.

TOP Functions

The TOP function, as the name implies, returns a specific number or percent of the records. The syntax begins with the keyword TOP and takes a single parameter, which represents the number of records or the percent of records you want returned. The other roles on the SELECT statement syntax we've used in the course are still going to apply. ORDER BY is optional when using the TOP function unless you're using WITH TIES. But without it, the records returned are going to be in no particular order. This query is returning the number of orders using COUNT and the company name. Let's look at COUNT. COUNT is an aggregate function, and I need to give you an introduction here because it's so much a part of the TOP function. When you're working with TOP, you're looking for specific values such as the top-producing sales agents or the most popular products. And to gather that information, you're going to need COUNT. COUNT requires GROUP BY when other fields are also included in the query. And, as I mentioned before, the module Aggregating Data in a Query is going to cover this in detail. So, the focus here is on TOP n and TOP % the query is ordering by this aggregated field named NumberOfOrders returning all of the companies who have ordered from Northwind. Using TOP, this is limited to the top 20 customers. Executing the query returns just the first 20 records. But looking at the full dataset, the 21st customer also has 32 orders, the same number of orders as the 20th customer. To keep from excluding records this way, when the cut off brakes on records that have the same values, you can use WITH TIES. When WITH TIES is added, it will look beyond the number of records requested and return records that have the same value as that last record in the cutoff. Adding WITH TIES and executing again now brings back that additional record that also had 32 orders, and a total of 21 are returned even though TOP is using 20. If the 22nd record also had 32 orders, then it would have been included as well. WITH TIES requires ORDER BY. I use TOP without an ORDER BY when I'm testing a query against a very large dataset. And this way, instead of returning, say, 500,000 records, I can see what the results will be by returning just a few records. This query is returning the

employee name using the function we wrote in the module Shaping Data in a Query, which is returning the employee's names with proper casing and again using COUNT to get the COUNT of orders associated with the employees using the EmployeeID field from Orders and then sorting by NumberOfSales in descending order so that the highest number of sales is first. There are 10 employees, but only 9 associated with sales. And I'm not concerned with employees that haven't made sales, so a simple JOIN is used. Adding TOP and then 5 as the parameter returns the top 5 employees. And ordered by the number of sales, this represents the top 5 employees based on those sales. In a query like this, I recommend always using WITH TIES because if you do have employees that would have been in the top 5 category based on the number of sales, you wouldn't want any of them excluded. When the parameter used in the TOP function is greater than the total number of records, then the full result set is returned, and TOP is ignored. The TOP function can also be used to return a percent of the total number of records. The syntax is the same, but you add in the keyword PERCENT. When PERCENT is added, then the value used as the parameter is converted to a float. For instance, if I want to return the top 30% of customers, I'd add the keyword percent. There are 27 returned. There are 91 customers, and 30% is actually 27.3. But PERCENT will round because you can't return a third of a record. Going back to the top employees and adding in 5 and then PERCENT only returns one employee, and that's because 5 is converted to .05, and .05 of 9 is .45. So in this case, it's only returning one record. With only 9 records, you won't get more than 1 employee until you enter at least 12 as the parameter because 12% of 9 is 1.08. It's important to consider the total number of records overall when using PERCENT.

Using Variables to Limit Results

A local variable is an object that can hold a single data value of a specific data type. In this context, the word local means that it's recognized by the query itself and nothing else. In a query, variables have three properties, their name, the data type, and the value at the time its instantiated, which means when an instance of the variable is created, you can change the variable values to return different results rather than writing an entirely different query. To set up the variable in your query, you start with the keyword DECLARE, and we've discussed earlier that a keyword is a reserved word in a programming language, and it has a special meaning to the database engine. After DECLARE, you name the variable. And variable names begin with the at sign, and the name should be something

meaningful as to what it will represent. And then after the variable name is the data type of the variable. Using variables is very common, and you can imagine a reporting tool where the user can enter the number for the top and then either designate a numeric value or a percent of the records. This query is using a variable of data type BIGINT, setting the value, and then replacing the parameter with the variable. The numeric expression that specifies the number of rows to be returned is implicitly converted, implicit meaning implied, so you don't have to be concerned with decimals. If you add in PERCENT, it's a float. Otherwise, it's converted to a BIGINT. This query is using the variable for the numeric parameter, but I've also added in additional logic. Imagine a reporting tool that allows the user to select not only the number, but also whether they'd like to see that result set returned as the number count or the percent. You wouldn't need to write two queries. Instead, you could use the if/else logic and another variable indicating whether that should be a number or percent. We only need one value because remember that with if else it works like either/or. When the expression used with if doesn't evaluate to true, then the else block will execute. Because we aren't setting a value to the type variable, then type will not equate to 1, which means the IF portion won't evaluate to true, and so the TOP PERCENT query will be used. Setting type to 1 and executing again returns 30 records because the IF expression evaluated to true. These queries generally use variables to determine the next set of records to be returned, and this is how it works. Instead of hardcoded values, this query is using two variables, fetchNum and pageNum. The OFFSET value will be determined by the number to fetch and the page number. When the page number is 1, then OFFSET will be 0. So the page number, if you subtract it by 1 and then multiplied by the number to fetch, in this case returns 0. Setting the page number to 2, subtracting 1 because we want the second set to begin with that 10th record, the value will be 10. If I change the number of records to be returned from 10 to 25, then the fetchNum value will be changed, but the pageNum would stay the same. OFFSET and FETCH are commonly used, not just with SQL, but with other types of applications. And now when you look at a table or other search platforms, you'll understand how they're working.

Review and Resources

Limiting the results using TOP and FETCH NEXT are used in data analytics, reporting, and pagination. We've worked with these functions, reviewing their syntax rules and using variables to return a specific set of numbers. And by now, you have the knowledge to envision what a query would look like that

could be used to populate a web table, how setting the number of records to be returned at a time and the section of rows to be returned are set. And we've also talked about ordering concerns across multiple platforms. This is a list of the corresponding query examples, as well as that web page including the data from orders and customers demonstrating how these functions work. And I've also provided some links to the Microsoft site that can provide you with additional information on these functions. In the next module, we'll look at the aggregation functions that are useful when determining the number of records in tables, how they can be used to check for data integrity, and how they're used to get the sum, average, minimum, and maximum values in a set of records. Please join me in this next module, Aggregating Data in a Query.

Aggregating Data in a Query

Introduction

Hi, I'm Tamara. Welcome back to Combining and Filtering Data with T-SQL. Aggregation is the functionality you use to perform higher levels of mathematical calculations. I'll show you how to use aggregation to answer questions such as how many of a particular item has been sold, what's the total volume of sales for last year, what's the average purchase amount per customer, and who is our best and worst customer in terms of sales? Then I'll walk you through filtering the query results using the HAVING clause and demonstrate organizing the aggregations using GROUP BY.

Using COUNT in T-SQL

COUNT returns the number of records based on the parameter used in the COUNT function. It requires a single parameter and a simple SELECT. Using the wildcard asterisk will return a count of the records in a table. Because it's a record-based function and doesn't perform any mathematical calculations other than counting, it returns a count of every record meeting the criteria and will also include null. If other fields are added to the SELECT statement in the query that are not being counted in this COUNT function, then the GROUP BY clause is required. Before, we filtered using the WHERE clause. But when you're using aggregation in your queries, instead of WHERE, you'll be using a

HAVING clause. Let's write some queries. This is one of the INNER JOIN queries we did earlier, and it's returning the items in an order with additional details. We used a simple SELECT statement in the result pane window to validate that all of the records were returned in the query, and then we added in DISTINCT to the Order Details query to get the correct count of the OrderIDs. And I mentioned earlier that there was a better way to get this validated. This is one of those ways. COUNT can be added to these queries and then only the number of records matching the query parameters will be returned. I'll use the asterisk as the parameter in this order query. And instead of returning all of the records, the result is just the count of every record in the table, which in this case is going to be correct because no one order will have two records. But adding COUNT with OrderID as the parameter in Order Details returns a count of every OrderID, and I just want a count of the distinct orders. I can use DISTINCT as we did before along with the COUNT function, but DISTINCT has to be added in along with the OrderID parameter, and this is why. I can add DISTINCT as I did before and then have a count using OrderID as the parameter, but I'm still getting back the 2155, which is actually the number of records in Order Details. The reason is DISTINCT is working off of DISTINCT value. And because COUNT is only bringing back one value, then DISTINCT won't recognize that we want the DISTINCT OrderID. Including it inside the COUNT function specifies that we want a count of the unique OrderIDs and not a distinct count of the records returned. Let's write a query that returns the product name and the number of times that product has been ordered. This diagram illustrates the relationship between Products and Order Details. ProductID is the foreign key in Order Details that can be used to return the count of how many times that product has been ordered. This isn't the number of products sold. There is another field in Order Details that contains the quantity purchased, and that will require using SUM. When we complete this query, we'll build upon it using SUM to return the number of times it was included in the order and the total number of products sold. We know these are the tables we need, and from our discussion on JOINs, we learned that one of the products had never been ordered. So to return all products whether they've been purchased or not, we're going to need a LEFT JOIN from Products to Order Details. The fields to be brought into the query are ProductName and then ProductID from Order Details. To get started, I like to bring in everything and then get the join put together on the tables I need. This is bringing everything in from Products and Order Details using a LEFT JOIN, then I'll do another quick validation using COUNT. Products can use the wildcard because each record is a unique product, and then using a DISTINCT COUNT of the ProductID from Order

Details will return a count of each unique product sold. This query returns one less than products and represents that product that was never included in an order. Going back to the query and ordering by Order Details, ProductID will return that one product with null, and we want to make sure this product will be returned, which is why that LEFT JOIN is so important. With the query framework started, now we can remove the wildcard and add in the ProductName from Products and then a COUNT of ProductIDs from Order Details. Because the ProductName is not part of the aggregation, this query is going to require a GROUP BY clause, grouping on ProductName. GROUP BY is required if there is a field in the query that is not included in aggregation. In this case, we're using COUNT. The GROUP BY statement groups rows that have the same values into summary rows. And here, we're asking for the count of products that have been ordered grouped by each product, and I'll change the ORDER BY to a COUNT of ProductID from Order Details. Executing the query returns all 78 products. Because we use COUNT on products and 78 were returned, we know that all of the products are being accounted for. I don't want to use DISTINCT in the query because it would return the DISTINCT COUNT of ProductIDs. Because each product only has one ProductID, this would return a count of 1 for every product that had been ordered and a 0 for the chocolate covered pickles. So where we use DISTINCT to validate the number of Products in Order Details, this time we want a COUNT of the ProductID and not a DISTINCT COUNT of ProductID. Executing the query, we now have the product name and the number of times that product has been included in an order. If I want to return only a list of products having a count of a number sold greater than 10, I'll need to use a HAVING clause. Before, when we filtered, we used the WHERE clause. But when aggregation is in a query, you'll need to use the HAVING clause instead. Executing the query returned 68 records, and those are representing 68 products that have been sold more than 10 times. This query is an example of one that could be used by Northwind company buyers to determine what products they should reorder or discontinue based on the number of times the products have been sold. Their cutoff point may not be 10, but if we add a variable representing the 10 instead of hardcoding, then this could actually be a report that would be called from a user interface where the user would just enter the number, and then the query would use that variable to determine the filter on the results. Let's add a variable of data type integer named timesOrdered and set the initial value to 10. Then, replace the hardcoded value of 10 with this variable. The result still returns any of the products that have been ordered more than 10 times, but it's using this variable as the parameter. Now we just need to change the value of the parameter, and the

query will return the results based on the parameter. This is a great example of a report you may be requested to provide. The user would enter the number of products they want returned, and then the query would use that parameter instead of a hardcoded value. Next, we'll use SUM to return the quantity ordered, as well as the number of times that product has been ordered.

Using SUM in T-SQL

SUM has the same rules that the COUNT function does regarding GROUP BY and HAVING. GROUP BY is required if the fields in the SELECT statement are not included as part of the aggregate function, and HAVING replaces WHERE. It takes a single numeric parameter. Where COUNT returns the count of the field used in the parameter, SUM adds the value of the fields. If there are no values in a field though, SUM function returns null, which is the absence of value, and I'll show you how to handle this situation in queries to return zero instead of null. This is the report from the COUNT query, returning the product name and the number of times it was ordered. Now let's add the total number of the product that's been ordered. You can move query windows either vertically or horizontally, and I'll move this report to a new horizontal window so we can use it for a reference as we change this query. The COUNT field has an alias of TimesOrdered so I'll add that in. And the field that contains the quantity of each product ordered is named Quantity, and it's in Order Details. This will be the parameter that we'll use for the SUM function. SUM will return an error if the parameter is anything other than a numeric value, and quantity is a data type of SMALLINT so this will work. And then I'll add the alias name of TotalOrdered. Executing the query does bring back the records, but that one product that was never ordered is returning null instead of 0. So I can add in a CASE statement to check for null. And then if it is null, return 0. Otherwise, return the sum. Executing again and we have 0 instead of null. Let's add another parameter to the HAVING clause to return records that not only have had a certain number of orders, but also a quantity order greater than 50. And I'll change ORDER BY to DESCENDING to return the best-selling products at the top of the query. The query is working, and now let's add another variable and replace the hardcoded value of 50. I'll name the variable TotalOrdered and set the value to 50. And just as we did with COUNT, the variable value can be changed and the query will work without this hardcoded value of 50. Let's look at the AVERAGE aggregate function.

Using AVG in T-SQL

The syntax rules for AVERAGE are the same as for the other aggregate functions we've looked at to this point. It requires a numeric value and can use the DISTINCT keyword just as SUM and COUNT. Let's look at some examples and create a report using AVERAGE to return the product, average number sold, and grouped by quarter and year. I've created a table in the Northwind database named Numbers so that we can see how AVERAGE and AVERAGE WITH DISTINCT work using a small dataset. There are 10 records, and some of these are duplicate values. The SUM is 48, and the COUNT is 10. The average then, just looking at these, is 4.8. But the AVERAGE function without explicitly casting to a decimal data type will round down to the nearest whole number, and you'll have to include CAST to return this in a decimal format to get the exact number. DISTINCT takes the distinct sum of the field and then divides by the distinct count. This query is demonstrating what's happening in the AVERAGE function. Using DISTINCT, it returns 42, which is the sum of the distinct values, and then divides by 7, which is the number of records with that distinct value to return 6. This is what's happening when you use DISTINCT inside the AVERAGE function. This is the report we'll be creating. It's using AVERAGE to return the product, average sold, and then grouped by quarter and year. We're going to need the date of the order to get quarter in year. This is in the Orders table, then ProductName and ProductID are in Products, and the quantity which will be used to get the average is in Order Details. The join will need to be a LEFT JOIN from Products to ensure that all the products are returned whether they've been ordered or not, and then details and orders will also need to use that LEFT JOIN so that any of these products aren't excluded. I'll start by using the asterisk and then bring in the tables with the proper joins. This not only checks to make sure everything is accounted for, but it also starts IntelliSense, which is helpful when we start adding in the individual fields. We can add in a WHERE clause at this point because there are no aggregate functions, and then returning anything where the OrderID in Order Details is null validates that that product never ordered is still being returned. With the join validated, we're ready to add in the individual fields. The first two are ProductID and ProductName. I like to execute these queries as I write to make sure nothing has been removed from the results. There are 2156 records, Order Details has 2155, and this LEFT JOIN is bringing back that one product that has never been sold. So we validated that the other fields being added hasn't eliminated that one record. Next is the average, which comes from the quantity sold in Order Details. Now that there's an aggregate function in the query, the GROUP BY clause is required

because we also have those other two fields that aren't included in an aggregate function. Remember that every field you bring in not included in an aggregate function has to be included in the GROUP BY. The definition of this query then is provide the average quantity grouped by ProductID and ProductName. Executing again this time to validate returns only 78 records instead of 2156, and that's correct because there are 78 products, and the results are now rolled up using AVERAGE and grouping by the ProductID and ProductName. Quarter is the value of the OrderDate field Quarter. We're going to need the DATEPART function. Recall from formatting dates and numbers that to get the quarter, the first parameter in DATEPART is q, and the second parameter is that datetime value, which, in this case, is the OrderDate field from Orders. Now, DATEPART isn't an aggregate function, so this also has to be included in the GROUP BY clause. This time, however, the grouping will need to include the quarter using DATEPART of the date ordered because we aren't using the order date. We're using the quarter of the order date. Then last, we need to add in that order date also using DATEPART. This time, the first parameter is year, and the second parameter is OrderDate from Orders. The OrderDate DATEPART year needs to be added to the GROUP BY clause. Execute the query to make sure the result set number is the same and that we aren't receiving any errors. And so the last thing to do is add in the alias names. Quarter and Year are also recognized keywords in T-SQL. It won't return an error if you use these as alias names, but a best practice is to enclose them in brackets to distinguish them as field names instead of keywords. Reviewing the data, notice that some of these quarter and year fields are null. We know that the data validation is returning the correct number of records, and there are also values here that are correct. This means there's a data integrity issue with the underlying data source. Because it's the OrderDate field from Orders that's being used to return the quarter and year, that's a good place to start the analysis. OrderDate in the Orders table allows null, and a simple SELECT query is validating that some of these order dates are not in the table. This is where those null values are coming from. At this point, it's been determined that there is data integrity concerns with the Orders table, and so you'll have to make a decision. The query is correct, so you could just leave nulls. But null doesn't give a lot of detail into what's happening. You can use a CASE WHEN statement to return something other than null, which is what I'd suggest. But what you decide to return is up to you and the business requirements. Next, we'll write a report using this AVERAGE function along with a MAX and a MIN to bring back a company name, the last order

they placed with Northwind, and the first order they placed with Northwind, including an average sales amount formatted as currency.

Using MIN and MAX in T-SQL

MIN and MAX are alike. Regarding all of the syntax rules for GROUP BY and HAVING, MIN returns the lowest value, and MAX returns the highest value. They take a single parameter. They accept DISTINCT and return a single value based on the grouping and the HAVING clause. The data type returned will be the same as the data type used in the parameter. And an important detail to remember for both MAX and MIN is that they ignore null values when evaluating. However, they will return null if the field being evaluated is null, and we'll see examples of this as we write the queries. This is the report we'll be creating. It contains the date of the most recent order, the date of the first order, and the average order amount for each company. We'll be using the Customers, Orders, and Order Details tables. CompanyName comes from Customers. The OrderDate field in Orders will give us the first and last order values. And then Order Details contains the three fields we'll be using to calculate the average order amount. This will take the three fields UnitPrice, Quantity, and Discount. Products also contains a unit price, but that unit price is the current price of the product. Because the product prices change, we need to use UnitPrice from Order Details, which will be the price of the product at the time it was ordered. Discount is also in this table and not Products because discounts can change, such as they're having a sale on a product, or perhaps a customer who orders in bulk will get an ongoing discount that other customers may not receive. So, discounts are applied to the individual item ordered and not on a product level. Just as with the other queries that we've written to this point, we'll start using the asterisk to bring in all the fields and then get the joins right and then get the numbers we're going to need to know to validate the query results. We need a LEFT JOIN here, just as we did when we were getting the average for the quantity sold on products, and that's because we can have customers that have never placed orders, so we'll need that LEFT JOIN there, and then we can also have orders that haven't been completed and may exist without any items in the Order Details records. The first thing we'll check is to see if there are any OrderIDs in Orders that are null using a WHERE clause. There are no aggregates at this time, so WHERE will work. And this is returning two records, which means there are two customers that have never placed orders. Next, we'll get the count from customers. And because this report is about customers, we need to make sure that there

are records for every customer in the table. This returns 91, and this is our target. This is the number of records we need returned, and we'll validate on this number. Now that we know the customer number, we can find out the number of customers who have placed orders using COUNT and DISTINCT CustomerID. This time 89 are returned, and that represents 2 customers that have never placed orders. If we do a COUNT on Orders, there are 832 orders being returned. But a DISTINCT COUNT of OrderID from Order Details returns 830, and that's going to represent 2 orders that have not been finalized and haven't had any items added to them. We now know the numbers, and we know that the join is correct, so we can start adding in the field. The first field isn't aggregated; it's just the customer name. So we'll bring that in and then remove the asterisk and then get the most recent order date for each customer. This uses MAX and then that OrderDate field from Orders. Now that there's an aggregate function, we're going to need to add the GROUP BY, which need to include the field or fields that aren't included in aggregate functions, and that right now is going to be the customer name. When we execute this query, we should have those 91 records in the result set because we've now grouped by each customer. Remember how we talked about MIN and MAX will ignore null values when evaluating for MIN and MAX. But when there's a null value and the query is grouping, they can return a null value, and these null values are representing those customers that have never placed orders. We have the correct number in the result set, and we validated that those two customers who have never placed an order are still being returned in the result set. Now we can use MIN and the OrderDate fields from Orders to get the first order date, execute again just to make sure we have the 91 records we expect in the result set, and we're good to move on to the next field, which will bring back the average order amount for each customer, and this is where we're going to use those three fields in Order Details. We need the quantity, the unit price, and a discount if there was any applied. We'll use the AVERAGE function and then apply a mathematical calculation to these three fields. We went over these calculations in detail when we looked at Dates and Numerical Values. So if you need a refresher, go back and review that part of the course. Execute again just to make sure 91 records are being returned. And now that it's validated, let's add some formatting to that AverageOrderAmount field. Formatting was also covered in Dates and Numerical Values, so we won't go over it in detail here. And now, all that's left for us to do is to add in the alias names for these three aggregated fields. Let's go back to this average order calculation just a moment. Returning an order amount is something that you're going to find could be used in a number of queries because you've got the products where

you would need to know, the total in sales, customers as we've seen here, and also you have employees that are attached to sales. And when you have a calculation like this that can be done in numerous areas, this is a good candidate for a user-defined function. One place where that calculation is called from also means one place to maintain. If something needs to be added or removed from this calculation, you could do that with one query versus going through all the queries and all the reports you've written that would use this calculation. I've created a user-defined function named OrderAmountPurchased, and it's using just these three values to do this calculation and a single parameter of the OrderID. We can also add our CASE WHEN statement into the user-defined function, which will greatly simplify our individual queries. Now we can remove this line, call that user-defined function, and then just add the formatting. So this is something you want to keep in mind as you're writing reports and queries. And when you see things like this, begin to form a pattern. Where you use a calculation over and over for multiple reasons, a user-defined function is a great way to handle those.

Review and Resources

We've covered a lot of material here, and working with aggregation is something you're going to want to become completely familiar with because these functions are heavily used in every aspect of data management from reporting to analytics. We've gone over the query syntax rules when aggregating using GROUP BY and HAVING, which replaces WHERE when aggregation is involved and how the COUNT function can be used to validate results. Here's a list of the queries and reports that were used in this module. I recommend practicing with these queries because that will provide you with a better understanding of the rules and how they work. Please join me in the next module where we'll look at how to use T-SQL set operators, which are used to combine the results sets for multiple queries.

Using T-SQL Set Operators

Introduction

Hi, this is Tamara. Welcome back to Combining and Filtering Data with T-SQL. In this module, we'll focus on set operators. In this module, I'll compare queries using joins to queries using set operators, and I'll explain their differences, as well as their similarities. I'll show you how to write queries using four set operators, and then we'll modify some of the queries we've written to use these set operators instead of joins. Along the way, I'll talk about some of the use cases where set operators would be a better choice over joins. So let's get started writing queries and using these set operators.

Difference between Joins and Set Operators

Where joins merge data from multiple tables using a single query joining on fields with common values, set operators return the results from multiple independent queries, which are fully functional queries, into a single result set based on the type of set operator used. Where joins merge data from multiple tables using a single query and joining on fields with common values, set operators return the results from multiple independent queries. Each query is fully functional, and the results are simply returned in a single result set based on the type of set operator you use.

Union All

The UNION ALL set operator returns all result sets from multiple queries, returning the results from each query into a single results set. Each query is a working query that could be executed on its own, and UNION ALL doesn't eliminate duplicates. This first query is using TOP and 3 to return three company names from Customers. And the second, query is using TOP 3 to return employee names from Employees. And its implementing UNION ALL, which means that all of the results from both of these result sets will be returned as a single result set. There are a couple of rules that apply to set operators. First, the result sets must contain the same number of fields. Second, the fields must have the same data type. You can use TRY_CAST or TRY_CONVERT as I've done here, which returns null for any value that couldn't be converted. Third, ORDER BY can only be added below that last query, and it's going to order all of the result sets from all of the queries. You can use ascending and descending and any of the other GROUP BY functionalities, but the ORDER BY only recognizes the field name from that first query. That first query as your primary query, and the field names that are returned with the final result set will have that field name from the first query. And you can use the field

number instead of referencing the name, just as you can with any of the other queries you write using joins.

Union

UNION returns the combined result set into a single result set, but eliminates any duplicates. This query is combining the City field from Customers with the City field from Employees. There are some duplicates in these tables, but only the unique values are returned, just as you would expect using DISTINCT. with joins, you have to specify which table contains the field you want return, and combining them into a single field would require additional work, perhaps in a spreadsheet or some other tool to get these into a single list. With UNION though, you can return them in a single field, and there's also a way that you can distinguish which result set returned the value. If you want to return a list of distinct values, but distinguish which table the values originated from, you can use UNION to bring back the distinct values, but also add a hardcoded value to differentiate which result set returned the value. A good use case for this scenario would be if you've been asked to return a list of cities where your employees are located, as well as the customers to determine which employees are in the same locations as your customers in the event that you wanted to have them do an in-person visit.

Intersect

The INTERSECT SET operator returns only matching results from the queries using that first query as the primary query. It's similar to the INNER JOIN. However, where INNER JOINs will return duplicates unless you use DISTINCT, INTERSECT removes duplicates by default and uses all the fields in the queries where all fields match. Let's start out by revisiting one of the previous queries we wrote in the module Querying Data from Multiple Data Sources. Using the INNER JOIN, this query returns a result set with all of the OrderIDs in Orders where there are matching OrderIDs in Order Details. And we had to add in the DISTINCT keyword to remove any duplicates. When we were filtering data using WHERE, we also added in a nested SELECT query to return only those orders that contained a matching OrderID in OrderDetails, but we can get the same result set using INTERSECT. And because INTERSECT compares all of the columns specified in the SELECT statement, then only matching records where all of the fields match will be returned. When we wrote our queries using

UNION, we used a hardcoded value to determine which table the value had originated from, and then it returned a combined result set of the employee cities and the customer cities along with that hardcoded value. But with INTERSECT, we can't use that hardcoded value because INTERSECT is looking for all the fields to be alike. So here, replacing UNION with INTERSECT, no records would be returned in the result set because of that second hardcoded value that we put in to determine the Employee table or the Customer table. Removing that hardcoded value returns three records, and these three records represent the three cities from Customers that also exists in Employees. And remember with UNION, we wanted to find a way to determine which employees were located in the same city as our customers. INTERSECT won't return that type of information though, and so UNION might be a better choice if you needed to know which table the record was referencing.

Except

The EXCEPT set operator does just the opposite. It returns a result set, again based on that first primary query, but the result set is going to be a list of all of the records that don't exist in that second query. If we use EXCEPT, the result set is a list of cities in the Customers table that do not exist in the Employees table, just the opposite of intercept because that first query is the primary query. This means that if the SELECT statements were switched, as I have here in this example, then the result set will return the employee cities that are not in Customers. Another way to write this query that we've done before would be to use the nested SELECT. But for readability and the ability to return a single list without using DISTINCT, using these set operators is the best practice because it also returns all of the values in a single field; whereas in the standard SELECT query, you can only return a specified field from a specified table. Throughout the course, we've used the Northwind orders as an example, and through this process we know that there are two orders that have never included any items. Up to this point, we've used LEFT JOINs and WHERE clauses filtering for null. We've used COUNT and DISTINCT COUNT to quickly identify the total number of orders, as well as orders without products attached using the primary and foreign key fields. And we've written queries filtering using nested SELECT. All of these approaches are valid and will return accurate results. But with EXCEPT, we can write queries with better readability that can also return this same result. This query is using a SELECT statement from Orders and another from Order Details. With EXCEPT, the results will return a list of these orders that don't have any associated items in Order Details. Because this first query is

using Orders, and it's the primary query, that OrderID field in Orders is the field that will be used to evaluate using EXCEPT. Switching these doesn't return any orders, and that makes sense because we know that OrderID in Order Details is a foreign key, and that constraint prevents an OrderID being added in Order Details that doesn't exist in Orders. But where OrderID is the primary key in Orders, a new order can be started without any other information being entered into any table where OrderID is a foreign key. Here's another example. We looked at customers earlier on and identified that we had customers that had never placed in order. We use JOINs and COUNT and nested SELECTs to validate and identify these customers. And now we can use EXCEPT with that first query being the CustomerID from Customers, the primary key. And then the second query returning the foreign key, CustomerID from Orders, or any other table that had a relationship to Customers using a foreign key. Again, CustomerID in Customers is required; CustomerID in Orders is only required when an order is placed and attached to that customer. And because of this, the results will only be valid if that first query contains the primary key to be used as the comparison value. And switching these SELECT statements with the foreign key first won't return valid results. So from these examples, you can see how INTERCEPT and EXCEPT are great tools for validating, as well as reporting.

Review and Resources

To summarize then, UNION ALL returns all results, including duplicates. UNION removes duplicates, but you can add in a hardcoded field if you need to know where the result originated from. There are rules for using set operators. The queries must have the same number of fields. The fields being combined must have the same data types. You can use casting and converting, which will return null where the data type conversion isn't allowed. And I'd suggest that you use TRY_CAST and TRY_CONVERT to minimize your errors. Explicit conversions still aren't allowed, and you can refer back to casting and converting for a detailed look at the explicitly not allowed conversion data types. ORDER BY is placed below the last query, and the first query is the primary query, which means that the name of the field referenced in the ORDER BY clause must come from that query, Or you can use the field coordinate. Just as with UNION and UNION ALL, the rules as to the same number of fields and the same data types apply. The INTERSECT set operator returns common values from multiple queries and returns the DISTINCT values as WHERE before with the other queries we've written. We've had to explicitly add the DISTINCT keyword. The first query is the primary query, which then

compares the sequential queries, returning matching values. The EXCEPT set operator does the opposite. It returns values that are not in the comparing queries with that first query being the primary query. This is a list of the queries we wrote in this module that you can use as a reference. Next, I'll show you how to further simplify queries using intermediary tables, common table expressions, and then implementing cursors to iterate through the results.

Simplifying with Intermediate Tables and Common Table Expressions

Introduction

Hi, I'm Tamara. Welcome back to Combining and Filtering Data with T-SQL. My goal in this module is to continue taking you beyond the basics of T-SQL by introducing you to common table expressions, or CTEs, which are temporary named result sets, and local temporary tables, which are only visible in the current session, showing you how to apply these to work with more complicated queries. Using some of the queries we've written so far in the course, we'll implement common table expressions and then temporary tables. We'll discuss the difference between temporary tables and common table expressions, go over the scope of these, and then end with an introduction to the cursor, which provides you with the ability to process the results one row at a time. So, let's get started.

Using Common Table Expressions in T-SQL

Common table expressions referred to as CTEs are temporary result sets which are released from memory once the query executes. As we write our own common table expressions, the syntax will become clearer, where at first it may seem a bit confusing. CTEs are constructed using the keyword WITH and then followed with the name you assign it. After the name, you add the fields within parentheses. And then using the AS keyword, a query is added to populate the CTE. The query to populate the CTE needs to have the same number of fields added in the constructor, and the data

types in this CTE are determined by the fields in the query. After the CTE is constructed and populated using a query, then another query consumes the result set. After the query is executed, then the CTE is released from memory. We'll use three common table expressions to create this report. I've created three queries that we're going to use to populate the CTEs. We've covered everything in these queries, so I won't be going into the detail. The full script for this report is included in the exercise files for you to reference this first query retrieves the employee and manager information. Next, we get the sales count and region count. And then this last query is going to use that user-defined function I showed you earlier when working with aggregation, and it brings back the total from each order with the calculations added for discounts, units, and price of each line item in the order. And then when these three CTEs are put together, we'll use the result sets from each one to bring back a final calculation that will give us the sum of the orders and the average order amount grouped by employee. To start out though, we'll construct a common table expression for each of these three queries, validate the results, and then combine them to get our final report. So starting with Employees, I'll open a new window, paste the query in, and we can now add in the syntax that's going to create the CTE. It starts with WITH and then the name you're assigning to the CTE. And then in parentheses, you define the fields that will be included. After that, you add in the keyword AS. And then inside parentheses, we place this Employees query, and this query will populate this new CTE. The number of fields you bring back from this query need to match the number of fields you define for the CTE. The field data types in the CTE will be determined by the field data types in this query. Then, the last requirement is to consume the results from the CTE in another query. If you don't do this, you're going to have an error returned, so it's part of the requirement. I'll just use a wildcard and a SELECT statement, and then the query this time is going to use the CTE name assigned, and this result has given us the first three fields we're going to need for that report. So now, I'll repeat the process using this second query, which will return the region and the sales count. Again, I'll open a new window, paste in the query, and then add in the syntax for the CTE. We're also going to need the EmployeeID for this CTE because when we get to the last query where we consume all of the results, were going to combine them together using that EmployeeID in the JOIN just as we've done when we've joined other tables. The only difference is this time we won't be joining tables. This last query will be used to calculate the sales volume and the average sales amount. So we'll do the same thing by opening another window, pasting in this query, and adding in the CTE constructor. All three CTEs

have been constructed, and now we can chain them together, but let's execute one more time to make sure we don't have any errors. This time though, we're getting an error, and that's because unlike anything we've done to this point, CTEs require that you place a semicolon at the end of anything that isn't part of the CTE. We have the USE Northwind statement at the top, and that's causing the error. So we need to add a semicolon this time at the end of USE Northwind. Executing the result returns the EmployeeID and that calculated order total. The final step is to combine these three CTEs into a single query that will bring back all of the fields needed for the report. I'll begin by copying all of these into a new window, adding the semicolon at the end of each one, and executing to make sure we don't have any errors. Now that they're working, we need to chain them together to return the results using a single SELECT query. To chain CTEs together, you remove the individual SELECT statement from the CTE and replace it with a comma. This chains those CTEs and will bring back the results with a single query. When you write that last query, you don't have to bring in all of the fields from these three CTEs. But to get started, just as I've done with the other queries we've written, I like to use an asterisk and get my joins correct before I start adding in the individual fields. Instead of table names, again, we'll use the CTE names. And we'll use a LEFT JOIN because the focus of our report is employees, and it's possible that we could have employees that have never had any sales, and we want to make sure those records are returned as well. Execute to check our query. It's unformatted, but we aren't having any errors returned, and we also have a record for each employee. And so the last step is to add in the individual fields and get that final aggregation, bringing back the average sales amount and also using SUM to get the total sales volume for each employee. Execute again, and we have our result set with all of the fields we'll need for this final report. We have a lot of nulls in here, and we've covered in detail how to handle nulls using CASE WHEN statements to bring back something other than just null. This is what the final query looks like that will produce this report, and this is the query that you have a copy of in the exercise files. Next, we're going to produce this same report, but we're going to use temporary tables instead of CTEs.

Using Temporary Tables in T-SQL

Temporary tables are very similar to the common table expressions we just looked at, and they're useful for processing data when you're applying complicated calculations or aggregation. The syntax is the same as any other query, and there's no limitation on the functions or methods you can employ.

They do require that you use alias names on any fields using aggregation or calculations that would otherwise return a result set without column names, and we'll look at a couple of examples when we start writing our own temp tables. To create and populate temporary tables, you use the pound sign and then assign a variable name. These temporary tables are only visible to the session in which they're created, and they're automatically dropped when the session logs off or you release them from memory using a DROP TABLE command. They can offer a performance benefit because their results are written to local rather than remote storage. Let's begin by writing a simple query that will return the count of the customers. Now, I'll make this into a temporary table by adding the INTO tmp statement which goes directly above the FROM statement. When I execute, an error is returned, and that's because temp tables require alias field names on all columns that use calculations and aggregation and normally wouldn't return the result set with a column name. Adding in the name CustomerCount and executing again doesn't return an error. But instead of the results, a message is displayed indicating that a query ran and records were affected. This means that the query populated the temporary table, but to view the results, we have to write another query against that temp table. Once a temporary table is created though, it can't be created again until it's dropped. Before I can create and populate it again, I'll need to use that DROP TABLE command first and release it from memory. I'll execute just this DROP TABLE command and then run the entire query again. This is why it's a good idea to add that DROP TABLE command at the end of your query because it will prevent you from having to do this DROP TABLE command manually every time. Let's take that final query we wrote using common table expressions and alter it to use temporary tables instead. I'm going to copy it to a new window, and then I'll begin by removing the CTE syntax and inserting that INTO statement and assigning a temp table name on all three of these queries. For Employees, I'll name the temp table ems. Regions and sales count will be orders, and then last will be sales. And I'll also add the DROP TABLE command to release it from memory at the bottom. I'll write SELECT queries for each of these three tables we created, and the results are returned just as they were when we used common table expressions. Let's alter that final query using CTEs and replace all the CTE references with these temp tables. I'll just replace the CTE names with the temp table names. And executing the query again returns the same results, but this time using temporary tables.

Using Cursors in T-SQL

You're going to run into situations where processing the records will need to be done one row at a time, for instance when you're applying logic or very complicated aggregate functions and calculations. This is where the cursor may be a good choice. Cursors though tend to be heavy in terms of processing because they process it one record at a time, which impacts optimization. So use them sparingly and consider temp tables or user-defined functions first. It's common to see these though, and it's important that you know how they work. This is a very simplistic example of a cursor, and I've done this on purpose because there's a lot going on with cursors, and I want the focal point to be on this rather than the query that's wrapped inside. Let's go over this cursor one line at a time. First, the cursor requires variables that will point to the fields of the individual records. In this example, I'm using the EmployeeID, FirstName, and LastName fields of the Employees table, and these are the variables that will be used to represent the values each time the individual record is processed. Next, the cursor is declared, and then it's populated with data using a SELECT query. After the cursor is declared and populated, it's opened using the OPEN command. At this point, we have the variables we'll need. The cursor has been declared and populated with the data opened and ready to use. The FETCH NEXT command is used to retrieve a record from the cursor. When the FETCH NEXT command is executed, a FETCH_STATUS is established. FETCH_STATUS is a system command that evaluates the status of the fetch action. A FETCH_STATUS of 0 means a record was retrieved successfully. The order of the fields that the cursor was populated with is the order the variables will be set to. For instance, EmployeeID is the first field in the cursor, and the variable empId is the first variable in the statement, which means empId is set to the records value of EmployeeID. The values of these variables then will be set to the values in the fields of that record. And because of this, the data types will need to match. WHILE is a method used in many different languages, and it's not exclusive to cursors or T-SQL. It's used to loop through or iterate through a section of code, and it uses a boolean value, true or false, to determine when to exit the loop. As long as the expression evaluates to true, the WHILE statement won't exit. In this case, FETCH_STATUS = 0 is the expression being used. Every time the processor reaches the end of the block, another FETCH NEXT command will be executed to return the next record in the cursor. Once the FETCH_STATUS equals anything other than zero, the WHILE loop will be exited. It's a really good idea to save your work often before running any type of a loop statement because you can create an infinite loop where the program will just run because there isn't anything inside the WHILE statement that will evaluate this expression to

false. And this is why the FETCH NEXT statement is also located inside the loop. It's used to determine if another record has been successfully retrieved. If it wasn't there, then the WHILE loop would just continue to process the very first record. So if it's accidentally left out, then the FETCH_STATUS will never return anything but 0. Once the end of the record set has been reached and the FETCH_STATUS is not 0, the CLOSE command closes the cursor, and then DEALLOCATE releases the cursor for memory. The PRINT command inside the loop outputs the result to the message window, printing the values of the variables as each record is evaluated. Let's add a temporary table and populate it with the EmployeeID, FirstName, and LastName from the Employees table and then hardcode a 0 as the fourth field, naming its SalesCount. UPDATE is beyond the scope of this course, but I want to give you an idea of a real-world application of where a cursor could be used. I've declared a new variable naming its SalesCount, and this will hold the count of each of the record sales using the variable EmployeeID in the WHERE clause. And then the temporary table SalesCount is updated to this value. Below the cursor, I'll add a query to return all of the records from this temp table. And then executing again returned both the printed results in the message window, as well as the result set from the temp table, which reflects the updates made for the sales count of each of the employees. In summary then, cursors are used to process a result set one record at a time. And because of this, using temp tables or user-defined functions could be a better alternative due to query optimization. Cursors are populated using a SELECT statement, and variables are declared to represent the values that will be processed. The iteration or looping through each record is done inside a WHILE loop with FETCH NEXT and the system command FETCH_STATUS to determine if the end of the record set has been reached. After all of the records are processed, the cursor is closed and then deallocated, which releases it from memory.

Review and Resources

That's it for Simplifying with Intermediary Tables and Common Table Expressions. We've learned how to use common table expressions, discussed the syntax and rules and the scope and how they could be better suited over JOINs and nested SELECTs. We've worked with temporary tables and cursors to process records one at a time. We've talked about how they can be heavy on resources and that stored procedures, user-defined functions, or views could be a better solution, especially if you have a task you can be using in multiple situations. This is the list of example files located in the exercise file

download relevant to this module, and I've added some links to other documentation you could find useful as well.

Summary

Summary

This also brings us to the end of the course. We've covered a lot of material, and I encourage you to play around with the queries, use them against your own databases, and remember the best practices validating your results and data integrity checks will set you apart from the crowd. I look forward to hearing from you, so drop me a note in the discussion channel.