Course Overview

Course Overview

Hello future SQL master. You've come to the right place to begin your T-SQL adventure. My name is Ami, and I've been living, breathing, working, speaking, and teaching SQL data in databases for longer than I would like to admit. I'm honored to be your guide for this ride. SQL is one of very few technologies that manage to survive for five decades, and even today, is still one of the most widely-used and most loved programming languages. And there are very good reasons for that. SQL is math, SQL is science, and SQL is art. Join me on this magical exploration of the essence of SQL. It doesn't matter if you're taking your first steps on the SQL path, or if you're an experienced developer, you'll find this course extremely valuable. My goal for you is to understand how SQL works, rather than show you how to use the syntax. Syntax is much better spelled out than the documentation. I'll show you the real magic of SQL, that which you won't find online. I'll help you make the switch to thinking in sets and warn you of the logical challenges and common pitfalls you'll encounter. This is what this course is about, getting you on the right track to become a SQL master with a solid foundation and a deep understanding. We'll begin with an overview of the SQL jargon and the tools we'll need. Then we'll quickly dive into the intricacies of SQL. You will learn about the underlying concepts of SQL and their relational model counterparts. You will realize how data is processed by every clause of the select query. You'll understand how to prepare source data, how to filter it effectively, and how to handle the complications of missing data. You'll learn to appreciate the subtleties involved in combining multiple rows and about the capabilities of SQL in supporting presentation, ordering, and paging. I'll also point out when it is best to avoid these altogether. This course is like nothing you've ever taken before. Are you ready to take the red pill and jump into the SQL rabbit hole? What are you waiting for? Let's get started.

What Is T-SQL?

Version Check

Module Introduction

Before we get to T-SQL querying, we need to spend a few minutes in preparation to make sure that our ride will be smooth and effortless. We'll cover some of SQL Server's terminology to get a common baseline. We'll briefly review some architectural aspects of SQL Server that will save you many hours of frustration. Then I'll show you the tools you'll need to use to follow along the demos and practice by yourself afterwards.

What Is T-SQL?

So what is this thing called T-SQL anyway? Let's start by defining what SQL is. SQL is the acronym for structured query language. Some pronounce it as S-Q-L, but I prefer SQL, it is just one syllable shorter. It was developed by Chamberlain and Boyce of IBM in the late 1970s, and later, standardized by the American National Standard Institute, ANSI, and the international standard organization, ISO. The first SQL standard was published in 1986 and the most recent one in 2019. Every vendor uses their own dialect of SQL and each one is made of subsets and extensions to the ANSI SQL standard. T-SQL is the short name for transact SQL and transact SQL is the dialect of SQL used by Microsoft SQL Server and Sybase. This course focuses on Microsoft SQL Server, although the principles we'll learn are valid for all other relational database management systems. So what is Microsoft SQL Server? Microsoft SQL Server is a relational database management system, or RDMS for short. It is a commercial software product that Microsoft originally licensed from Sybase, but was significantly improved over the years. It is in the top three most popular database management systems in the world today alongside Oracle and MySQL, and that includes all types of databases. So what is a relational database management system? These days there are many types of database management systems, or DBMSs. Some use the relational model and other use other modeling paradigms, such as hierarchal, object-oriented, document stores, key value stores, and graph databases, and our DBMS is a relational database management system and it uses SQL, which is based on the relational model. So what is the relational model, you ask. The relational model is a data modeling paradigm based on set theory and first order predicate logic. The relational model was developed by Dr. Edgar F. Codd in the late 1960s. With it, Dr. Codd developed a family of algebras to express relational operations and queries. Side note, although these days SQL has become synonymous to relational databases, Dr. Codd was very much against SQL since it violates some fundamental relational model principle. I'll

point out some of these violations later in this course. And what is a database? Well that's a trickier one. Wikipedia's definition of a database states it is an organized collection of data. While this is technically correct, it is like saying that a person is an organized collection of carbon-based molecules. Technically, correct, but not very useful. If you're interested in some more background current trends and history, see the link provided in the exercise file. The truth is that databases are much more than that. In the past few decades, data and databases have become the most important asset of mankind. Data, databases, and data practice are changing the world in fundamental ways. Some say data is the new oil. Unfortunately, with the new oil, also comes the new snake oil and there is plenty of that around, but that's a topic for another course. To understand what we're dealing with here, let's examine SQL Server's architecture and services.

A Little SQL Server Architecture

I've been fortunate enough to witness SQL Server's development since version 6.5 way back in the 90s when the name reflected most of what it actually did. Today, many still use the name SQL Server, but refer to the set of services under its very wide umbrella. These services include the core SQL database engine and the machine learning integration analysis, reporting, replication, and many other services. In this course, we'll deal with the core SQL database engine. The core engine can be installed on Windows or Linux operating systems and in a Docker container. Most cloud providers offer SQL Server as a managed cloud product, and in Microsoft's Azure, you can also get an Azure SQL database, which is a SQL Server database in a Platform as a Service offering. I will use a local installation of SQL Server on my Windows machine. Whether your SQL Server is installed locally on the network or if you're using a cloud service, it is convenient to visualize the server service as an abstract entity that lives in its own little bubble. SQL Server is a multiuser server and the only way to communicate with it is via preconfigured communication channels, typically, TCIP ports. Users and application servers use the provided application programming interfaces, or APIs, to send queries to the server and receive back results over the network. SQL Server offers two authentication methods for users and application clients. A client can request to be authenticated via SQL authentication, which means that SQL Server will use the login credentials it stores internally in its system tables. The alternative is for the client to authenticate via Windows authentication. This requires an Active Directory infrastructure. When a user needs to be authenticated, SQL Server reaches out to the Active

Directory and offloads the authentication responsibility. This allows for single sign on scenarios so that users don't need to provide their password for every resource. The last aspect I want to quickly cover before we get our hands dirty is the object hierarchy within SQL Server as it may be different than what you're used to. The blue rectangle represents a single operating system environment. On this operating system, we can install one or more independent SQL Server instances. The first instance is typically installed as a default instance. Alternatively, we can use containers for this. Let's ignore the multiple instances for now as what I'll show you is going to be true for all of them. Within each instance, we can create one or more logical containers called databases. I'll focus on just one database for simplicity. Within each database, we can have several sub-logical containers called schemas, and within these schemas, we can finally create our objects. These could be tables, views, store procedures, and others. The hierarchy is quite elaborate, but it offers a lot of flexibility and management advantages. Now let's see how to access objects within SQL Server. Every object in SQL Server has a fully-qualified name that consists of four parts separated by periods, the server or instance name, the database name, the schema name, and finally, the object name. You can imagine what a hassle it will be if we had to write this fully-qualified name every time we wanted to access an object. And indeed, SQL Server allows us to omit parts of the full name and it will fill in the blanks for us with default values. Omitting the instance name will default to the current instance we're connected to. Omitting the database name will default to the current database context of our session, and I'll show you in a few minutes how to set this context. Omitting the schema name will default to the DBO schema for administrator accounts or the default schema, which was explicitly specified for the user. DBO stands for database owner and is a reminiscent of days past. Of course, the object name itself can't be omitted. You might be wondering at this point what all this has to do with query in T-SQL, but let me assure you that the past 10 minutes may have saved you hours of frustration if you haven't worked with SQL Server before. You'll see what I mean when we'll start using the client tools and actually writing queries. And now that we have some idea of what we're dealing with, we're ready to see the tools.

Tools

In order to follow along with the demos and practice by yourself, you'll need to make some choices. The first thing you'll need is a SQL Server instance to manage your data and process your queries.

And if you decide to go for a local install like I did, Microsoft provides you with two free editions to choose from, the developer edition, which is not licensed for production, and the express edition that has several capacity and feature restrictions, but either of these editions will do just fine for this course. Alternatively, you can sign up for a free Azure starter account and use an Azure SQL database or a pre-installed SQL Server virtual machine. If you opt for an Azure SQL DB, you might be able to get by with a simple query editor available on the Azure portal, but for a local install, you'll need an integrated development environment or IDE. The IDE allows you to connect to the server, execute queries, and get your results back. Microsoft provides two excellent free tools, the traditional SQL Server Management Studio, or SSMS, and SSMS is my favorite tool. I'll be using it for all the code demos. Another good option is Azure Data Studio and you can check it out on the Microsoft website. Alternatively, if you absolutely can't install anything and you don't have access to Azure or other cloud providers, you can use one of the online query services, such as dbfiddle, sqlfiddle, and the likes. My personal favorite is dbfiddle.uk mostly for its reliability and non-commercial nature, but any of the others that provide SQL Server will do just fine. Be aware that these online services don't persist your data so you'll need to create all objects and data every time you start a session. You can save and share the session's URL and your code with it. And of course, you'll need the exercise files. You can download the exercise files from the course's home page. You'll find all the links to the various tools and services that I mentioned in the exercise file for this module.

Demo: Tools

Say hi to SQL Server Management Studio. Every time you launch it, it'll ask you for the server name you want to connect to. I'll choose my local instance. If you're using a server on Azure, you'll need to provide the URL. Make sure you choose the correct authentication method to avoid frustrations and if you're a local administrator on your computer like I am, you can use the Windows authentication. If you're not, either use SQL authentication and provide your password or talk to your network administrator. On the left is the Object Explorer window. It provides convenient navigation to databases and most tools and wizards via right-click Context menus. Your Object Explorer might look somewhat different than mine. It depends on the services you have installed and on your permissions. You can dock, hide, close, and float window panes like in Visual Studio, and in fact, SSMS is a Visual Studio shell. The top section is your toolbar where, you guessed it, you can access all editor tools.

Let's open a new editor window so we can start executing queries. I can use the New Query button or simply hit Ctrl+N. I can open multiple editors at once and switch between them using the mouse or Ctrl+Tab. Each editor can connect to a different server. If you hover your mouse over the editor title, you'll see the file you're working on and the server you're connected to. If you ever mess up your window arrangement, you can always reset it from the toolbar. Let's begin by creating a new database using a simplified Create Database statement and call it NewDB. It's a good idea to terminate every statement with a semicolon, even if it's not always mandatory. To execute a query, I click the Execute button on the toolbar or hit F5. If I don't highlight anything, the entire script in the window will be executed. If I want to execute a subset of the statement, I might highlight them first, and only then, execute. On the top left is the current session's default database context. By default, it is set to the master database when I first connect to the server as an administrator. Master is a system database so don't mess with it. While I can still access objects from any database using the fully qualified name as we've seen earlier, it is a good practice to change the context for the session to the database you're working with. This will minimize the chances of accidently messing with objects in the wrong database. To change context to the newborn NewDB, I can either select it from the pull-down menu or issue a Use NewDB statement. Now we're ready to execute statements, but we'll do that in the next module. If you want to become more familiar with SSMS, there is a course in the Pluralsight library on this topic. It's a bit old, but most of it is still relevant. You know by now where to find the link.

Module Review

This module was a warm-up exercise just so we can set a baseline. We've covered some of the terms that we'll use and there is plenty more of these to come. We saw some of the basic architectural aspects of SQL Server, those that we'll need in order to work efficiently, and I showed you some of the basics of using SSMS. By the end of this course, you'll feel right at home with it. From this point on, we'll deal exclusively with queries and the select statement. As I said one, and I'll say it again, this course's focus is not on syntax. I don't bother remembering most of the syntax. What parameter comes first, where I need a comma or a semicolon and so on, is a waste of my precious, and unfortunately, dimensioning gray matter memory buffers. Many users are biased against online documentation, in general, as they tend to be over technical, hard to read, and often confusing, but don't let that discourage you. SQL Server offers mind-blowingly good documentation, much better than

most other database vendors with easy-to-read and understand detailed explanations and plenty of great examples. Even after 25 years of dealing with SQL Server, I still have the documentation open in front of me whenever I need to write SQL code and so should you.

Our First SELECT

Module Introduction

I'll use our first statements as an opportunity to practice the tools and cover some of the concepts and terminology that we'll be using for the rest of this course. We'll begin with an introduction to one of the most important concepts of SQL, query execution order. We'll learn about sets, expressions, operators, functions, aliases, data types, and much more, and we'll execute our first SQL statements SELECT without a from. So without further delay, let's get started.

What Is a SQL Query?

There is no better place to start than right at the end. What you see in front of you is a skeleton of a complete SQL SELECT query in all its fame and glory. Consider this as a map of the path that we're going to cover in this course. By the end of it, all these strange seemingly unrelated words will make perfect sense. The numbers in green represent the order of execution of the various clauses that make up a SQL statement. Every SELECT query begins at the FROM clause. This is where the source dataset is prepared and this set is the only data that will be available to all following clauses. Once prepared, the set is moved to the WHERE clause and the where applies a filter to eliminate rows from the set using predicates and predicate is a logical expression that can evaluate to either true or false. It is essentially a yes/no question. Every row gets evaluated using a predicate and any row for which the predicate doesn't evaluate to true is eliminated. The filtered set then moves onto the next phase, the GROUP BY clause. In group by, individual rows can be combined into groups based on grouping expressions. Expressions consist of symbols that evaluate to a value based on their context, one plus one, quantity multiplied by price, the square root of two, and so on. After grouping is complete, the group set moves on to yet another filter, the HAVING clause. Here, we can filter whole

groups from the set, instead of individual rows. This was not possible in the where filter as these groups were not yet formed. Next, the group then filtered set moves onto the SELECT clause. In the select, we provide all expressions that we're interested in sending back to the client application and this doesn't necessarily have to include all data from the source set. After select expressions have been evaluated, we can sort them with the ORDER BY clause and limit the number and offset of rows that will be returned with the OFFSET FETCH clause. I'll say it again, this order of execution that I just described, and probably doesn't make much sense yet, is the most fundamental aspect of SQL queries and you need to integrate it into your way of thinking about SQL and that's what we're going to do in this course. I can't emphasize enough how important it is and how much easier it will make your life. The order of the modules in this course also reflects this execution order. Now for more of your favorite parts, terminology.

More Terminology

A set is a collection of unique elements and it is one of the most fundamental aspects of the relational model, and consequently, of SQL. All elements of a set are unique and distinguishable from one another, a rule that SQL doesn't always enforce. Elements have no predefined order. A set can be one of our customers, world countries, books in a library, or orders. In the relational model, these are called relations. Elements of a relation are called tuples and their counterparts in SQL are rows. This implies that rows are unique and have no order. Uniqueness of rows in a table is guaranteed by a key. A key is an attribute or a set of attributes that uniquely identify each row. The key can be a person's name, a book's ISBN, or an order's number. Tuples consists of attributes and their SQL counterparts are the columns. Columns require unique names so we can reference each one individual. While attributes in the relational model have no order, SQL violates this rule and allows referencing columns by their original position in some cases. But what I do what you to memorize is that a set by definition has no order, neither the rows nor the columns. Aliases in SQL are friendly names that we can assign to hold datasets or to individual expressions. When referring to a dataset of a named object such as a table, by default, the alias is the object's name, the employees, books, products, or countries table. It is convenient and sometime necessary to assign an explicit alias to a dataset. Columns of tables and views also have a name. By default, this name is used as their alias, C1, first name, order date, ISBN. We can assign a new alias to an existing column and to an expression as any expression involving

more than just the base column no longer retains the column alias. First name and last name together can be aliased as a full name pie r-squared as a circled area, and so on. SQL is a strongly statically-typed language. This means that every expression in SQL has a specific data type. T-SQL offers several families of data types. The numeric family offers integers of varying sizes, decimals, and floating-point types. The string family offers varying length and types of strings encoded in either ASCII or Unicode. T-SQL binary types are technically considered part of the string family, but they do have their own distinct characteristics. We usually use binary types to store media like images, video, and audio. Temporal types deal with representing time-related attributes. These can include dates, time, a combination of date and time to represent a specific point on the time continuum. T-SQL provides several special use case types that support representation of hierarchies, spatial data, and the so-called unstructured data types such as XML and JSON documents. If you've ever used other database management systems, you'll find that there are two very important types that are still missing from T-SQL, an interval type and a Boolean type. And for those of you who just jumped from their seats and shouting, but T-SQL has bit, well the answer is no. Bit is not a Boolean type, it's not even close. It's the best T-SQL currently has to offer, but it's not a Boolean, it's a pure numeric. Operators and functions are used to create new expressions from existing ones. And T-SQL offers numerous operators and functions and even allows for creating user-defined functions. You should be familiar with most of these from other programming languages, and in most cases, they serve very similar purposes. I'm not even going to try and list all of them. I'll say it again, this course is not about syntax. To learn more about available operators and functions, refer to the documentation. I will, however, showcase some common ones in the demo, which is coming next.

Demo: First SELECT

Even though I said every query processing starts at the FROM clause, T-SQL does allow for SELECT queries without one. This is not standard SQL, and if you happen to come from an Oracle background, you're probably used to adding FROM Dual to these queries. And you can think of T-SQL SELECT without a from the same way. Imagine that the absence of a FROM clause represents a dummy source set with a single row. Unlike Oracle's dual, this source set has no columns. Let's execute a SELECT with a literal string value expression X in string delimitators. This will return the single row with the single column and the value string X. Note that without the string delimitators, SQL Server will

try to evaluate X as an identifier as an alias for a source column which doesn't exist, and if I execute the query this way, I'll get an error. Expressions may consist of multiple subexpressions with operators. You can use SQL Server as an expensive calculator and use it to evaluate mind boggling expressions, such as 2 multiplied by 7 or the square root of 2. Separating select expressions with a comma causes each one to be returned as a separate column in the result set. See how the result set states there is no column name for either expression? I can assign an alias as part of the expression definition using the AS keyword. Let's add aliases, execute the query again, and now you can see that our fancy expressions do have an alias. Earlier I said that every expression in SQL has a static data type. For literal constant expressions consisting of sub expressions and operators, the resulting data type depends on the individual types that form the expression. There are precedence rules that you should be familiar with to avoid errors. Look at the literals 2 and 7, both are integers, and so their multiplication also results in an integer, but the same is true for division. So 7 divided by 2 will result in the integer expression 3 and this might not be what you expect it to be. Most people would expect the decimal value three and a half. To get the decimal result instead of an integer, we can introduce a dummy decimal to the expression such as multiply by 1.00, but we must be careful as the order of operations will affect the results as well. Without parentheses, SQL Server first divides these integers, and only then, multiplies the integer results by the decimal and the result is three again. In order to force the order so that the division will be a decimal 1, we need to use parentheses, but T-SQL provides the CAST function to explicitly convert expressions from one type of another and you should use them instead of trying to introduce such dummy operators. It's much better to use CAST 2 as decimal and then divide. If we construct foolish expressions, such as trying to add a string and an integer, SQL Server will attempt to unfoolish it by converting one of the expressions to match the data type of the other. In this case, SQL Server attempts to convert the string to an integer and these hidden conversions will end up being an endless source of grief and bugs in your code. See what happens if the string happens to contain any non-numeric characters, we get a conversion error. To add to the confusion, the plus operator means different things for integers and for strings. For strings, plus means concatenation. Always make sure your expressions have explicit and consistent data types. In the exercise file, you'll find the link to the data type precedence topic. Print it out and hang it in your workspace. It'll prove to be very useful. Date and time values are coded as strings and converted by SQL Server to their base types. String to date conversion depends on your locality and

the defaults for your database. If you live in the United States where date representation defeats any common sense, you may run into challenges if you're not careful. So always use the ISO format, YYYY and MDD, for expressing dates. We will cover more terminology and more gotchas in the following modules.

Module Review

This was our first taste of query execution order, a topic I promise to hammer in relentlessly. I'll say it again, it is that important. We covered some of the important terminology that we'll use for the rest of this course and for the rest of your SQL career. We learned about sets and how they relate to SQL, we've seen what expressions, operators, and functions are, and how to use aliases. We saw that SQL is a strongly statically-typed language and that it offers the most common data types and then some. We saw a demo of how to use these constructs in a SELECT query without a FROM clause. Now the basic foundations are set, and we're ready to begin our dive into query execution.

The FROM Clause

Module Introduction

Our trip continues with a first step in every query execution, the FROM clause. We'll begin to realize the steps that involved with the evaluation of the source datasets of any query. We'll see how to select from a single table and what steps are involved when the data that we need to process resides in multiple tables.

TSQL Demo Database

In order to execute queries, we must have a simple database to work with. I chose to use a highly-simplified electronics retail database for this course. You'll find the script to create this database in the exercise files. If you look at the script, you'll see that even though we didn't cover CREATE TABLE and the INSERT statements, these are mostly plain English and you can use the documentation to fill in the blanks. The customers table uses the customer's name as its key and also contains its country.

The key is represented with a key icon in a yellow name to distinguish it from the non-key attributes. Now this design probably won't work in a real-life scenario as it means that we can't have two customers with the same first name, but for our purposes, this simplification will actually be very beneficial. You might have noticed that the customers don't have a surrogate key, something like a customer id or a customer number and that's deliberate. I believe that the habit of using surrogate keys or ids for every table is one of the most destructive habits ever to plague the database world. Although, you are very likely to see it used pretty much everywhere, I hope that the experience with this demo database will show you that there is a flip side to the coin. The orders table does use an autogenerated order id as its primary key and the reason is that for orders the id makes sense since once the order id has been generated in the database, it is no longer used just there. Order id is used in the real world to identify the order by both customers and the seller and this means that the order id is now a valid attribute of the order itself. That was not the case for customers as a customer id would never be used to refer to a flesh and bone customer out in the real world. The blue arrow pointing from the orders table to the customers table represents a SQL foreign key. A foreign key is the counterpart of the relational models referential integrity constraint. It enforces the rule that only valid customers, those that exist in the customer's table, can place orders. The key for the items table is the item name and the second attribute is its color. This too won't work so well in the real world, but for our simplification purposes, it will do fine. Note that with this design, if you want to store multiple items with varying colors, you'll need to change the key to a composite one consisting of both item and color, but in our retail shop, we only sell white headphones, black amplifiers, and blue cables. Lastly, the order items table that stores all the items that were sold in each order, the key for this table consists of both order id and item name. This allows multiple items to be purchased in the single order. Order items has two foreign keys to ensure that we only sell valid items that exist in the items table and that the order id is that of a valid order. The data that you see on the slide is the actual data that we'll use in this database and you will quickly become familiar with it in this simple tiny database will prove to be very helpful for training purposes. Don't worry if these keys, foreign key, check constraints, create, and inserts don't make complete sense yet. You'll learn more about them in the following courses on this learning path, but we must start with some database for this course as well. To create this database, open the T-SQL demo db file and execute it in full. We are now ready for our first real query.

Demo: FROM Single Table

Let's begin by viewing the content of the customers table with a query SELECT * FROM Customers. The *, also known as an asterisk sign, is a shortcut that tells SQL Server we want to retrieve all columns of the source set. While it is a convenient shortcut for training and testing, never use it in production code. For our training purposes, the star is perfectly okay. What I want you to always have in mind is order of execution. First, the customers table is evaluated by the FROM clause, from there, it moves onto the next phase, which in this simple query is the SELECT clause. The SELECT evaluates all expressions for each row from the source set to construct the results. Following this simple rule, pause the video for a minute, and guess what will be the result of this next query. Select the string literal Pluralsight from Customers. Let's think like SQL Server and follow execution order. First, the customers table is evaluated in the FROM clause. Then this set moves onto the SELECT clause, which as we said, evaluates all expressions for each row. Expressions can be columns from the table like in the previous query with a star or a literal constant like the string Pluralsight in this query. Pluralsight is a valid expression, even though it doesn't refer to any columns and the rules apply to it exactly the same way. It is still evaluated for each row just like any other expression. Now let's execute the query and see what happens. Does it make sense? It might be easier to visualize this if we add back the original columns from the source. Let's execute again. Does it make more sense now? Let's look at the orders table selecting all columns and all rows with their respecting column names. Note that if I add an operator, even a meaningless one like adding 0 to OrderID, the resulting column no longer inherits the name of the original column. Therefore, we need to provide an explicit alias using the AS keyword. I can also provide an alternative alias for an existing column, such as client instead of customer. The two key takeaways from this demo are following execution order starting with the FROM and moving on and aliasing with the AS keyword. It may seem trivial now, but you will thank me for it once things start to get complex and that's going to happen quickly.

Introduction to Multiple Tables in FROM

Sometimes the data that we need for a query is not stored in a single table, but instead, it is split in multiple ones. We may want to show orders together with the customer's country or the item that was ordered with its color. In both cases, the data doesn't exist in a single table. In order to do that, we need a logically consistent way to connect or join these separate data sources together. Let's first see

a visual illustration before jumping into SQL. Here, we have an orange set and a blue set and we need to join them together for a query. The character elements represent a whole row in a table such as a specific customer or an order. Every join begins with a cartesian product. A cartesian product means that every element from the orange set is matched with every element from the blue set to create the green set you see on the slide. This is known in SQL as a cross join. Orange set cross join blue set results in the green set. For a SQL cross join, processing stops here and the cartesian product moves on to the next phase, but cross joins are not often useful. Typically, we only want to match rows from both sets based on some common denominator. We might want to see a customer with his orders or an order where the items that were sold in that order. We rarely need to join a customer to all orders or an order to all items. This is known in SQL as an inner join. An inner join begins with a cartesian product from the previous step and requires specifying a matching condition called a join predicate. This will be used to evaluate each pair of elements from the cartesian product. Only pairs for which the predicate evaluates to true will be kept and all others will be eliminated from the set. In this example, the predicate states that the characters need to be the same in both sets and the only pair that evaluates to true is DB, and therefore, it's the only one that's kept and all the others are eliminated. Inner join processing stops at this point and the query continuous with only the matching pairs being passed to the next phase. These could be, like we said, a customer with his order or an order with its items. But what if we want to return all customers regardless if they had any orders or what if we want to show all orders with the items, but we also want to show items that were never ordered. To get these rows back, a third step is needed. Let's say we want to keep all elements from the orange set regardless of whether they had a matching element in the blue set. To do that, we need to designate the orange set as a reserved set by stating a left outer join instead of an inner join. The left keyword designates the set on the left as the reserved set. When we specify an outer join, SQL Server goes back and reintroduces all of the elements or the rows from the orange set that failed to pass the predicate evaluation in the previous step with the inner join, in this case, these are A and C. Since all rows have the same structure in a table set, we must pair these elements with something. Since they have no matches, what would we pair them with? SQL Server has no choice, but to compliment all the reserved elements that we're reintroduced with a null indicating they had no match from the blue set. We will learn more about nulls in the next chapter, but for now, it's enough to say that it indicates a missing value. Let's see a short demo how this whole thing works with SQL queries.

Demo: Multiple Tables

Suppose we want to see the data about customers and their orders in the same query. Note that since I used customer name in the orders table because it's a key, I don't need to join back to the customers table if all I need is the customer's name. This is one of the main benefits of using keys that consist of real attributes, such as the customer name, instead of meaningless ids. These are called natural keys. If I need to get additional customer attributes such as the customer's country, it won't help me and I'll have to join the orders to the customers. The data no longer resides in a single table. The first step in any join is to create a cartesian product, meaning every customer will get matched with each order. Let's execute this cross join between customers and orders. Side note, since I used the SELECT *, I get two columns in the result set for customer, one that came from the customers table and another that came from the orders table. The result is no longer a valid set as I can't uniquely reference each column by name. This is one of the limitations of using star. And for a cross join, this is what will get passed to the SELECT clause. In most cases, we don't really want to see Bob's country next to Jack's order, which leads us to the next processing phase, the inner join. The inner join takes this cartesian product and adds a matching phase to eliminate these logically redundant rows. To do that, I need to specify the keyword inner, instead of cross, and provide the predicate to tell SQL Server what rows I want to keep. In this example, I want to see a customer's order next to his customer row. At this point, it starts making sense to alias the tables using a shorter name to make the query a bit more concise. I'll alias the customers as C and orders as O. C.Customers refers to the customer column from the customers table and the predicate states it should be equal to O.Customer, which refers to the customer column from the orders table. This is our join predicate. I want to see the customer row next to his orders. SQL Server evaluates this predicate for every row of the cartesian product and eliminates rows that did not evaluate to true. This leaves us with the desired rows. Jack's order is next to his customer row and so on. Time for another quiz. What do you think will happen if I use a join predicate that is always true, such as 1 = 1. Pause the video for a minute and see if you can guess. And I'll say it again, always follow processing order and your SQL life will be nothing, but fun. All joins start with a cartesian product. Every row from this cartesian product gets evaluated by the predicate. The predicate, in this case, will be true for all the rows. Therefore, none of the rows from the cartesian product will be eliminated. This silly-looking INNER JOIN is actually a cross join in disguise. While this is not something you'll use in a real query, it demonstrates the way that SQL Server processes joins

very well. Let's go back to the first inner join between customers and orders. The predicate matching of customers and their orders had a side effect. It eliminated all customers who never made an order from the results. Let's say we want to get these customers back so we need to tell SQL Server that we want to designate the customers table as a reserve table and we do that with an outer join. Changing the inner to left outer tells SQL Server to take the results of the inner join and add a final processing step to bring back all rows from the customers table, those that didn't have a matching order, adding no indicators for the columns of the non-reserve table orders. Executing this query brings back Kelly and Sunil with no indicators for their order attributes.

Module Review

To sum things up, in this module, we covered the first processing phase of every SQL query the FROM clause. We've met our demo database and saw how to select from a single table source. We covered processing of the three basic types of joins. The cross join created a cartesian product, the inner join eliminated rows that didn't match the join predicate, and the outer join designated a reserve table from which all rows were reintroduced back into the set. There is much more to learn about joins and it will be covered in depth in the following courses on this learning path. I'll say it once again, memorizing these three-simple processing phases will help you tremendously when you get there. It'll make the more subtle aspects you will encounter much clearer and easier to understand. Dealing with entire datasets in a single query is rarely useful. In most cases, we only need a subset of the rows and that's what the next module is all about.

Filtering with WHERE

Module Introduction

Now that we know how to construct our source dataset, it's time to move onto the second phase of query processing, filtering rows. We'll learn all about SQL's unique Ternary, or three-valued logic, we'll see what NULLs are about and how to handle them correctly for filtering purposes. We'll see how to

write logical predicates and we'll put it all together with a demo of filtering with WHERE clause. This is going to be a fun one.

Understanding NULLs

NULLs whose function is to represent missing or inapplicable data have always been the problem child of SQL. As you saw in our demo database, we store the country of our customers, but we may have customers that refuse to provide this information and we don't want to reject them because of it. This is a case of missing data. On the other hand, we may have fasteners in our inventory and one of the fastener attributes is thread size, but some of our fasteners may not have threads at all, meaning that the thread size attribute is not missing, it's simply inapplicable to these particular fasteners. Initially, the relational model had no such concept as missing or inapplicable data, and theoretically, we could design a database that requires no NULLs whatsoever, meaning it has no missing nor inapplicable attributes. In spite of Dr. Codd's resistance, and as is often the case, reality bites harder than theory and market forces often dictate language evolution in the standards. In 1975, only 5 years after its birth, Dr. Codd surrendered and introduced the concept of NULL and integrated it into his relational model. In 1986, NULL found its way into the first SQL standard and was forever set in stone. The concept introduced so much complexity and so many anomalies, one cannot even fathom how many man hours, dollars, and pulled out hairs were sacrificed to the NULL gods over the last five decades. What I do want you to memorize is that NULL is not a value. Zero is a concrete numeric value. An empty string is a valid string value. NULL, on the other hand, indicates the absence of the value. You can call it a marker, an indicator, a state, but never call it a value, and this is critical for understanding how we are about to deal with NULLs.

3VL (Ternary) Logic

You are familiar with two-valued logic from every day life. When you ask your kid, did you do your homework, you are presenting him or her with a predicate that can evaluate to either true or false. This is also known as a yes/no question. When you go to the pharmacy and check the expiration date on a box of aspirin, you're evaluating the predicate, has this product expired. This can evaluate to either true or false. With three-valued, also known as Ternary logic, every predicate may evaluate to one of

three possible logical states, true, false, or unknown. If your kid pretends to be fast asleep when you present him with a question, you won't know if he did his homework. If the expiration date on the box of aspirin is covered by a sale 40% off sticker permanently glued, you won't be able to tell whether it expired or not. The missing piece of information leads to the inevitable answer I don't know. My favorite parable for explaining this concept is an old American game show called Let's Make a Deal. If you're not familiar with it, search on YouTube and you will see why capitalism is going bankrupt. In this show, a contender is invited on stage and presented with three closed doors. Behind these doors are prizes of significantly varying value. The prize can be a goat, a sofa, a new washing machine, or a brand-new car. Contenders get to pick a door to get the prize behind it without knowing what it is. The host starts asking contenders questions and offer them various deals in return for either switching to another door or taking a known prize instead of the unknown one. Now imagine yourself on stage as a contender and me as the host. All three doors are initially closed, and I ask you, is there a car behind door number one. Your honest answer is, of course, I don't know or no. Then I ask you is the prize behind door number one the same as the one behind door number two. Your honest answer is, again, I don't know as both are closed. If I ask you whether they are different, your only answer, again, is I don't know, so write down rule number one. Any comparison of two unknowns is always unknown. Now I open door number two and reveal a prize, a free trip to South America. Not bad. I ask you, is the prize behind door number one the same as the one I just revealed behind door number two? You answer, I don't know. You can't tell if they are the same because door number one is still closed. So write down rule number two. Comparison of any known value to an unknown value is also unknown, but what if I ask you a slightly different question. Is it true that you don't know whether the prizes behind doors number one and two are the same? To this, you answer true. I don't know whether or not they are the same. This is a yes/no question. You either know or don't know. And note that this has nothing to do with comparison. This is a state predicate. Write down rule number three. A state predicate can evaluate to true or false. I'll remind you of this when we get to the code. It'll help you a lot clear some of the confusion around NULLs. Here is a cheat sheet you want to print out and hang in your workspace. The two left most columns define two logical states named a and b. The remaining columns show the logical evaluation for these states for or and equal and not. Pause the video for a minute and make sure all this makes sense to you before we move on.

Logical Predicates and Operators

In addition to the standard and, or, not, and equal operators, T-SQL offers additional logical operators all, any and some, between, exists, in, and like. Let's see what these are about. The predicate X larger than all A, B, and C will evaluate to true only FX is larger than all of them. This can be stated as X larger than A, and X larger than B, and X larger than C. You can replace the larger than operator with any other comparison operator such as smaller than, equal, not equal, and it will still evaluate the same way. It is crucial to understand the logical breakdown of these predicates. For all, if any of the operands A, B, or C happens to be null, the predicate will always evaluate to unknown because of the end. For any or some to evaluate to true, X needs to be larger than any of the operands, but not all of them. So if one of the operands A, B, or C happens to be a NULL, the expression may still evaluate to true if X is larger than one of the other operands, but it can't evaluate to a false as that will require X being smaller than or equal to all of them and we don't know if it is larger than the NULL. Unknown or true is true, but unknown or false is unknown. If you still find this confusing, don't worry, you're in very good company. I still scratch my head occasionally with this logic. Just go back to the cheat sheet from the previous slide and take another look. In is a more concise way of writing X equals any or X equals some. IN simply doesn't require specifying the quality predicate, it is assumed. It breaks down to the same underlying logical expression and the same rules apply, but there is one logical pitfall that I see developers often fall for and it has to do with NOT IN. Unlike IN, NOT IN translates to an expression consisting of the individual operands, but not with an or between them, but with an end. So with NOT IN, if one of the operands happens to be a NULL, the entire predicate will always evaluate to a NULL. Can you see why? If not, don't worry. Go back and look at the cheat sheet once again. I told you it's going to be useful. X BETWEEN A AND B evaluates to true when X lies in the range between the boundaries and that includes the boundary values as well. I've seen SQL developers struggle with this too, especially when dealing with time ranges. Just remember that the boundary values are included and you'll be fine. LIKE is used for string pattern matching using wildcards. I'll show you a demo of using LIKE very soon. Of course, you can mix and match multiple predicates with and, or, and not and they follow the standard precedence rules. When using multiple predicates, always use parentheses to minimize confusion and prevent tricky logical bugs, it'll also make your code clearer and easier to read. I'll show you how to do that in a few minutes. The only missing piece of the puzzle is how we can check whether an expression is or is not a NULL. When comparing a value to a NULL or even one

NULL to another, the result is always unknown and we do need a way to confirm whether any particular expression is a NULL. Remember the South America vacation you didn't win not long ago when you were on stage with me? Remember that after asking you about the prize behind the closed doors to which you answered I don't know, I phrased a different question, is it true that you don't know if the prize behind the closed door is the same as the one you see. The latter was a completely different logical predicate. What I asked you was a state question. Is it true that you don't know to which you could confidently answer true, I don't know what's behind the closed door. The exact same trick applies to NULL predicates. Instead of comparing values checking whether they're equal or whether one is smaller or larger than another, SQL offers us a state predicate called is NULL and its counterpart is NOT NULL. This is not a comparison predicate. It is a state predicate and that can evaluate to either true or false, but never to unknown. X = NULL, not equals NULL, larger, smaller, or any other comparison to NULL will always evaluate to unknown no matter if X is NULL or not, but X is NULL will evaluate to true if X is indeed NULL and X is NOT NULL will evaluate to true if X is a known value.

Filtering with WHERE

After this long, but important introduction, let's see how where works. You might have forgotten by now that where is the topic of this module. The FROM clause passed on a single set of rows constructed from one or more data sources and that got handed over to the WHERE clause. The WHERE clause, which consists of one or more logical predicates such as country = USA. The predicate is evaluated for each row of the set, and for each row, it may result in either true, false, or unknown as we saw earlier. If it evaluates to true, the row gets to live on to see another phase. If the predicate evaluates to either false or unknown, that row is eliminated never to be seen by any subsequent processing phase. Okay, that's enough theory. Let's get some code action. SELECT * FROM Customers WHERE Country = NULL returns an empty set as expected, but in order to strengthen execution order, let's follow it once again. The customers table got evaluated by the FROM clause and then was passed onto the WHERE clause. The WHERE clause evaluated the predicate Country = NULL for each and every row, and as you know, comparison to NULLs always evaluate to a known, so not a single row made it through the filter. This filtered empty set got moved to the SELECT clause, which returned all column expressions, but no rows. Changing the predicate to IS NULL returns Bob's row as Bob is the

only customer with a NULL country attribute. Changing the predicate to IS NOT NULL returns everybody, except Bob. To see all orders within a range, we can use the between operator. For example, WHERE OrderDate BETWEEN January 1st and January 3rd, 2019. This query returns three of our four orders. Note that BETWEEN filters out the NULLs as well. If I select all customers whose country name lies between A and Z, Bob doesn't show up. To see the item attributes for a turn table in an amplifier, I can use the IN predicate. SELECT * from Items WHERE Item IN string Turntable string Amplifier. To see all items, except the turn table in the amplifier, I can change the IN to a NOT IN. Let's execute, and indeed, here are all our other items. Now if I add a NULL to the operand list and execute the query with the IN again, it doesn't affect the result when using the IN predicate. It still returns the rows for a Turntable and an Amplifier, but for the query using NOT IN, if I add a NULL operand to the list, this will cause every single row to evaluate to unknown, and when I execute it, I get an empty set. You might be dually wondering why anyone in their right mind would add a literal NULL to an IN list and you would be right. No one would, or at least, should. But SQL is a highly composable language, and that means that where literal constant expressions are allowed, so are queries that return expressions and these may return NULLs, but I'm getting a bit ahead of myself so I'll stop. As you progress through the courses in this learning path, you'll see for yourself how important this logical distinction is. Let's see a simple wildcard stream match. Let's say I want to see all items whose name begin with an A. I can do that with the like predicate using the percent wildcard. The percent wildcard denotes 0 or more characters, any character. Executing this query returns both Audio Cable and Amplifier. If I want to see all items that have the character N in their name, I can use two % wildcards, one before and one after the n for the pattern. In this case, the like will evaluate to true no matter where in the string the n character appears. Let's execute it and we get back the headphones and the turntable.

Module Review

In this module, we learned about SQL's three-valued logic and how every predicate can evaluate to either true, false, or unknown. We learn about NULL and the anomalies that it introduces, and if you want to read some more about nulls, visit the Wikipedia page for SQL NULL, but I suggest you may want to wait a little bit, at least until you finish this course and get a bit more practice, otherwise, it may just confuse you. We learned how to use common logical predicates and operators, how to handle

NULLs with IS NULL and IS NOT NULL state predicates, and how to use all of the above to filter out rows with a WHERE clause. Now we know how to construct datasets and how to filter them, but that's not enough. You are now ready for the next level processing where we'll change the form of the dataset to get additional types of insights. This might be a good time for a 10-minute coffee break if you haven't had 1 yet.

Grouping Rows

Module Introduction

Now that we've constructed the dataset and filtered it, we can move onto the concept of grouping and how it is processed by SQL Server. We'll learn why we need groupings at all, how grouping fits in the general scheme of query processing, and about the logical limitations grouping introduces for the rest of the query. We'll see how to use the GROUP BY clause to create groups and how to use the HAVING clause to filter them.

What Is Grouping Good For?

Relational databases store granular facts about real world entities. An orders table typically stores all information about every order ever made. The customers table consists of a row for each customer with all its attributes. So far, in the FROM and WHERE clauses, we dealt exclusively with those individual roles and had access to all of their details. This is about to change. In the slide you see, there are many people. Imagine that these are our customers and each one of them is represented by a row in the customers table. With only the FROM and the WHERE clauses, we could ask questions, such as what are the attributes of customer John Doe, or what are the names of all customers that wear blue pants, or what are the names of all male customers that are 6 feet or taller. Granted, we have these attributes stored in the database, of course. These are all questions that can be answered by getting the source set, the customers table, and filtering based on individual attributes such as name, pants color, gender, and height. The answers to these questions are individual rows as well and consist of the original attributes for each customer, but sometimes these details are not what we're

after. Instead, we may want to get higher level insights about a set of rows representing a group of customers, which is based on some common denominator. Questions such as how many people wear each color of shirt or what is the average height of customers from each country. These types of questions are fundamentally different than the previous ones in two ways. First, we can't answer them just by looking at the individual customers one by one. We must look at them as groups of customers. All customers who wear blue shirts in one group and all customers who wear red shirts in another. Second, the answers to these questions doesn't consist of the individual customer attributes as well. Instead, it's a single answer for the entire group. The number of answers to the question what is the average height of customers per country will be a single value or a single row for each country, even if we have hundreds or thousands of customers from that country. The output row will consist of the common denominator, in this case, the country name and an additional derived attribute, such as the average height of the customers from that country. It doesn't make sense to list individual customer's names, heights, or shirt colors at the same time since we only get one answer row per country. And in each group, we may have customers with many different names, shirts, and pants colors, and heights. It doesn't make sense to ask for the group on one hand and for the details of the individual customer on the other hand at the same time. Once we group customers by country, we lose access to the individual attributes, except for the country, which is the common denominator for the group. The exact same logical limitation applies to SQL queries as well as we'll soon see.

Grouping Rows

The GROUP BY clause operates on the filtered row set that was passed from the WHERE clause. And I'll say it once again, you should always follow query execution order. At this point, these are still individual rows. The group by expressions define how we want these individual rows to be grouped and we do that by specifying the common denominator for the group. GROUP BY country will take all individual customer rows that have the same value for their country attribute and place them in a single group, one group for USA, one group for India, one group for China, and one for unknown countries. Now something interesting happens. As the groups are formed, the only values guaranteed to be the same for all the rows within each group are the ones that were specified as the group by expression. And from this point on, the queries working set will consist of just one row per group. In this case, we have four countries, hence four output rows. We can no longer refer to any column

which is not part of the GROUP BY expression directly. Any such expression may have multiple different values within the group like Jack and Kelly from the USA group. How would you return both in a single row? We can't refer to multiple values when the output row is just a single row. It doesn't make any sense like we saw with a customer's heights and shirt colors. It's easy to visualize it when you think about it as if the set transforms to this hybrid shape of elements within each group. One value for all the GROUP BY expressions, but potentially many for each of the other columns. The only way to reference a column which is not part of the GROUP BY expression is to instruct SQL Server to either pick one of the values from the group or calculate a single value based off them and this happens to be the definition of an aggregate function. Aggregate functions operate on a set of individual elements, but return just one value, just what the doctor ordered for our groups. You've been using aggregate functions all your life, even if you didn't call them by that name. Minimum, maximum, average, count, these are all aggregate functions. They take in a set of values and return one value. It could be one of the values from the source set like minimum or maximum or a calculated one like average and count. T-SQL offers additional statistical aggregate functions such as standard deviation and variance. They all follow the same basic rule. They take in a set of one of more values and always return just one. You'll learn more about groupings and aggregate functions and their different flavors in the following courses on this learning path, but for now, let's see a short demo. Let's build our query from scratch phase by phase following execution order. I'll start with a SELECT * of all Customer rows. Next, I'll add a WHERE filter to eliminate unknown countries and then add a GROUP BY Country. Executing this query as-is raises error 8120 stating that Column Customer is invalid in the select list because it is not contained in either an aggregate function or the GROUP BY clause. The reason is that the star translates to all source columns, which in this case are customer and country, and SQL Server is rightfully complaining it cannot show me the customer name as there are many multiple customers potentially in each country and the group by will force it to return only one row per country. I can no longer use the star after introducing the GROUP BY clause. I can refer to the Country, the group's common denominator, and use aggregate function on any of the other columns. Let's say I want to count how many customers we have in each country. For that, I can use the COUNT * function. COUNT * simply counts the number of rows within each group. In the following courses in this learning path, you will learn much more about the different aggregate functions and their subtleties. An interesting side note about grouping in NULLs, if I remove the filter from the query

reintroducing Bob's row, it now gets its own group with NULL for a country. But let's add a new row for Jane, another customer with a NULL country as well, and execute the query again, you can see that Bob and Jane got grouped together in a single group and this may seem odd as we learn that one null is never equal to another null, how come they're treated as equals? Well the truth is they're not really treated as equals in the mathematical sense. They are treated as being the same. Both are nulls and so they get grouped together. The state comparison, in this case, makes sense. Otherwise, we would get a different group for every single customer with a null country. I'm going to leave Jane's row in the table for now. We'll use it in the next demo as well.

Group Filtering with HAVING

After grouping is complete, we get an opportunity to apply another filter. The HAVING clause is closely related to the WHERE clause in the sense that it uses logical predicates to determine which groups get to stay for the ride and which gets eliminated. The only difference between having and where is when they occur. The WHERE clause took place right after the FROM clause, so it only had access to the individual rows. The HAVING clause takes place after the set has been grouped by the GROUP BY clause. It operates on the group set and can no longer reference the individual rows. This means it will eliminate whole groups at once and it also means that it can use aggregate functions for its predicates. Having can filter out countries with less than two customers and this is something we simply couldn't do in the WHERE clause. Let's see a short demo. If I want to see only countries with more than one customer, I'll add the predicate having counts start larger than 1. I couldn't have done it in the WHERE clause as it took place prior to the grouping. If I try to execute a query with an aggregate function in the WHERE clause, I'll get an error stating an aggregate may not appear in the WHERE clause unless, and don't worry about the exceptions just yet. Side note, another interesting point and often overlooked is that you don't have to use aggregate functions in the HAVING clause. It is perfectly legal to use non-aggregate expressions in the having predicates, but doing so is probably not such a great idea since after the grouping, without aggregate functions, we can only refer to the group expressions themselves. And filtering on the group by expressions would have been more efficiently done in the WHERE clause. It would have saved SQL Server the extra work of first creating the groups, and only then, eliminating them immediately after. Isn't it great how execution order plays out?

Module Review

A very important fact, not all relational database management systems respect the rules we discussed for group by as SQL Server does, and most notably, SQLite, which is one of the most widely-used relational database management systems, if you can call it such, allows referencing expressions without an aggregate function, even for non-group by expressions. The creators of SQLite came up with the most brilliant solution to the logical challenge of having multiple values per group. They decided that it makes sense to simply pick an arbitrary value from the group, instead of raising an error. They even have a name for this so-called feature, they call it bear columns. You can look it up. MySQL may suffer from a similar sin, but at least it doesn't do so by default. In my humble opinion, this so-called feature should be added to the list of mortal sins. Here, we learned why we need grouping at all and about the fundamental logical aspect of grouping. We saw how to use the GROUP BY clause, how SQL Server processes it, and how it changes the dataset so that we can no longer reference any columns directly, except the ones that are part of the group by expression. All others must use an aggregate function. We covered the HAVING clause and how we can use it to eliminate entire groups. As I said, there is much more to learn about grouping and aggregations. We're only touching the tip of the iceberg, but I believe that you will find this lesson very valuable when you reach the more advanced courses on this learning path and learn about the intricacies of dealing with groups, the various flavors of the aggregate functions, how to use outer joins and aggregates together without falling into traps, and we're now ready to advance to the SELECT clause.

Evaluating SELECT Expressions

Module Introduction

You might be wondering why I chose to have a dedicated module for SELECT when we've been using it all along. Indeed, this module is going to be a short one, but there are some important aspects that we need to cover. We'll see how select expressions are processed and how to deal with NULLs for

prettier presentation to client applications. We'll see how we can eliminate duplicates with DISTINCT and cover its logical implications.

Evaluating SELECT Expressions

Following execution order, SELECT takes place after HAVING. Much like the way we visualized it in previous phases, SELECT processes every row, and for each row, it evaluates all expressions all at once. More on that in a minute. These expressions may consist of the source columns, expressions based on these source columns, literal constants, functions, or any other valid expression. If the query had a group by, all the rules we discussed in the previous module regarding referencing non-group by columns only with aggregate functions apply, of course. As part of defining the expressions that will be returned to the application, it's a good idea to provide friendly aliases. We saw that in the first module that direct column references inherit the column name as their alias, but any operator or function revokes this inheritance. A relational set requires uniquely referenceable attributes and you should get into the habit of providing unique and friendly aliases for every expression. After all expressions and rows are evaluated, DISTINCT can be used to eliminate duplicate rows. ALL is typically omitted from the query and is assumed by default. With ALL or without writing anything, SQL Server returns all rows without looking for duplicates. More on that in a minute as well. You might be still wondering about the statement I made a minute ago that SELECT expressions are evaluated row by row, yet all at once. It may sound contradictory so let's clarify that first. For each row, all expressions are evaluated at once and this has logical implications. For example, you cannot use an alias that was defined for one expression in another expression in the same select list as they are all evaluated at once. The listing order of the expressions is irrelevant. As for rows, it is easier for us to think of the set as being processed row by row. The truth is that logically all the rows are processed as a single unit as well, but since this has no logical implications for query processing, it is easier for us to visualize it as if it was done row by row. Let's see a short demo of these concepts in action. We've already seen plenty of SELECT * statements, so let me a bit more verbose this time. Let's see all columns for the OrderItems table. SELECT OrderID, Item, Quantity, and Price from OrderItems. Executing this query demonstrates once again that all columns in the result set inherit the base column names as their aliases. Now if I'll add amount, which is quantity multiplied by price, and execute again, as expected, SQL Server no longer retains the alias of the source columns so I must provide an explicit one AS

Amount, nothing new here. I prefer to encapsulate expressions in parentheses just for better readability. If I try to use the alias amount that I just created in another expression, let's say 90% of amount, SQL Server will raise an error stating that it doesn't know what amount is. And now, you too know exactly why it is so.

Dealing with NULLs

We've already discussed how to deal with nulls for filtering purposes using is null and is not null state predicates. For presentation purposes, we can eliminate nulls from the result by replacing them with a more user-friendly value. T-SQL provides several functions to do just that. The most commonly used one is the IS NULL function. IS NULL is used to replace a null value with a literal constant or other expression, typically something like not available or the integer number 0 for numeric expressions. Do not confuse the ISNULL function with the IS NULL logical predicate. ISNULL is a proprietary T-SQL function. It is a specific case of the standard COALESCE function using only two operands. COALESCE can be used with any number of operands and returns the left most non-null or known expression. COALESCE is more portable and is supported by most major database management systems. In turn, COALESCE itself is a specific case of the more generic case conditional expression. Under the covers, both IS NULL and COALESCE are evaluated as a case expression. For SQL Server, IS NULL has some internal performance optimizations, not so much for select, but mostly when used as part of a predicate. NULLIF can be used to compare two operands and it evaluates to null when these operands are equal. We'll see a demo soon enough.

Using DISTINCT

After all expressions have been evaluated for all rows, the optional distinct takes place. The purpose of DISTINCT is to eliminate duplicate rows. Now technically speaking, a set that had duplicates to begin with is not a proper set. A row is considered a duplicate of another row only if all its elements or columns are equal. Like GROUP BY, NULLS are treated by DISTINCT as being the same, even though, technically not equal. In fact, DISTINCT and GROUP BY share a lot of similarities. Both take multiple source rows and group them into a single output row. The difference is that DISTINCT is applied after all select expressions have been evaluated and it applies to the entire select expressions

set so aggregates can be used and don't make much sense. DISTINCT doesn't change the shape of the set like GROUP BY does. It can only reduce the number of rows by eliminating duplicate ones. Let's see a demo of DISTINCT and with dealing with NULLS. Our task now is to return all countries where we have customers. If I use a simple SELECT Country FROM Customers, I'm going to get back all countries from all customer rows, and therefore, the USA and NULL country appear twice. This is, in fact, a SELECT ALL query, although you rarely see the all explicitly specified. To eliminate these duplicates, I have two options. I can either add DISTINCT to the left of the expression list or use a query with a GROUP BY Country clause. Let's execute both options, and indeed, the result is the same, but if I add Customer to the SELECT, DISTINCT will apply to both expressions effectively returning all rows since Customer is the unique key for this table. Usually, users don't like seeing the word NULL in their reports and I can use any of the conditional functions we saw earlier and replace it with a string, Country and NULL. Let's do it with ISNULL, execute, and there you go, a much prettier result. We also need to remember to add an alias using AS Country. Even though SQL provides plenty of functions for dealing with NULLs, there is a more fundamental question at play in my humble opinion. Is it the database's job to deal with presentation at all or should these NULL replacements be done elsewhere. We're going to see a similar dilemma in the next module and I'll try to answer both at the same time.

Module Review

In this module, we saw how select expressions are processed row by row, yet all expressions at once. We learned how to use the DISTINCT to eliminate duplicate rows and the similarities between DISTINCT and GROUP BY including how both treat NULLS as being the same, but not as equals. We saw how to replace NULLS with more meaningful values for presentation purposes. The relational aspect of query processing are pretty much done at this point. Is our query's job done yet? To find out, we need to step out of the relational domain altogether, which is exactly what we're going to do in the next module.

Ordering and Paging

Module Introduction

Our last module covers the last two phases of query processing, presentation ordering and paging, also called pagination. I'll begin by sharing my opinion regarding ordering, in general, and why it doesn't make sense to perform it in the database. We'll learn about the ORDER BY clause, which allows us to enforce row order and about query determinism and tiebreakers. We'll wrap up with a final phase of a select query paging result sets using both offset fetch and the proprietary T-SQL top operator.

Law of Order

You may recall from our first module that the definition of a set states it has no order. When order is implied on a set, it seizes to be a set and becomes a cursor. As such, it has no place in the relational model that deals exclusively with sets. A cursor in SQL is a row level operation. This is another aspect where SQL doesn't comply with the relational model. It is important to order a result set for presentation purposes so users can easily navigate the rows and there is no doubt about that, but there is a much deeper root architectural issue involved here. Whose job is it to arrange the data for a presentation? The most commonly used 3-tier software architecture paradigm separates the responsibilities for presentation, business rule processing, and data management to their respective tiers. Ideally, these tiers should be completely independent both physically and logically. Keeping the separation of responsibilities is critical for modularity, maintaining well-defined boundaries and interfaces, and it allows for easy upgrading of a single tier when requirements change or when a more suitable technology becomes available. So what does it have to do with SQL queries? Well I claim it is not the database's responsibility to deal with presentation and that ordering is the sole responsibility of the presentation tier. You wouldn't expect the database to handle font choices, paragraphs, colors, text box boundaries, indentations, and animations, right? Then why should it deal with ordering? My experience has shown that in many cases developers simply find it more convenient to throw an ORDER BY clause on the query instead of dealing with ordering in its rightful tier. SQL is such a convenient and flexible language and developers need to write the SQL queries anyway. So what's wrong with taking a shortcut and have the query order the result? More often, it is simply a default habit, and unfortunately, this habit may have significant implications on query performance, data consistency, and it may introduce bugs. Moreover, many developers use SQL to process business

rules in the database and I find this to be an even worse practice, but that's a topic for another course. That said, there are cases where ordering in the database does make sense like we'll soon see for paging, but these should be the exception, not the rule.

ORDER BY

Following execution order, ORDER BY receives a dataset after it was processed by the SELECT clause. Theoretically, the ORDER BY should only be able to see expressions that were defined in the SELECT clause. SQL Server and most other database management systems relax this limitation and allows for ordering by expressions, even if they were not part of the select. Since ORDER BY takes place after select, it can use aliases that were defined there and this is the only clause where we can use these aliases. Again, we see execution order at play. For each ordering expression, we can decide whether we want it sorted in ascending or descending order. If a sorting direction is not specified, ascending is used by default. In T-SQL, NULLs are always assumed to have the lowest ordering value. Other database management systems, such as Postgres SQL, for example, allow us to explicitly state whether we want NULLs to be giving the lowest or the highest ordering value using the NULL's first and NULL's last keywords. In T-SQL, you'll need to use a conditional expression such as the case that we've seen previously to replace NULLs with the value that is higher than all others. It's the only way to make NULLs sort last. To make things even more fun, the default for NULL ordering also varies between database management systems. The same Postgres SQL, for example, gives nulls the highest ordering value, by default, the exact opposite than SQL Server.

Determinism and Tiebreakers

A deterministic algorithm is an algorithm which when giving the same input will always produce the same output. This concept applies to SQL queries as well. A deterministic query is a query that when executed multiple times over the same dataset is guaranteed to return the same result. This property can be affected by many factors, but for our discussion, we'll focus on the ORDER BY aspect of determinism. When ordering by a non-unique expression, meaning that two or more rows have the same ordering expression value, SQL Server is free to return these tied rows in any order that it chooses. There is no guarantee that subsequent executions of the same query, even if the underlying

data does not change, we'll return the rows with equal ordering values in any consistent order. If you need to make sure that rows are always returned in the exact same order, it is your responsibility as a SQL developer to make sure that the ORDER BY expressions make up a unique ordering value per row. This can be done by explicitly adding additional expressions through the ORDER BY clause to make it unique. These additional expressions are often referred to as tiebreakers as it is their purpose to break the ordering value tie between these rows. We'll see a demo of using a tiebreaker in the upcoming demo.

Paging Result Sets

Paging is a technique often used for queries that return more rows that any human can process effectively. When you perform a Google search or an Amazon product search, you get back only a handful of results based on some ordering criteria such as relevance, popularity, or as the case in the query that you see, OrderID. When you're ready to move onto the next page, you click a button and the next set of results are retrieved and displayed. While there are valid arguments that paging, much like ordering, is not the responsibility of the database, in this case, I find that it may make sense, especially when dealing with very large result sets. The total cost of caching large dataset in the presentation tier may significantly overweigh the cost required to issue additional database queries by orders of magnitude, but your mileage may vary. T-SQL offers two ways to limit result sets. First is the top operator. The top operator is specified in the select clause before the select expressions, and after the optional distinct. It has been supported by SQL Server since the Sybase days. It is a proprietary operator and is not supported by other database management systems. TOP does offer some unique functionality, such as the WITH TIES, which allows TOP to return any additional rows that share the same sorting value as the last row, even if it means returning more rows than what was specified for the TOP. TOP can also be used with a percentage indicator of the total number of rows, instead of a fixed number, and can even be used without an ORDER BY clause, which means that SQL Server can return any rows that it chooses. But TOP has no support for offset specifications, and therefore, it cannot be conveniently used for paging purposes. The standard ANSI offset fetch clause for paging is supported by T-SQL. OFFSET FETCH is specified following the ORDER BY clause and I find it to be self-explanatory. Most other database management systems use the non-ANSI, but very common

LIMIT OFFSET syntax. There is no complex logic or confusing concepts here, so I think we can dive right into the demo. To retrieve all orders and

Demo: Ordering and Paging

sort them by order date with the latest order displayed first, I'll use the query SELECT * from orders, ORDER BY OrderDate DESCENDING. To retrieve all order items in ascending item order, I'll use SELECT * from OrderItems ORDER BY item with or without ascending. While in most cases I advise not to rely on defaults and to specify the full syntax omitting the ascending is such a common practice that I doubt anyone will find it confusing. SQL Server allows for ordering by expressions other than the ones coming from the SELECT clause. For example, I can select only OrderID and Item from the OrderItems table, but sort on quantity. This isn't used often as sorting by an invisible column appears to users more like a random order. Now let's see what determinism is about. The query Select * from OrderItems ORDER BY OrderID returns multiple rows for the same order. Since some rows have the same sorting value, SQL Server can return them in any order that it sees fit. The only reason headphones were sorted before the MP3 player for OrderID 1 is that there is no reason. This order can change next time I execute the same query. And under the covers, there is probably some hidden reason most likely an optimizer plan choice or a storage engine access pattern choice, but as far as we're concerned and for all practical purposes, order of rows with the same sorting value is completely nondeterministic and random and you should treat it that way. It is your responsibility if you so desire to add a tiebreaker expression to guarantee deterministic order. Adding item as a secondary sorting expression will guarantee determinism for this query as the combination of item and OrderID is the primary key of the table, and therefore, unique. If your query involves multiple data sources, GROUP BY, or composite expressions, it is your responsibility to make sure that the ORDER BY is unique if you want your query to be deterministic. The truth is that, in most cases, order determinism doesn't really matter that much. Well it wasn't really the database's job to begin with and a set, by definition, has no order. Lastly, let's say we want to return the top three items based on their total quantity sold in all orders. This is a good opportunity to revisit the methodology that I use to solve SQL challenges. I always follow query execution order, which is what I've been hammering into your head for the past couple of hours. Always start with a FROM clause. In this case, the FROM consists of only the OrderItems table as it has all the data that we need. If I were to use an item ID, I probably would have

had to join back to the items table to get the item name. I start with a SELECT * to be able to execute the query and see what I'm dealing with. For this query, we don't need a WHERE clause, so I move onto the next phase, the GROUP BY. The challenge states that we want to see the number of items sold per item, which immediately tells me that item is our GROUP BY expression. After introducing the group by to the query, I can no longer keep the star in the select. If I try to execute it, I'll get our familiar error 8120, the same one we've seen before. After the GROUP BY, I can only reference Item, which is the GROUP BY expression, and an aggregate function for all other columns. In this case, I need the SUM of Quantities, which will give me the total number of items sold. I'll give this expression a meaningful alias, NumberOfItemsSold, and execute it. Looking good. All that is left is to sort the results by the SUM of Quantities, and for that, I can use the alias I created, ORDER BY NumberOfItemsSold DESC. Let's execute it and looking good. Now for the last piece of the puzzle, limiting the result set to the TOP three items. Using TOP is very simple. I add TOP 3 after the SELECT and execute the query. And even though TOP works and is pretty simple, I usually prefer to use the ANSI standard offset fetch. I find it to be more readable and it has the bonus benefit of allowing paging. So let's remove the top and add OFFSET 0 ROWS FETCH NEXT 3 ROWS ONLY after the ORDER BY. Execute the query again, and as you can see, we get the same result. If I want to return the next page, all I need to do is to change the offset value to start from row number 3, instead of 0. Let's execute it again, and now you can see that SQL Server skipped the top three rows and we get back our least popular items, the MP3 player and the headphones. Please stop and look at these queries for a minute. About 2 hours ago, these queries would have seemed intimidating, and even if they were familiar, did you really understand how they were processed and how do you feel about them now?

Module Review

I intentionally left one last crucial myth about ORDER BY that needs to be debunked. A query without an ORDER BY clause is not guaranteed to return rows in any order. This may sound obvious, but I still see many developers believe an old myth that states that even without an ORDER BY clause, if an index of one type or another exists on the table, rows will be returned in index order. While empirical observations may occasionally seem to suggest that it is indeed the case, there is no guarantee it will be so the next time you execute the query. Never assume any order for rows unless you explicitly specify an ORDER BY clause, and even with an ORDER BY, a query may not be deterministic if two

or more rows have the same ordering values. I've witnessed too many cases I my career where this wrong assumption surprised organization in production sometimes with pretty devastating results. In the last step of our adventure, we left the relational model behind and covered presentation ordering and paging. I've shared my opinion regarding responsibilities for ordering, in general, and its placed in the software architectural stack. We learned how to use the ORDER BY clause to guarantee presentation order using ascending and descending ordering expressions. We covered the concept of query determinism and of using tiebreakers. Finally, we saw how to limit result set using TOP and how to perform paging using offset and fetch. This concludes the technical part of this course. Coming up next is a short summary and wrap up. You want to stick around for a few more minutes. I think you'll find the following module very useful.

Wrapping Up

Module Introduction

So here we are approaching the end of our short adventure together, but before we say goodbye, I want to do a quick review of what we've learned, sum up some key takeaways, the ones that I believe are most crucial, I'll try to solicit your feedback, and provide my recommendations for your next steps. I'll share with you a list of my favorite resources to advance your SQL career, and only then we'll say farewell.

Course Review

We began this course a few hours ago learning what T-SQL is all about and its roots in the relational model. We learned what SQL Server is, the editions that we can use, how to authenticate using SQL and Windows authentication, and how to access objects using fully qualified names. We covered some of the tools for editing and executing queries and focused on SSMS. In the second module, I introduced you to query execution order for the first time. I can't emphasize enough how important it is, but you probably had enough hearing me say it in this course. We've covered the terminology we used throughout the course and what sets our both in the relational model and in SQL. We covered

some basic data type families, such as strings, numerics, binaries, and temporal. We saw the most useful operators and functions and a demo of how to use some of them. Here, we got introduced to Jack, Kelly, Sunil, and the others. We learned about their shopping habits in our demo retail database. We learned how to use the FROM clause to construct dataset using single tables and how to join multiple tables together. We learned that each join begins with a Cartesian product, how inner joins eliminate pairs that don't match, and how outer join can get us back the rows from the reserved tables the one that didn't have a match. Next, we learned how to filter rows using WHERE. I introduced you to the controversial concept of null to represent missing data and how it introduces three possible logical states, true, false, and unknown. You also almost want a free trip to south America. We learned how to deal with NULLs using state predicates, and I showed you demos of using the most common ones, such as in, between, and like. Then we learned what groupings are good for, how they impact the result set and the limitations that they introduce. We learned about aggregate functions and how these can be used to return, move one value per group. Next, we saw how SELECT expressions are processed row by row, yet all at once. I showed you how to use aliases for expressions and how to use distinct to eliminate duplicate rows. We covered how to handle nulls using NULL replacement functions and CASE conditional expression. In our final module, I shared my opinion regarding the responsibilities of the software architecture tiers and why I think ordering should not be performed in the database. We learned all about the ORDER BY clause and how to page result sets using TOP and with OFFSET and FETCH. We saw how important query determinism may be and how to address it with tiebreakers.

Takeaway

You can probably guess what I'm going to say here as I've repeated it over and over throughout this course. SQL is based on the relational model. The relational model deals exclusively with sets. You must shed any procedural imperative mindset to really understand SQL and become a successful SQL developer. In fact, I believe that mastering the relational module is a must if you plan on taking SQL seriously. Query execution order is one of the most fundamental aspects of SQL. Every time you approach a SQL query no matter if you're writing a new query or trying to figure out somebody else's code, this should always be front and center in your mind and I hope that, in this course, I managed to plant the right seeds for that. NULLs are cursed pitfalls for many developers. I've seen cases where

developers were so afraid of dealing with NULLs that they attempted to sweep them under the rug by using things like empty strings or negative numbers and all sorts of creative inventions. These always ended up causing way more harm than good. SQL has all the means you could ever need to handle NULLs as long as you understand them and follow their logic carefully. Next, don't sweat over syntax. It should not be your main concern. The documentation remembers it all and there is no shame in looking it up online if you need to use a function or not sure which operator has precedence over the other. I still use the documentation extensively even after 25 years of writing and teaching SQL. Understanding the underlying concepts and foundations are vastly more important to your success. I've spent a lot of time

Feedback

and invested a lot of love and care in making this course, and I would like to kindly ask for your help to make it even better for the next revision. What was your favorite part about this course? What parts did you find challenging and how would you improve them? Do you think there was anything missing from this training you think should be added? What parts were too long or too tedious and what can be shorter? If you tell us what other SQL trainings you would like to see and enough people share your opinion, there is a good chance you'll get to see your wish come true. On the course discussion page, you can post questions, answer other people's questions, provide feedback, or just chat with me and the others. I'll be monitoring this page regularly and will be happy to answer any question you might have or just say hi.

Next Steps

So what should you do with everything that you've learned? The human brain is an amazing neural network and courses like this one inspire new pathways and memories, but if you don't practice, it will all be soon forgotten. Believe it or not, in 1 hour from now, you'll forget 50% of what you now remember. Within 24 hours, you're going to lose about 70%, and within a week, 90% will be gone. The only way to remember long-term is to practice, practice, and practice once again. Take any additional high-quality SQL training you can find, Pluralsight courses, articles, blog posts, whitepapers, YouTubes, and anything you can get your hands on. Mastering the relational model will provide you

with the most valuable understanding of the foundations which SQL is based on. It is not an easy topic by any means, but I find it to be one of the most fascinating subjects I've ever learned. I highly recommend that you get involved in the SQL community. SQL and SQL Server have a huge following and a vibrant ecosystem from Twitter channels and Slack chat groups to in-person user groups, trade conferences, there's always things to do, new things to learn, not to mention, meeting passionate and smart people who will help you keep you motivated and practice and learn some more.

Additional Resources and Conclusion

Pluralsight's library has hundreds of hours of SQL Server training and that's on top of this learning path, which you should, of course, complete. In addition, there are many more hundreds of hours of general SQL language courses. SQL Server's documentation doesn't cover just T-SQL, it includes excellent articles and how-to guides that you can use to improve your skills. If you find it too technical or can't find what you're looking for, communities, such as Stack Overflow always have plenty of experts anxious to share their knowledge with you. When posting on Stack Overflow and other forums, please take the time and prepare your questions carefully so that others can easily understand what you want. And always be kind and thankful to those who attempt to help no matter if you like their answer or not, and the ones that you do like, please mark them, thank the answerer, and upvote all the others. Wikipedia has a vast free knowledge base about technical subjects. I often use it for my research as a starting point and take the links from there. At this point, I would like to thank you for your time and your patience making it all the way here to this final slide. I hope you enjoyed taking this course as much I had preparing it, and feel free to drop me a note on the course discussion forum. I always appreciate feedback good or bad or even just a hi. Have a wonderful rest of your day, and I hope to see you in my next Pluralsight course.