Course Overview

Course Overview

Hi everyone. My name is Ryan Booz, and welcome to my course, Programming SQL Server Triggers and Functions. I'm currently the chief software architect at KCF Technologies by day and a blogger, author, and speaker usually focused on SQL Server and data analytics tools. In this course, we're going to dive into the fundamentals of using SQL Server triggers and functions showing how triggers allow you to validate and react to schema and data modifications while functions help you provide consistent methods for accessing the data across the application. Some of the major topics we will cover include using DML triggers to interact with data, using DDL and login triggers for schema management, best practices to get better performance out of your triggers, and creating user-defined functions that perform well in all circumstances. By the end of this course, you will have the skills and knowledge needed to administer and develop efficient, high-performing triggers and functions. Before beginning this course, you should be familiar with SQL Server 2008 or above, be comfortable with writing SQL code beyond basic select and insert statements, and be comfortable with SQL Server Management Studio. I hope you'll join me on this journey to learn triggers and functions more deeply with the Programming SQL Server Triggers and Functions course, at Pluralsight.

Validating and Modifying Data with DML Triggers

Version Check

Overview

Welcome to Programming SQL Server Database Triggers and Functions. My name is Ryan Booz. In this first module, we'll be talking about validating and modifying data with DML triggers. Let's get started. In this module, we'll be discussing what DML triggers are, their anatomy, and how you can use them to track changes within your data. We'll look at common use cases for triggers, look at the two kinds of triggers that you can use called INSTEAD OF and AFTER. Within those triggers, there are special tables called INSERTED and DELETED that let you manipulate and work with the data that's been modified in the database. And finally, we'll talk about trigger execution order and how you can specify one trigger to run first and one to run last. Let's go ahead and get started.

What Is a DML Trigger?

So first, what is a DML trigger? Well, DML stands for data manipulation language, and it's that vocabulary of standard T-SQL commands that you already know that attempt to retrieve data, modify data, or manipulate it, things like SELECT and INSERT and UPDATE and DELETE. Now I say attempt here for a very specific reason, which we'll see as we go through this module, but what you have to recognize is it's those specific kinds of commands that access and manipulate the data that we're talking about. Now data in a relational database is stored within tables, a concept we're used to, and DML triggers watch for data manipulation events. Now the event doesn't actually have to mean that data was manipulated, but that someone at least attempted it, and that is why we said in the previous slide that what we're looking for is attempted actions. So inserting, updating, and deleted. Those are the kinds of actions and commands we're talking about, and we can watch for those events and then react to them, specifically to the data that was modified. Now one quick thing to note here and something we will look at is that triggers can also be created on views, and it can be useful, although it's a little bit of an edge case, but something we're going to look at later and why you might want to do it. So what does this look like in practice? Well, let's say I take a simple INSERT statement, and I insert one row of data into this table. What happens if I have an insert trigger that's looking for that insert event? It would be passed that row of data, and I could do something with it within the trigger. And the same thing would occur with an update event. If I were to take the last three rows of this table and increase the quantity by one, the update trigger would be passed those three rows of data, and I could do something within the database. I might check a constraint. I might update data somewhere else. And, again, the same thing happens for a delete event. If I delete row number 3 in this case, the trigger is passed that row, and I can do whatever's necessary within the business requirements of my database.

Understanding AFTER vs. INSTEAD OF Triggers

So now that we have a general understanding of what a trigger is and what it's reacting to, let's talk about the two kinds of triggers that you can use within SQL Server. Now within SQL Server, there are two kinds of triggers, INSTEAD OF and AFTER triggers. In almost every way, they really are the same. The only difference is where and when they do their work, but they're working on the same data. So first, let's review the similarities and then look at their differences. The first thing you need to know is that if you don't specify exactly what kind of trigger you're creating, you're actually creating an AFTER trigger. It is the default. Both INSTEAD OF and AFTER triggers have access to inserted and deleted data, and both execute within the DML transaction that began this event, so the UPDATE statement, the INSERT statement, whatever that was. The real difference comes in these last two points, which speaks to the difference of when the data is actually passed to the trigger and the responsibility of the trigger to do something with that data. In an AFTER trigger, all the constraints have passed, and the data that's passed to the trigger is already considered good. The INSTEAD OF triggers actually called in place of the DML action. So when I invoke an INSERT statement, the database engine itself doesn't insert the data, it passes that data to the trigger to do something with. And if the data is to be persisted, it is the job of the trigger to do that. Let's look at what that really means, and it might give you an idea of when you would want to use one over the other. So in an AFTER trigger, it's the DML event that is modifying data that gets this whole process started, but all of the constraints are already passed. They're already looked at. And if all those constraints pass, then the data is saved to the table, and then, and only then, is the AFTER trigger executed. And what's passed to the trigger, again, are these special virtual tables called INSERTED and DELETED, which we're going to look at. INSTEAD OF triggers work at the very beginning of the process. They become the actual event. So the DML event that modified the data executes this trigger instead of actually doing the work itself. And so what happens is the data is passed into the trigger to work on that data, and it's the trigger's job to save the data and manipulate it if necessary. And it's at that moment that the constraints that are already present in the database are checked. And if all those constraints pass, then, and only then, will the data actually be saved. Now it's worth noting here that once that action occurs and the data is actually saved to the table, it might trigger another trigger, an AFTER trigger. So, again, the INSTEAD OF trigger is taking the place of that work that would be done by the database engine itself. Now you might be asking, how do I choose between an AFTER and an INSTEAD OF trigger? Why would I use one over the other? Well, I'd just like to present a couple of ideas of when one makes sense over the other to give you some guidance. Consider using an AFTER trigger when you really need to rely on the data that is passed to the trigger. You want to make sure that the database and the constraints that have already been set up have been passed and what you receive is valid data. You might also want to use an

AFTER trigger when the data that is being modified is going to be used in some way in that trigger to affect other tables. And so if constraints are necessary to make those modifications, you need that data to be persisted first, then an AFTER trigger is your only option. And honestly, to an extent, sometimes it's really worth simply saying use an AFTER trigger if there's no compelling reason to do the additional work of an INSTEAD OF trigger. However, INSTEAD OF triggers really are very useful in some specific cases. One is that they tend to actually run more quickly. Because they are taking the place of an event that saves data first, you're taking away one action within the whole transaction cycle. The second one is that sometimes you need to modify data that's coming into the table before it can be saved. Now this is often the case in legacy systems. Maybe you have an old connector that is consistently sending an old value in that no longer exists, but it's not worth updating that system right now. You can actually catch that data first, modify the bad value into something that's good, and then save it into the database. And then the third reason is, again, often with legacy systems, if you change a schema, for instance, oftentimes you'll need to put a view on the updated schema for older legacy applications. You can actually put an INSTEAD OF trigger on that view to help save data back into the new tables without updating the application. So this can be really useful in legacy systems.

Understanding the anatomy of DML Triggers

Okay, so now we've talked about what triggers are, we've looked at the two kinds of triggers that you can use and some reasons you might want to, now let's actually look at the guts and the inside anatomy of a trigger. The first thing to note here as we just look at how you create a trigger within SQL Server is that starting with SQL Server 2016, we got the very helpful CREATE OR ALTER statement. Previous to 2016, you couldn't create or alter with one statement on many objects within the database. So for triggers, you would first have to check to see if it existed before you could alter it. And then you'll notice we have to name the table, and then we have to decide are we going to do an AFTER or an INSTEAD OF trigger? And then what DML event are we looking at? Is it the insert, the delete, or the update event? Now what I'd like to do is talk about a number of points, things you must understand about triggers as you begin to think about you can develop them to be useful within your application. The first one, and this is probably the one that catches almost everyone out of the gate, this can really be a surprise if you're coming from another database system like Postgres. In Postgres, you can actually specify triggers as either batch triggers or per row triggers. So in SQL Server, you have to develop these knowing that you could get more than one row and act accordingly. What that means is this: When I insert that one line into a table, what is passed to the trigger is actually a batch of rows. It just happens to be one row. Same thing with the UPDATE statement. When I update those three rows, what the trigger gets passed is actually a batch, a virtual table of three rows. And no different with the DELETE. When I delete even just one row, what the trigger receives is a virtual table, a batch, of one row. The second thing you need to note is that triggers should not return data. SQL Server specifically says that you shouldn't do it. It's a deprecated action, something that will be not allowed later. So if you're not supposed to return data from a trigger, what can you do? Well, the answer is really almost anything else. You have access at the moment the trigger's invoked to your entire database system. And so you can select data, declare variables. You can check settings in another part of your database. The possibilities really are limitless; however, as we're going to see later, you really need to be very mindful about the work you're doing because this is all happening within the same transaction as that DML event that started this whole process. Now tables can have one INSTEAD OF trigger for each DML event. So you can have one insert INSTEAD OF trigger, one delete INSTEAD OF trigger, and one update INSTEAD OF trigger. However, you can have multiple AFTER triggers for the same DML event. Now why would you do this? Now often, people will create multiple insert triggers or delete triggers or update triggers because there are specific business needs that maybe don't apply to every database. Now if you do have to create multiple triggers for specific events on a table, do yourselves a favor, and please make sure that you name those triggers in a way that helps you understand what it is each of those triggers is doing. Next, triggers are nested by default, up to 32 levels. So every SQL Server instance, by default, will allow triggers to, by their very actions, invoke another trigger. That means that when I insert data into one table, if the trigger that gets called modifies data in a second table and that table has a trigger on it, then that second trigger will also fire. That's what nesting means. And, again, naturally it can happen up to 32 levels deep. After 32 levels, the transaction will be ended. You can also have what's called a direct recursive trigger where an event modifies data which somehow actually calls the exact same trigger again; however, this is a setting that has to be enabled within SQL Server and not something we're going to talk about much in this course. All nested triggers execute in the same transaction, and that means that when one fails somewhere in the pipeline, the entire transaction is rolled back. And finally, let's discuss execution order of AFTER triggers. Now recall, we said that you can have multiple triggers for the same DML event. So if I have multiple insert triggers, when I insert data or I call an INSERT statement, I don't have any guarantee which trigger will be called first, and this can be a real problem if one of the triggers expects data to have already been manipulated by a previous trigger. Now within SQL Server, you can specify a first and a last of that group of DML trigger events. Recognize that you can only have one first and one last trigger specified within a table, and all the other triggers will still execute in an unspecified order. And finally, you need to recognize that any time you modify a trigger that is specified as first or last, it will reset its order to none. That means that any time you call a CREATE OR ALTER or an ALTER trigger, that trigger, if it was first or last, will now have no order, so you need to be very diligent to call the stored procedure that resets that order every time you do a modification.

Utilizing the INSERTED and DELETED Virtual Tables

Alright, so now we've talked about what triggers are, we've talked about what you can do inside of them, but I'm guessing you still have one question. How do I work with the data that's been modified? I'm glad you asked. Remember that I've talked about these INSERTED and DELETED tables a couple of times. So when a trigger's executed because of a DML event, virtual tables are created with the data that was modified and passed into the trigger. The INSERTED virtual table contains the new data. The DELETED virtual table contains the old data. Update triggers always have both virtual tables, so when you update something within a table, both virtual tables are created, and you get to see what the data was, the old data, and what the new values are, the new data. So let's go back to our example and see what this would look like in practice. When I insert that row at the bottom of this table, this is what the trigger would see. It would see an INSERTED virtual table with that data passed in, and there would be no DELETED table. If I update those three rows of data, remember, I incremented the value by one for these three rows, and so what happens is I get an INSERTED table, which has the new values, 4, 11, and 13, and a DELETED table that has the old values, 3, 10, and 12. In the DELETE statement, deleting that one row of data would create a DELETED virtual table with that row, and I would have no INSERTED virtual table. Now this means that you need to be very mindful about how you work with this data. A simple UPDATE statement across a very large table that it affects all those rows, you will literally be passed a virtual table with two copies of each one of those rows. And so less is more. You need to be very mindful about what you're doing. Check that rows are actually modified. Don't do any work if nothing actually changed. Check for modifications on only specific columns. Maybe this trigger only needs to work if something was changed in a specific column. Find the differences between the old and the new data. Only work on those differences in an UPDATE statement. And always keep your transaction short lived. Be mindful that every time you create a trigger on a table for an event, every single one of those triggers will fire within the same transaction, which means that transaction is going to stay open longer and longer the more work you do. One way to get around these long- running transactions is using a tool called Service Broker within SQL Server. We're going to take a very brief look at this in module 3, and there are other courses within Pluralsight that help you understand Service Broker more. Being mindful of the work you're doing within triggers means that almost every trigger will have at least one of these statements at the very beginning so that if no data was passed in or no data that we care about, we won't do the work, and we won't keep that transaction open any longer than necessary. And I want to call your attention to two things on this slide. The first is that you will often see the UPDATE function used within update triggers. This allows you to see if a specific column was part of the SQL UPDATE statement. The one thing most people don't realize is that this doesn't actually tell you that the data was modified. If I simply set a value equal to itself, that doesn't matter. This function will still tell me that that column was part of the UPDATE statement, and that is why I often recommend considering something like this EXCEPT statement. This allows me to query the differences between the inserted and the deleted data to determine which rows actually had data modified. You can do it across all columns or just specific columns, and you'll see examples of that in the demonstrations. But the key takeaway here is that each of these methods and some others are useful tools to help you understand if further work should happen within the trigger or if you can stop at the very beginning and not do any unnecessary work. And that means that almost every trigger is going to look something like this at the beginning, and we'll look at this as we do the demos. We check the row count, we make sure nothing prints out from the trigger, and then we might do those additional checks on the virtual tables, again, just to ensure that we don't do extra work we don't need to.

Common Use Cases for DML Triggers

Now, before we get to the demos, I do want to talk about one last thing. Why would you use triggers? Quite honestly, there's a number of people that would say never use a trigger. There's too many possibilities for things to go wrong. It's too opaque. We don't understand why data is changing because it's happening within the database layer. And those are all really valid arguments, but they're really useful tools within a database when you're talking about multiple systems connecting to the same data source, all of which you might not have control over. I'm going to talk about five. First, one of the most common use cases that almost every system I know of uses triggers for is to log DML events. We want to know at some cases, really important tables, when something changed within that table so we can create a trigger that logs those kinds of actions and the data that was manipulated. Now, often we need to insert or modify data in another table because of something I did on this table. I insert a new customer. I need to make sure a record's created in a different table to prepare for incoming data from that customer. Sometimes we just need to make sure default business logic is always honored. If we must always have a value in a table and it's something that cannot be done with a normal constraint, sometimes we just need to do that. Maybe it's a calculation that needs to happen, but we can't make it a constraint that's normally checked by the database. Now I talked about this one earlier, and it is an edge case, but really useful for legacy systems. When we change a schema, you can actually create a view that mimics the old schema for legacy systems, and you can create an INSTEAD OF trigger on that view to help save and manipulate data from those legacy systems. And the last thing I've seen many legacy systems use triggers for is referential integrity. Now I'm going to tell you out of the gate you shouldn't do this. Many times those systems were created before we had standards like foreign key constraints. I would tell you never to use triggers simply to check other tables for the existence of data if a constraint can be used. So please don't use this moving forward, but you might see it in examples of legacy systems. And now that we've done all the work to understand what these are and how we could use them, let's actually get to some demos so you can see how these might be useful within your system.

Using INSERT AFTER Triggers

Now, in this first demo, we're going to look at the INSERT AFTER trigger and use this as an example to demonstrate how triggers work. Now remember that it's the DML action that causes the trigger to execute, not the actual data being modified. Also, all constraints have passed before an AFTER trigger is executed, so you're guaranteed that before this trigger fires, all the other normal constraints, foreign keys, and so forth have already passed. And any errors that cause a rollback within the trigger will affect the entire DML action. So if I have an INSERT event and something about the insert fails within the trigger, the entire transaction is rolled back, and that data is removed from the table. Now, in this demo, we're going to look at specifically just creating triggers and that process and how they work, and then we're going to look at how we could use an INSERT AFTER trigger to prevent incorrect data from being inserted based on some business constraints. Let's get started.

Demo: Creating an AFTER INSERT Trigger

Okay, after all that learning, let's go ahead and create our first trigger. We're going to start with an AFTER INSERT trigger. Now before I start the demo, a couple quick notes that will pertain to all the demos you see. I'm working on a fresh copy of WideWorldImporters from Microsoft. It's a free demo database that you can download and use yourself to recreate these demonstrations. Any time you see me highlight SQL and execute it, I'm usually doing it with the F5 key rather than my mouse, so you won't see that moving around. And when the results window pops up and it disappears, I'm using Ctrl+R to do that. So if you don't know that key stroke exists, that might be a helpful hint for you. So the first thing we're going to do is create an AFTER INSERT trigger, and this is going to be on the orders table, and it's simply going to let us know that it fired. So it's a very simple trigger. I'm going to select the code, hit F5, and now we can go over to the Triggers folder and see if that trigger exists. And there it is. So it did create the trigger, and it hopefully is ready to do what it needs to do. And now for the rest of these demos, I'm going to hide the database window so that we have more space to see what's going on. So now let's see what happens. I created a trigger, and I'm simply going to pretend to insert something. And that means I'm going to select something from the sales order table that doesn't exist. And this is to demonstrate that even though nothing actually gets inserted, the trigger is executed. Remember, we are always talking about events. The INSERT event occurred, the database is going to react to that, and the trigger is ready to do something. And so, as we talked about in the slides, one of the things you should do really in every trigger is validate that something needs to actually be done. So this is some of the boilerplate that we talked about, so I'm going to recreate that trigger, CREATE OR ALTER as we talked about before, and I'm going to run that exact same statement. Again, we're selecting from the sales order table an order that doesn't exist. We saw previously that it did execute, but now what happened is that the trigger did execute. We just checked to see if anything needed to be done, and we said, no, there's no rows that have been modified, so there's no reason to execute the rest of the code. Next, I'd like to demonstrate that concept that AFTER triggers only are executed if all of the other constraints within the database are met first. So this first INSERT statement, I am using a valid CustomerID, that's the first number, number 10, and we'll see that the INSERT INTO, it does get called. So we insert a row; the trigger is executed. Now if I use an invalid CustomerID, in this case I'm using 0, you'll see that the constraint fires first, so the trigger never actually got called because the database system stopped the rest of the transaction execution. Alright, so now let's talk about what it would look like to actually do something more useful within the trigger. So we actually want to create something that does work. In this case, we're going to put a constraint of our own on the database, something we couldn't do with a regular database constraint. We're going to check the incoming data and see if the customer is on a credit freeze or not. So we're inserting data into the sales table, but we're going to check the customer table to see if they're on credit freeze. And if they are, we're going to stop that transaction and roll it back. So the meat of that trigger is here where we take the values from the INSERTED table, we join it to the customer table, and we check if the IsOnCreditHold is true. And if it is, we roll back the entire transaction. Now the one thing I'll say here is this is probably not the best solution for a production database. In this case, if only one customer out of many are inserted and only one of them is on credit hold, the entire transaction would be rolled back, so you would want to think this through a little bit more, but this would demonstrate that we can join to other tables, and we can do those checks. Let's go ahead and create this trigger. So, again, we're doing a CREATE OR ALTER, so we're just simply recreating that same trigger. And now we're going to update a customer as if they were actually on credit hold. It's that same CustomerID I used earlier. So we'll update them. We'll try to insert data into that table using that customer. And now you see that the constraint's passed. There's nothing about these values that's wrong. We simply said in the trigger, if that customer has an OnCreditHold set to true, stop it, and that's exactly what we've done here. So that's one example of using triggers. Now we can add multiple conditions in here, again, things we might not be able to do within a regular constraint of the database. So we can add multiple checks. In this case, we have our OnCreditHold check, and now I could join to a different table. Hey, is the salesperson that you are saying was a part of the sale actually a salesperson? And if they're not, roll this back. So what I want to demonstrate here is that this is linear. If the first check fails, then the whole transaction stops there. We don't have to wait to get to the next check that we are doing. Let's go ahead and do that. So we now have the same trigger with two different checks in it. And we're going to take that customer back off credit freeze, so they could make purchases now, but I'm using a salesperson ID 9 that is not actually a salesperson. And you'll see that we pass the first check. We got to the second one that says, hey, you can't do this. This person is not a salesperson. But if I put them back on credit freeze and I once again run that INSERT, you'll see that the first check is what fails. Now we're getting a failure on the credit freeze, not on the salesperson. So there's multiple ways you can use this within your trigger for specific business constraints within your application.

Using INSTEAD OF Triggers

Now, in the second demo, we're going to look at INSTEAD OF triggers and how they work, and, again, we're going to use the insert INSTEAD OF as our example. Now remember, INSTEAD OF triggers act on behalf of the DML action, so data has not actually been manipulated in the table. They execute before any other AFTER triggers, and so they are the first ones in the line to do anything. No constraints have been checked yet. The raw data is passed to the trigger to do work with. It's only once that trigger saves data or manipulates data within the table that constraints are checked. And remember, INSTEAD OF triggers can be attached to views to proxy data to the tables in manipulation of data. And just like AFTER triggers, if anything within this trigger or event causes a rollback, the entire DML transaction is rolled back. Now, in this demo, we're going to look at how we can use an INSTEAD OF trigger to correct data before it's inserted. Now this might be an example of that legacy system. And then we're going to actually create an INSTEAD OF update trigger on a view so we can see how we might use that to modify data from a legacy application into a newer schema. Let's go ahead and get started.

Demo: Creating an INSTEAD OF Trigger

In this demo, we're going to look at INSTEAD OF triggers. But to demonstrate how they differ from AFTER triggers, I want to show you first how they get called and their relationship to one another. To do that, we're going to first create the AFTER INSERT trigger again. It's simply going to announce whether or not it was called. And then we're going to create the INSTEAD OF INSERT trigger. Now this looks exactly the same except for that part that says INSTEAD OF. And, again, it's simply going to announce whether or not it was executed. And so again, I'm going to select OrderID 0 from the sales order table to show that it doesn't actually matter if data is modified. It's the event, the INSERT event that triggers all these things to begin. So let's go ahead and do that. Let's see which one gets called. So we see here that it's the INSTEAD OF trigger that got executed first, and we don't see any reference to the AFTER trigger. Now remember, INSTEAD OF triggers fire in place of the DML event. And because this trigger did not actually proceed to insert data, the AFTER trigger wasn't called. No data was actually modified in the table. So hopefully that helps you understand the relationship between INSTEAD OF triggers and AFTER triggers. Let's go ahead and drop that first trigger so it doesn't get in our way. And now that we have the INSTEAD OF trigger, we can demonstrate a couple of other things that make them unique. Now I'm going to go ahead and try that same insert where CustomerID 0 doesn't actually exist. The constraint should fail. But because we have an INSTEAD OF trigger, it executes, and we get no constraint failure. Remember, no constraints are checked before the INSTEAD OF triggers are called. We have to do that action. So what does that look like? Let's look a little bit further, and we'll see that we could do the exact same check, is the customer on hold? If not, if all of our internal trigger checks are passed, we are then responsible for inserting that data. So we're the ones that need to do it. Let's go ahead and modify that trigger. Let's see what happens this time. I'm going to go ahead and set that customer back to being OnCreditHold, and let's see what happens. Alright, so this trigger at this point now works or appears to work the same. Now what makes this unique from our previous demo is that the AFTER trigger would have already had the row inserted into the table, so we would have had to actually roll back that log and that transaction. The INSTEAD OF trigger didn't get that far. It failed our check, and so nothing actually got inserted. And as I said in the slides, that's one of the reasons people might tend towards an INSTEAD OF trigger when performance is really something you need to consider. Alright, let's take them off hold, and let's try it again. And now we'll see that we passed our check, and one row was actually inserted into the table. We did our work, and the data got there.

Demo: Correcting Bad Data with INSTEAD OF Triggers

For this next example of how we're going to use an INSTEAD OF trigger, I wanted to give you a little context of what the demo is showing you. As I said at the beginning of the demo, this is often useful with legacy systems. In this case, we have a legacy system that's used to sending in this kind of sales order. And you'll notice that along the way we started including a new column called ExpectedDeliveryDate, and our new applications are using that date, and it's a required field. We want to make sure it's always filled in, but it may not be worth the cost, the development time, and effort to go back and fix the legacy system. This is the kind of thing we can fix in an INSTEAD OF trigger before the data gets saved to the database. That way the legacy system can continue to work, we can get the data we expect, and the business can continue to make sales. So let's look at how we can correct that data coming from the legacy system. To demonstrate what the problem is, I'm going to attempt to insert into the Sales.Orders table with a null date for the expected delivery. And you'll see that the constraint within the database fires and says that's not allowed. You need to have an ExpectedDeliveryDate filled in. So to solve this problem, we're going to create an INSTEAD OF INSERT trigger, and we're going to talk about how we can catch this specific scenario. The beginning of the trigger looks exactly the same as all of our other triggers so far. We still have the credit freeze check in here just in case that customer is on a credit freeze. But further down, we do a couple of new checks. First, we do a SELECT 1 FROM INSERTED WHERE the ExpectedDeliveryDate is NULL. This will simply tell us if any row within the inserted batch has a null ExpectedDeliveryDate. If it does, we then select all of the data from the INSERTED table into a temporary table. This allows us to modify the data before we prepare to insert it. And finally, knowing that there's at least one delivery date that is null, we update that temporary table and set the ExpectedDeliveryDate to today plus 10 days anywhere the date is null. Once we've done that, we can do the same INSERT statement we saw previously, and now we shouldn't get that constraint exception. Let's see what happens. With the trigger created, we'll try the exact same INSERT statement with a null delivery date, and now we get no constraint error. That means the data passed our constraint, we inserted the data into the table, and it had a new date. That's one quick example to show you how using an INSTEAD OF trigger, particularly with legacy systems that you know might be sending bad data, can be a really useful tool in your SQL tool belt.

Demo: Updating Data through a View

For this last demo, we're going to talk about saving data through a view. And, again, this is often really useful for legacy systems that are expecting a table or a view to be accessible that has changed over time. So we have our system again, and it used to be able to save an address for a customer by simply passing the CustomerID in and providing the address details. Along the way, we updated that address to be stored in three separate tables as the system grew. So what we will do is create a simple view that extracts the data from these tables, presents it in the form that the legacy system is still expecting so that when it runs the UPDATE statement we intercept the data, we correctly save it to the underlying tables, and now the system can keep running, and the legacy system can continue to be used. Okay, so let's see what this looks like in practice. The first thing we'll do is create that view of the customer address data. We'll verify that it's working, and now we can create the trigger on that view to update the data behind the tables. Now we're only going to create an INSTEAD OF update trigger. We could make it an INSTEAD OF UPDATE and INSERT and do a little bit more work, but for the sake of this demo, we're simply going to do one DML event. The top of the trigger looks exactly the same as the others. We check to make sure data is actually being modified. And now we do something unique. We create a temporary table that mimics the data coming in through that view. Once we have that table created, we insert everything from the INSERTED virtual table into the temporary table. This gives us a temporary workspace to modify data values before we do the insert. Once we have that completed, we first update the DeliveryCityID in the temporary table in preparation for finally updating those three tables behind the scenes. And once we do that, it is the job of the trigger to complete the DML event, actually creating and executing the UPDATE statement. Once that's completed, we can create this trigger. And now we can try to do an update. What we're first going to do is look at a customer. CustomerID 10 currently has an address in the zip code 90061. It's expecting that simple table. And so what we're going to do is mimic that legacy system by running an update event, passing in the address lines, postal codes, city, and state, the textural fields, and recognize that the trigger will go find the proper IDs and put this data in the right place. We run our statement, and now we check to see if the other tables were actually updated correctly. And sure enough, the address was changed, the postal code was changed, and the DeliveryCityID was now updated to Bell, California. Hopefully this gives you one more example and idea of how using something like an INSTEAD OF trigger on a view could help you transition legacy systems and other processes.

Using DELETE Triggers

Now let's talk about DELETE triggers and give you a couple examples of why you might want to use them. Now the first thing to recognize and just remember is that within a DELETE trigger you will only have access to the DELETED virtual table. And the other point I just want to bring forward, because I've seen this happen a number of times, is that recognize that specifically with an AFTER trigger the data has already been deleted from the table, and so if you're not thinking carefully, you might want to join to another table within the trigger to check something, but that data is already gone, so the join won't work, and you'll never get the data you expect back. As we look at DELETE triggers in these demos, we're going to discus two things. The first is how to prevent the deletion of data based on some business constraint, something that cannot be expressed through a constraint already existing within the database. And the second thing we're going to look at is how to log the deletion of data through some kind of custom table within our database, which can be really useful in the future if we need to audit the saved data over time.

Demo: Protecting Data with DELETE Triggers

And now, let's jump right into talking about how a DELETE trigger can help you prevent the removal of data that you want to have some control over, something that a normal database constraint can't help with. It's not a foreign key issue. It's not a minimum or a maximum or a range value. It's something business related. In this case, we want to make sure that if an order line on an order has the PickingCompleted filled in, that means the order has been fulfilled. We don't want that information to be removed. So we select from the DELETED table, that virtual table, and if any rows in that table have that date filled in, we want to roll the transaction back. Let's go ahead and create that trigger. Now let's try and delete that one row. And we'll see that our trigger did fire, and it prevented the deletion of data. So that's a great use case for DELETE triggers, and I see that often, and it's really useful when a database constraint that already exists can't satisfy the business need.

Demo: Creating an Audit Log with DELETE Triggers

Now for this demo, I would like to show you how we can use a DELETE trigger to log information about data that was just deleted. Now this really could be applied to any kind of trigger for any event, INSERT, UPDATE, or DELETE, but it's often in that delete state that we need to keep an audit log of things that have happened. Now this table is a simple example of the kind of audit table you could create. There are many examples you can find of the kinds of things you might want to log. In our case, we're going to log the time the action happened, who did it, what kind of operation it was, a delete in this case, the table, and the schema. But the last column, the LogData column, which is an nvarchar, we're going to use that in this example to store a copy of the row as a JSON document. That will allow us to search it, but it also gives us an easy framework for getting that information out. That might not be the most efficient path depending on your needs, but it's something you can use as an example to start to follow through. I'm going to create that table. And then looking at this trigger, you'll notice that the top is very similar to all the others. We first validate that we actually have data that's been deleted, it exists in that virtual table, and we can do something with it. Once we have it, we're going to select each row out of the DELETED virtual table and cross apply its ID back onto itself so that we can reselect the whole row out as a JSON document to store in that last column. There are many paths and ways you could do this kind of operation. Again, this is simply an example of how we could get all the pieces together so that we can log it. Let's create that trigger. Okay, now that we have that table and the triggers applied, let's verify that nothing yet exists in this logging table. And now I'm going to update invoices to remove OrderID for any orders below OrderID 10. And the reason I'm going this for this demo is that there's a foreign key constraint on invoices, so if I don't get rid of those OrderIDs out of the table, I can't delete the data. So I want to be able to show you that, so this is a simple way to make that happen. So nine rows were affected. And then we're simply going to delete those nine rows or at least attempt to. And we see that nine rows were in fact deleted. Now let's see what happened in that AuditLog table. And sure enough, we now have those nine logs of what occurred inside of this table. You'll see that it tells us it's a DELETE operation, and that last column has the JSON representation of all the columns within that table. Now what you might not know about SQL Server 2016 is that you can actually query on JSON data. So if I wanted to search this log table for DELETE operations on the Orders table, and I can even name an ID, OrderID 4, I can check to see if something exists in that table. And sure enough, it does because I deleted everything below ID number 10. And this is a simple example of how you could use something like a delete trigger to log information about that critical data, things that you might not be able to specifically prevent deletion on, but allows you to have an audit trail of when these kinds of major events take place.

Reacting to Modified Data in UPDATE Triggers

And now as we look at the update triggers, there's a couple of things I want to make sure you're clear on as we head into these examples. The first is remember that the update trigger is the only one that we get access to both the INSERTED and DELETED virtual table. And one really key point here is that all rows that had a modification attempted on them will be contained within both of those tables regardless of data actually changing. This is the one tricky thing with updates. All we know is that an event update was applied to rows of data. We don't actually know if data was modified in those rows. So if someone happens to run an UPDATE statement across an entire table, even if nothing changed, the INSERTED and DELETED virtual tables will contain every single row in that database. And so that really means that one of the key things I always instruct people to do within UPDATE triggers is to make sure you save yourself the work. If nothing was actually changed or if only some data was changed, find that data, and only do your work on that data. Now in this demo, we're simply going to enhance the logging trigger that we had for the delete. We're going to determine now if it was a delete or an update that changed the data and log it appropriately. And if it's an update trigger, we're going to make sure the data actually changed. If nothing changed, we don't care about logging the data.

Demo: Logging Changes with UPDATE Triggers

Now, as I begin this demo on the AFTER UPDATE trigger, I want to do something different from the previous examples. This time I want to demonstrate that both the INSERT and the DELETED tables are always passed into the UPDATE trigger. So when I actually do update a row of data or multiple rows of data, both tables will be filled with the same number of rows, it's just that the INSERTED table would have the new data, and the DELETED table would have the old data. However, even if I update a row but change nothing, that row still gets passed in. So let's look at that example here. I'm simply going to take a count of the INSERTED and the DELETED tables every time this trigger is executed and print that count to the screen. Let's create that trigger. And now let's update the sales order table, but we're going to pass in that OrderID = 0 again. So nothing is actually updated. You'll see the trigger is fired, but the count of rows is 0, so that's what we expected. However, even if I update only one row, and I just change the comments, we'll now see that both tables have one row. So the INSERTED has the new value, the comment that says Nothing to see here, and the DELETED table has the old value. Alright, now that we've seen that, let's talk about updating our previous example of the log. I'd like to show you how we could enhance that logging example from the DELETE trigger demonstration. We're going to create an UPDATE trigger, and in this case, we're going to decide whether or not the data that was modified actually contains changes. So if a value simply got reset, we won't log those changes. We do this with a modifier called EXCEPT. And in this example, we're simply saying the data that was passed in from the DELETED table, only select out the values that are different from the INSERTED table. Remember, we have a copy of both rows in their different states within those tables. So in this case, we'll only get data returned that actually had changes. And that's what we want. Once we have that data, just like the delete example, we'll log that information to the table, selecting it out as a JSON document. Let's go ahead and create that trigger. And now, let's go ahead and reset a value inside of the sales order table. In this case, I'm simply going to change the ContactPersonID. Five rows were updated. Now if you remember in my previous example, we removed everything less than 10. That's why we have five rows that are updated here. Now let's select from the log table. And we see that we have new additional rows, and the operation is UPDATE. And we've logged the value previous to what it was before we changed it. In this case, you'll see that the ContactPersonID of each of those rows is different. It is not 45 because that's the state it was in before we did the update. Now let's see what happens if we simply reset that value, we run the exact same statement, so we are actually modifying the row, but let's see if we log anything. So we see that five rows were updated, but let's select out of that log. In this case, we haven't entered any more log information. From a data perspective, nothing changed. We don't necessarily need to know that the action occurred. We just want to keep that audit when something is modified. So hopefully this gives you another idea about logging and how you can inspect those INSERTED and DELETED tables to only act upon data that truly changed.

Modifying Trigger Execution Order

And finally, let's look at the ordering of AFTER triggers. Remember that an AFTER trigger is not ordered by default. There's no guaranteed order. We're allowed to set one as the first execute and one as the last execute. And one of the biggest reasons we would need to specify a first or a last is because one of those relies on data already having been modified by previous triggers. In this demo, we're going to use the sp_settriggerorder stored procedure to specify a first and last trigger. And to demonstrate how the order impacts the data that you have access to, we're going to enhance that logging example and see that if we don't have triggers in the right order and one of them manipulates data, we don't get the right stuff logged, and so we're going to look at how that can be used.

Demo: Setting Trigger Order

In this last demo about trigger creation, we need to discuss the ordering of triggers. Now as we said, I can have one trigger per event per table that runs first and one per event per table that runs last. To demonstrate that, I'm going to create two triggers, both AFTER UPDATE triggers. One is called CreatedFirst, and the other one is called

CreatedSecond. Now they are being created in that order. And then to try and execute these triggers, we're simply going to set a value equal to itself on the orders table. So even though no data is changing, we should see those triggers execute. They do execute, and you'll notice that they are executed in the order that I created them. Now this is actually just a side effect of how SQL Server keeps this data stored. There is no guaranteed order, something we've said a couple of times now. And to demonstrate that, we're going to look at what would happen if a trigger got recreated. So I'm going to drop that CreatedFirst trigger, and I'm simply going to recreate it. This is the exact same code. Nothing has changed. And then I'm going to run the exact same UPDATE statement. Nothing has changed here except that I dropped a trigger and recreated it. And now you notice that the CreatedSecond trigger is actually executed first. Now again, this is really a side effect of that creation order. That's not something you can rely on. Instead, we need to use some built-in functionality within SQL Server. To order triggers, we use the stored procedure called sp_settriggerorder. With that stored procedure, we have to name the trigger, the order we want it to fire, and the event type it is for. Again, you could have one first and one second. So we're going to reorder these, CreatedFirst happening first, CreatedSecond happening last. And now we're going to run the exact same UPDATE statement and see that the CreatedFirst is now executing first again. So it really is that simple to order your triggers within the events on a table. The one thing to remind you of here is any time you alter or modify any of these triggers you must call the settriggerorder again. It will have its order value set to none, and now there is no defined order for that trigger until you call the stored procedure again.

Demo: Ordering Triggers for Correct Results

I want to set the stage for how this last example of trigger order can really impact actual query results. Let's say for instance you have a sales table, and you have salespeople. Let's pretend that a new requirement comes from the sales department. Any time a salesperson leaves the company or is decommissioned, their sales need to be transferred to another salesperson. Sometime later, a new requirement comes from sales. Not only do they want all of the orders to be reassigned, but they want to have a log, a count, of how many orders salesperson 1 had before they left the company. And you'll see that if we don't have these triggers in the correct order, and depending on which kind you create, the data may no longer be available to query. So let's see how we go about getting the results we expect by using trigger order settings. First, let's select how many people we have as salespeople within our organization. We see there are 10 salespeople. And for the sake of this demo, every time someone is reassigned from being a salesperson, we're going to assign all of their sales to PersonID 20, Jack Potter. This first trigger satisfies that need of changing those orders. The way that we do it is with the EXCEPT query we've seen earlier. We select the PersonID and the flag IsSalesperson from each table and only return the ones that are different between deleted and inserted. So when they used to be a salesperson, they're old data and they are no longer a salesperson, and the new data, we get that record back, and we can use that. We then take that temporary data in the temp table, and we reassign all of those sales orders in the sales order table to PersonID 20. Let's create that trigger. And as we said, at some point later, we get the second requirement. Now in this case, we're going to do the same EXCEPT query. I would probably handle this differently. Maybe these should be in the same trigger, or maybe they have to be separated because not every database has the same requirement. So we're going to do that same EXCEPT query, get the same list, and then we use the same auditing query we've seen earlier. We're going to iterate over each row in that modified data temporary table, and we're going to select using the PersonID in that table a count of all the orders in the sales order table. And we're simply going to add that count as a JSON file into that logging table we've seen earlier. Let's go ahead and create that trigger. And now, let's check our AuditLog. There's nothing in there, I've deleted things from previous demos, and we're simply going to do this reassignment. We're going to say PersonID 3, who is currently salesperson, let's set them equal to not being a salesperson. They've left the company. One row is affected, and now let's see if our first trigger actually did its job. And it looks like it did. We reassigned those sales, and the second trigger actually logged the data for us. There's one problem, however. You'll notice that OrdersBeforeReassign, that count, says 0. But I know this person, and I know that they've been here a long time, and they've actually had thousands of orders in the pipeline. What went wrong? It's the trigger order. Remember, there is no guaranteed order unless we set this. What happened is the first trigger happened to fire first, and it reassigned all of those sales in the table. It's an AFTER trigger. And then the second trigger came along, got the same ID, and tried to join to the sales table, but all the orders were gone. Let's see what happens when we reorder those two triggers. I'm going to use the stored procedure, sp_settriggerorder, name my two triggers, give them an order, and then see what happens. Alright, so now let's go ahead and run another one. Now I've reassigned the first person, so I can't use that ID again. This time we're going to use PersonID 6. Let's reassign all of their sales. And that happened, and now let's see what we get. And there you go. By reordering the triggers and making sure that the data was available when we needed it, we can now see that before that person was reassigned they had over 7, 000 orders. Hopefully this gives you an idea of why that order is necessary. And I'll remind you one more time. Because this order is critical to the business logic that's required, any time either of those triggers are modified or deleted and recreated, the trigger order must be set again, or you could end up with the first result and orders not being counted correctly.

Review

Now we've covered a lot of ground in this module, and triggers can be used in many ways to help us keep the business value within our database consistent and applied across multiple applications. We've looked at what DML triggers are, how you can use AFTER and INSTEAD OF triggers for your various use cases and why you might choose one over the other. We've demonstrated and talked about the INSERTED and DELETED virtual tables and the real value of having access to all those changes. We looked at a couple common use cases and demonstrated how you might use triggers to accomplish those. Specifically, we looked at how to log data, which is far and away one of the best reasons to use triggers even in light of some new developments within SQL Server. And finally, we looked at DML trigger order and why you might need to specify a first and a last trigger based on your business needs.

Protecting the Database with DDL and Logon Triggers

Overview

Welcome back to Programming SQL Server Database Triggers and Functions. In this second module, we're going to talk about protecting the database with DDL and logon triggers. In this module, we're going to discuss DDL triggers and how they differ from the DML triggers we talked about in the first module. We'll also look at login triggers and how you can use them to manage security events, and we'll end by looking at a couple common use cases.

What Is a DDL Trigger?

So what it is a DDL trigger? Well, as you would expect, the acronym stands for something very similar to DML. It stands for Data Definition Language, and these are the statements within T-SQL that actually modify and manipulate schema and the database, things like CREATE and ADD and DROP and ALTER. Now these are really useful as a tool, specifically for database administrators. DDL triggers watch for those schema modification events at the database or the server level, something you can apply to all databases within your server. Much like the DML events, DDL triggers react to those DDL modification events. Now they're not attached to a specific table or schema. This is a database or server-level across multiple database kind of events. Now what this means is that if you create a DDL trigger to prevent something from happening, you're creating more work if that modification action needs to happen. This is usually a really good thing, and so it's something to take note of and use wisely. And honestly, one of the best ways to talk about DDL triggers is that they provide an easy and accessible way to prevent mistakes within your database and to log changes to your schema if you need to see what changed over time.

Understanding the Anatomy of DDL Triggers

So much like a DML trigger, let's look at the anatomy of a DDL trigger. Now the way that you create them looks very similar. We have the same CREATE and ALTER we talked about previously. The difference here is that you specify whether it's for the database or for the entire server. And then what you're creating is not an INSERT, UPDATE, or DELETE. You're specifying the DDL events or apparent event group of the kinds of events you care about. Do you care about DROP and ALTER table? Well, then you name those event types. DDL triggers execute after the event has completed, and that's really something to understand specifically say for an ALTER or a DROP event. If you prevent a table from being dropped and the drop then occurs and the DDL trigger says, no, roll that back, there's a lot of logging that happens there within the database, so it's just something to be aware of. Probably in the end you're happier that the table wasn't dropped. It's just something to recognize within the transaction scope and the length. It's also worth noting that DDL triggers can be defined for events at the database or the server level. So maybe you have a policy that says tables can't be dropped in any database in this server. Well, that's something you can apply at the server level, but maybe within a specific database you want to log a specific kind of event that doesn't make sense across all databases. This is what it would look like to create the same trigger at the database level and at the server level. The only thing that changes here is where you're creating it, the ON DATABASE or ON ALL SERVER. Similar to the DML inserted and deleted virtual tables, DDL triggers give you information about the event that occurred through a special function called EVENTDATA. EVENTDATA provides detailed information about the event that occurred, and it's really useful for logging DDL events. It is an XML document that's returned from that function. One of the great pieces of information that's returned with the EVENTDATA function is the actual command text that triggered the event.

Examining DDL Trigger Events and Event Groups

So what are these events I keep talking about? Well, the reality is within SQL Server 2016 and above, there's at least 275 DDL trigger events. What are those events? What is that? Well, at high level, it's things like creating and altering tables, indexes, functions, procedures. Even the triggers we're talking about in this module, you can create DDL event triggers on the creation, alter, or deletion of those triggers. Now there are many more than what I'm showing on the screen, but hopefully you get a flavor for what these kinds of events look like. At a second level, SQL Server allows us to specify groups of events. So rather than having to specify CREATE_TABLE, ALTER_TABLE, and DROP_TABLE on a trigger, we can simply say DDL_TABLE_EVENTS, and that includes these three events. So something like DDL_INDEX_EVENTS includes more than just the CREATE, ALTER, and DROP. It also includes full text events. And what's more, there are some event groups that are simply parents to other event groups like Service Broker. There are many kinds of event groups for Service Broker. We have queues, and routes, and message types. And if you want to log or do something with all the events that could happen within Service Broker, you can specify DDL_SSB_EVENTS. And you'll see other examples of things we would expect, and it's just an easy way to name multiple events with one group.

Common Use Cases for DDL Triggers

Now the common use cases for DDL triggers in most cases boils down to two things. One is preventing the modification of a schema within your database. When you give access to more than a small group of people, you want to ensure that the database doesn't change unexpectedly. And likewise, it's a great use for auditing events that occur at the schema level within your database, knowing when triggers were modified, knowing when columns were added or dropped, knowing when someone attempted to drop a table or create a new table. It's a really great way to keep a log of what's happening within your system.

Using DDL Triggers to Audit Events and Prevent Changes

And so, as we go to the demo, we're going to look at those two things specifically. First, we're going to prevent the altering or dropping of tables, and then we're going to look at how you can use DDL triggers to log events within your database. Let's get started.

Demo: Preventing Schema Changes with DDL Triggers

In this first example, we're going to create a simple DDL database trigger that will prevent the dropping or altering of any tables within this database. If someone attempts to drop or alter a table, we'll respond with a message telling them that the trigger must be disabled in order to complete that action. Let's create that trigger. Now again, remember, this is a database- only trigger right now, so if I were to switch to another database, this DDL trigger wouldn't take effect. Now the first thing we'll attempt to do is drop that AuditLog table. Now this is a great example of why you might want to use DDL triggers to prevent actions like this. If you were to lose that AuditLog table, depending on your backup situation, you could lose valuable information. Let's go ahead and try and drop that table, and you'll see that our trigger did execute, and it's preventing us from dropping that table. It says that we must disable this trigger if we want to complete that action. Well, what about altering a table? Let's try and alter the person table. And we have a great idea to let their TwitterHandle be put into that table. It's a simple thing, there's nothing nefarious about it, but again, because this trigger is set up on the DROP and ALTER events of tables, we aren't allowed to do that. Well, let's say we really do need that action to occur; there's a new business requirement. And so this is where we can disable the trigger. It's a simple statement, DISABLE_TRIGGER, and the name of the trigger. And in this case, we have to specify where that trigger exists. It is on the database that we're in. This is not a server-wide trigger. We can disable that trigger and then see what happens if we try that ALTER statement again. And now that statement completes. Anytime we disable a trigger, we want to make sure that we're very good about reenabling the trigger once our work is done.

Demo: Logging Schema Changes with DDL Triggers

Similar to our examples of logging data with DML triggers, we're going to create a separate AuditLog table for DDL events and actions. For this example, we're going to log the time the event happened, what type of event it was, the person that executed it, and then the actual T-SQL command which caused this DDL trigger to execute. Let's create that table. To do this, we're going to create one master DDL trigger within this database. We're using the

DDL_DATABASE_LEVEL_EVENTS event group, which encompasses almost all the DDL events within the database. Now in your specific case, it would probably be best to target exactly the kinds of things you want to audit and log. You'll notice that what happens is we use that special EVENTDATA function that we talked about in the slides. That EVENTDATA function will return an XML document, and we can then use the XML querying capabilities of T-SQL to get specific pieces of information out of that XML document, and we'll use that to insert into that table we just created. We create that trigger, and then we perform some actions. First, I'm going to attempt to alter the people table again, and now we want to add InstagramHandle as another feature of that table. Ah, but wait a second. I forgot that we still have that other DDL trigger that's preventing me from actually making that alter. Well, I wonder, did this log anything within that table simply because I attempted it? It turns out it didn't. That trigger rolled back the transaction before we got to the point of actually altering the table, and so nothing was logged. So to do our work, we need to disable that trigger and attempt that action again. This time it completes successfully, and we'll see that our DDL logging trigger did enter an entry into this AuditLog table, including the entire T-SQL command that caused this trigger to execute. And remember, any time we disable a trigger to do work, particularly DDL triggers, we want to make sure that we're very faithful to re-enable them so that the work they were intended to do can continue to carry on.

What Is a LOGON Trigger?

And now let's talk about logon triggers. In SQL Server, a logon trigger is one that responds to the LOGON event, but it only executes after a user has successfully authenticated with the server. Now before we go any further, I must ask you to stop and think about what you're doing. Logon triggers are powerful, and they can render your server inaccessible and unusable if they are not implemented correctly. You must proceed with caution. And this is where I will point you just briefly to the side of SQL Server administration. I implore you to make sure even in doing these demos that you have the dedicated administrator connection enabled. This is your one get out of jail free card when you're working with SQL Server. If you do something, even with a logon trigger, that prevents you from normally logging in, you can connect to your SQL Server through one thread and do some work on the server to get yourself out of a jam. In fact, even as I was preparing these demos for this course, I almost locked myself out of my local server if it wasn't for the DAC. Please do yourself a favor. Read up on what the DAC is, and make sure it's enabled on all of your machines. And now as we proceed, I simply want to say there be dragons with logon triggers for those who don't set up the DAC and prepare appropriately.

Understanding the Anatomy of LOGON Triggers

So let's talk about the anatomy of a logon trigger. As you can see, the syntax for creating a logon trigger is very similar to all the other trigger examples that we've seen. The two things that are different is we have one option of what this trigger is for. It is FOR LOGON. So we know that because it is FOR, that means it's an AFTER trigger. And I've put here on this slide the text for EXECUTE AS and UserName. We're going to look at this a little bit in the demo, but we will dive in a little bit more into permissions during module 4. Now a couple things about logon triggers as we prepare to show them. Logon triggers are always created in the master database; however, they don't actually show up inside of the database within SQL Server Management Studio. You can find any server-wide server triggers from the master database here in the Server Objects folder. You can also find them by doing a select * from sys.server_triggers. You are allowed to have multiple logon triggers for the server. This means that when a user logs in and they successfully authenticate to the database, you can decide which logon trigger runs first, which one runs last, and if any others run in the middle in a non-specified order. All output from a logon trigger is diverted to the SQL Server error log. That means that when the logon trigger is executed, a user simply sees an error and something about a trigger that caused the error, but they don't actually see the message that you printed out. Instead, that message would be put into the SQL Server error log specifically so that administrators and others can see them there. And finally, logon triggers provide information about the event with the EVENTDATA function that we saw earlier from the DDL triggers. Now the information that is provided with the EVENTDATA function is slightly different from a DDL trigger, but it provides a great way to get some detailed information about the LOGON event, and it is specifically useful for logging LOGON events. And I want to remind you that you want to make your logon triggers as fast as possible because every connection from every application, whether SSMS or a real application being served from your database, will use them. If something takes a long time or you're trying to do too much work, you will be unintentionally slowing down every application.

Common Use Cases for LOGON Triggers

Much like DDL triggers, the common use cases for a logon trigger really boil down to two things. The first is to prevent a user from actually connecting to the database even if they have successfully authenticated. Now a really good example of this is not allowing your server to be overused by people within your organization. Maybe you allow people to connect with SSMS for instance, but only once or twice per user so the connections don't get used up. You don't prevent them from logging on, you just prevent them from logging on and using too many connections. And the second reason most people would create a LOGON event is simply to do some kind of custom logging when people authenticate against a server. Now remember, these are only fired once someone has successfully authenticated. So this is not useful for denied authentication, but it is a way to provide some context and maybe put something into a central logging database that is useful for other applications to see. And I want to remind you one more time, before we proceed with these demos, please stop and make sure that you have enabled the dedicated admin connection on your server before you accidentally lock yourself out and prevent work from happening any further.

Demo: Preventing Connections with LOGON Triggers

In this first demo, we're going to prevent users from connecting to the database multiple times using a logon trigger. For this first demo, we're going to create a local user within this SQL Server environment. Now, I am not going to practice any good standard security measures when creating this user. I simply want to show you how a logon trigger can be used within your environment for interacting with a LOGON event. The other thing I want to point out is that I've switched to the master database. This is where logon triggers need to exist, and so we're going to start there through this demo. The first thing I'm going to do is create that test user. Again, it is a local user, and it has a very insecure password. The second thing I'm going to do is grant this user the VIEW SERVER STATE permission. This simply allows any user to select from dynamic management views within SQL Server. As you'll see in our trigger example, this is necessary to make the trigger work. Now as you'll see, the syntax for this trigger is very similar to all the others. It simply has the FOR LOGON as the event and it does use a couple of functions that we haven't seen elsewhere. The one that is most notable is the sys.dm .exec_sessions view. This allows us to see what connections are currently existing on our server, and in this case, make a count for how many this user specifically has. Now obviously, this trigger is for demonstration. If you were to do something like this on your server, it would need to be more generalized, and you'd want to think very cautiously and specifically about how you want to count the number of connections users have and who you want to limit based on what criteria. In this case, we're going to say if this login_test user has two connections open, they will not be allowed to open any more connections. Let's create that trigger, and let's try this out. I'm simply going to create a new query window, which will initiate a connection based on the user I currently have. So I'm going to switch that user to my new local user with my very secure password, and we get connected. Now again, I'm going to open a second query window, which you'll notice does show the user login_test. And now let's say this user really loved to use SQL Server Management Studio and wanted to have another window open. As we do that, you'll see that this user gets an error. All they know is that the login failed because of a trigger execution. And remember I told you that these events happen within the SQL Server error log. So let's see what that looks like in the error log. In this case, you'll see that there are two examples of the user being denied access. Now I didn't specifically print any messages, so all we know is the LOGON event did fail because of a trigger execution. One thing I'll note is that if you don't add output to your trigger, the message itself is not very helpful about which trigger actually caused the problem.

Demo: Logging Authentications with LOGON Triggers

In this demo, we're going to create a second logon trigger to capture information with the EVENTDATA function and log it to an audit table. Much like the examples we used for DDL triggers, I'm going to create a simple database to store this information. Within that database, I'm going to create a simple logging table. This looks very similar to the DDL audit table we had in the previous demos. This time we're going to store the process ID that was part of the LOGON event and the host for the user that was trying to connect. And now, let's see how we would use a logon trigger to insert information into that auditing database. The one thing that is unique about this trigger, as I mentioned in the text above, is that I am adding the WITH EXECUTE AS parameter. We're going to talk about permissions more in module 4, but this is necessary because that database that I created does not allow our new local user to insert data. There are numerous ways that we could go about providing that access, but one way is to tell the trigger to execute as somebody that does have access to insert into that database. That's what I've chosen here, and we'll talk more about permissions later and which one might be the right choice. As far as what we do within the trigger, this looks very similar to the example within the DDL trigger demos. Within that table, we call the LOGON event, we select information out of the XML that comes back, and then we insert that information into that table. Let's create that trigger. And now, let's see what's in that table. We expect it to be empty. And now let's create a new query window. Now this is as my local admin user, and let's see what happened inside of that table again. Now you'll notice that lots of activity happens with a logon trigger, and you'll see much of this, and you might want to decide how much of this you log. And you'll also notice that if I take that window we created and I change to my new local user, because I correctly put on the EXECUTE AS example, the user's logged in, and we get results showing that they were successfully logged into this database server. The last thing I want to remind you is that you can reorder logon triggers if necessary. This looks very similar to how we talked about it in the previous demos. The only thing unique is the namespace parameter. This must be provided with logon triggers, and it must be set to SERVER. If you don't, you'll get an error message, and you will not be able to set the order of these triggers.

Review

In this module, we have looked at what DDL and logon triggers are within SQL Server. We've looked at the EVENTDATA function and how it can provide additional information of both the user and the event that caused this trigger to execute. We've looked at common use cases for both DDL and logon triggers, both in preventing bad things from happening and logging an audit trail of events that we care about. And finally, I reminded you that you must use extreme caution when utilizing logon triggers, having that dedicated admin connection ready and available should something go wrong.

Working Smarter with Triggers

Overview

Welcome back to Programming SQL Server Database Triggers and Functions. My name is Ryan Booz, and in this module, we're going to talk about working smarter with triggers. In this module, we're going to discuss some aspects of security and execution context with triggers. We'll look at the MERGE statement and some challenges that it presents when dealing with multiple triggers. We'll look at how triggers can often hide or make work transparent and difficult to debug. Next, we'll discuss what it means to overuse triggers and how you need to be cautious about the work that you're doing. And finally, we'll briefly discuss Service Broker and how you might be able to do asynchronous work with triggers and Service Broker. Let's get started.

Trigger Security

So let's first discuss trigger security. There are two things that we need to be aware of specifically when dealing with triggers inside of SQL Server. The first is an anti-pattern with logon triggers that you might find in old legacy systems, something I want you to be aware of. And the second is the execution context that the trigger is executing under when it runs. Now when I talk about an anti-pattern with logon triggers, I'm almost always talking about something you'll find in legacy systems. These systems would validate against data that's easily spoofed. Specifically, that meant usually hostname and application name. The intent was to prevent someone from logging in if they weren't coming from a specific host or using a specific application, maybe like SSMS or the web application itself. Unfortunately, both of these options are easily modified through most applications and directly on the connection string. As an example of this, I want to show you a logon trigger that tries to prevent access based on the application name. Now this exact same kind of logic could be used for hostname as well. As you can see, this trigger tries to prevent someone from logging in if they are not coming from TrustedApp1 or TrustedApp2. Unfortunately, simply appending AppName=TrustedApp1 to the end of this connection string makes SQL Server believe that I'm using that trusted application, and therefore, this security is easily bypassed. If your system is using logon triggers, I encourage you to take a look at them and ensure that you're not trying to prevent logon through these kinds of means in something that's easily bypassed. The second thing to discuss is execution context. Now execution context actually exists for triggers and functions and stored procedures within SQL Server. By default, triggers are executed with the permissions of the CALLER. This means that it's possible for someone to create a trigger that would ultimately give them more access than they have. Once somebody with a higher-level admin access runs that trigger, they would then be granted access to more of the system. One way to prevent this or at least be more intentional about it is to be specific with your execution contexts. We can modify the execution contexts of our triggers by adding the WITH EXECUTE AS clause to any trigger that we create. When that is added, anytime this specific trigger fires, regardless of it being DML or DDL, it will execute as the user specified. Let's look at what kind of execution contexts you can actually add to your triggers. Now by default, as we said, if nothing is specified, the trigger will run as the CALLER. That means whoever actually ran the DML or DDL statement, that's the person that it will run as. This also means that that user must have permission and access to everything that the trigger tries to do. You can also specify SELF. SELF means the person who created or modified the trigger last. Now I would caution you to use the user_name variant of the execute context that we're going to look at in just a minute. One of the reasons SELF is not preferred is because it's unclear. When you look at schema that's generated through a system and you see EXECUTE AS SELF, it's hard to know who that SELF was, so try and be more clear when you specify context. The third potential is OWNER. Now this is different from SELF in that it's the actual owner of the trigger, and so someone else who last modified it might not own it, but they have permission to modify it. And one caveat to be aware of is that OWNER, as an execution context, is only permissible on DML triggers in most functions. And then the last way you can specify context is with the user_name or login_name specified. This is usually the clearest option when you want to choose something other than the CALLER context. If for no other reason that when you generate SQL from tools that show this trigger, you'll know exactly who it's executing as. As we finish taking about security and triggers, I simply want to remind you to be very thoughtful and explicit with your permissions. As your system grows, you want to make sure that the right users can do the right things. In general, always use the principle of least privilege when you are dealing with triggers, functions, or stored procedures in thinking about who they're executing as. And finally, as you think about triggers and security, recognize that triggers are generally transparent to most users and developers and sometimes even DBAs. Make a habit of checking the sys.triggers and sys.system_triggers views. This will allow you to be aware of what's happening within your database and within your system. Track changes to what happens within those views, and ask questions if necessary.

The Problem with MERGE and Triggers

The next thing I'd like to look at is the MERGE statement, how it interacts with triggers, and the problems that it can produce sometimes if you're not aware. So what is the MERGE statement? It was introduced in SQL Server 2008. It provides a way through one statement to do an INSERT, an UPDATE, and a DELETE. In other technologies, this is often called an UPSERT. It's something that a lot of people had asked for for many years, and the implementation can be useful under the right circumstances. This is what a MERGE statement looks like. You provide a source_table and a target_table. The source_table has the data that you want to use for the UPDATE and the INSERT and as a source to determine what needs to be deleted. When you join those two tables together, SQL Server then tries to match rows, and when it finds it, it updates the columns that you specify. Any rows it doesn't find in the target_table it will then insert based on the statement you provide. And finally, you can have it delete rows in your target_table if they're not found in your source_table. The real problem comes in how it interacts with the triggers itself. Remember I've said a few times through these modules that we specifically put this one IF NOT EXISTS statement in all of our triggers. The reason we do this is that a MERGE statement will actually execute each of those DML actions regardless of whether something changed within that specific DML context. I'd like to show you that in a demo.

Demo: MERGE and Triggers

In this demo, we're going to see the impact that a MERGE statement can have when multiple DML triggers exist for a given table. Now remember, the reason you would choose a MERGE statement is to attempt to do the INSERT, the UPDATE, and the DELETE all in one SQL statement. To demonstrate this, I'm going to use the Application.People table, and I'm going to create a mismatch between that table and a temporary table I'll create from it so that we can see each of these statements take place. Here's how I'm going to do it. First, I'm going to insert a new user into that Application.People table called Ryan Booz, and I'm just using the minimum amount of information necessary to create a new person. Second, I'm going to create a temporary table with those same columns, just the minimum amount necessary to create a new person. Next, I'll insert all of the users that currently exist in the Application.Person table into that temporary table. That's a little over 1, 000 rows. This now gives me the target_table, the real table in the database, and the source_table, this temporary table I've just created. And so now we want to create some kind of mismatch so that we can see how the UPDATE, the INSERT, and the DELETE take place. To create this mismatch, I'm going to do two things. First, I'll create a second user in the temporary table only. So this table, Fake Ryan, does not exist in the real Application.People table. Then, inside of this same temporary table, I'll delete that first user I created. So now we have two tables, and they both have a mismatch. They both have a user that doesn't exist in the other. And when we use one of them as a source, we'll be able to use that MERGE statement to update, insert, and delete each of those values. So now let's go ahead and create our trigger. You'll notice two things about this trigger. First, it is an INSERT, UPDATE, and DELETE trigger. The second is that it doesn't do any of the normal checks that I have cautioned you to do throughout these modules. Instead, we're going to save that row count as a variable that we can use to print to the screen so we can see how many rows the trigger was told it was going to be modifying. And then when I normally would check the inserted and deleted virtual tables to decide if I should do any work, instead, we're simply going to check each of them and try and figure out which trigger was executed with each iteration. So when we do a MERGE statement, all three of those statements are triggered, and therefore, this trigger is executed three different times. Let's go ahead and create that trigger. And now I'd like to actually do the MERGE statement itself. Now if you've followed me up to this point, hopefully we've come to the same conclusions. I expect a little bit more than 1, 000 users to be updated. I expect one new user, Fake Ryan, to be inserted into the person table, and I expect one user, Ryan, to be deleted from the person table because they no longer exist in that temporary table. And so you'll see that I'm using that temporary table as the source, the

Application.People table as the target. I'm joining them together on the PersonId and then determining whether I do an UPDATE, an INSERT, or a DELETE based on what matches between those two tables. Let's go ahead and see if we get the results we expect. Now after a couple of seconds, you'll see that I did actually update a little bit more than 1, 000 rows, but this is the big surprise. Every time this trigger was executed whether with the INSERT, UPDATE, or DELETE, the trigger was told that there were 1, 000 rows that were going to be updated. I expected the INSERT and the DELETE trigger to only show one row, and this is exactly why we need to put those checks into our triggers. Among other potential issues with the MERGE statement, this is one that can really cause some performance problems when dealing with triggers. Now let's see if we actually did get the results we expected. The Ryan user no longer exists, but the Fake Ryan user does now exist in the real table. Now let's just quickly see what happens if we simply run this all again without doing anything with additional users. I'm going to truncate the temporary table just to erase it, simply insert all the new users again, and then I'm going to run that exact same statement. Now this time I expect those same 1, 000 plus users to be updated, no users to be inserted, and none to be deleted. And sure enough, what we can see is that only the UPDATE trigger actually did work. That is the only trigger that had inserted and deleted virtual data, but both the INSERT and the DELETE trigger were executed, and they were told that there's the potential of over 1, 000 records to be updated. So again, you can see it's really important to keep those checks at the beginning of each of your triggers.

Bypassing Transactions in Triggers

The next thing I want to discuss is the potential of bypassing the transaction within the trigger and one specific use case where this could actually be useful for you. Now let's just revisit the basics of transactions within the trigger. Remember that the trigger is subject to the calling transaction. So if there's a rollback at any point within the DML transaction, the work within the trigger is actually rolled back as well. One potential side effect of this rollback is that any audit logging you might have been trying to do in the trigger is also rolled back as part of that transaction. And sometimes this can almost feel like it defeats the purpose of maybe a specific kind of audit log you're trying to maintain. Now this is kind of an edge case, but there might be reasons why you want to know that an action was attempted even if a constraint ultimately rolled back that transaction. I'm not specifically saying this is a great use case. I just want you to know that there are some ways around that transaction within the trigger if you have a need that arises. So how do we do this? Well, it turns out that this is where table variables can actually come to the rescue if you have a need like this. Table variables provide a means for bypassing the trigger transaction scope. The reason they can do that is because table variables are scoped to the module that they are declared in. They do not exist in the overall transaction that's happening around them. So we can use this to our advantage in the special use cases where we might want to get information out of the trigger even when a rollback occurs. Now I do want to deal with a couple quick things when it comes to table variables. Depending on where you've come from within SQL Server, you may still have a couple misconceptions. The first is that table variables are not inherently better or worse than temp tables. They are not memory-only tables, and they can actually be spooled to disk just like regular temp tables. They do now allow creation of indexes if the need arises in SQL Server 2014 or greater. And as we said, they are not subject to the calling scope transaction. Let's go ahead and look at a brief example of how we could use this to log information even in the event of a rollback.

Demo: Logging Information Outside of the Transaction

In this demonstration, we're going to see how we can use table variables inside of our trigger to still save information to a logging table even when rollback occurs. To demonstrate how we can get information out of a trigger even in the event of a rollback, I'm going to use the orders table and create a DELETE trigger. You'll notice I do the exact same checks I normally do, but I'm going to do something different at the beginning. I'm going to define a temporary table variable called deleted, which has all of the columns of the order table with the same data types. Into that table, I'm going to select everything from the deleted virtual table. So I have a record of everything that's in the deleted virtual table inside of a table variable within this trigger. The rest of this trigger begins to look like what we had previously. We're doing a check that says if the PickingCompletedWhen date is not null, we're going to roll back the action so that these rows can't be deleted. However, because I have this table variable and it is scoped inside of this trigger, I can then take the information, even though we've rolled back the outer transaction, and do something with it. In this case, I'm going to do a couple of SELECTs to determine whether or not I want to log all of the data or just a subset of data. So if the table attempted to delete, but our trigger rolled the information back, I'm not going to actually log all of the row data in that JSON document that we saw earlier. I'm just going to set that equal to null. However, if the user actually was able to delete these rows, it passed our trigger checks, then I'll log the entire row of data as that JSON document. Let's go ahead and create this trigger. And let's see what happens. So first, I'm going to go ahead and delete all orders below an OrderID of 20. Wait a second. I didn't even get to my checks. Oh, that's right. This is an AFTER trigger. Because a foreign key constraint did not pass, it didn't even execute anything inside of here, so we didn't log anything. Okay, well there's nothing wrong with that. We can fake this data by deleting some of those OrderLines for now. Alright, so now let's attempt that delete one more time, and let's see if our trigger actually does the rollback we expect. Now we see that our trigger took over. So it passed those foreign key constraints, our trigger was executed, we rolled back the information with our message, The OrderLine has been fulfilled and cannot be deleted. Let's see, however, if we were able to log that the delete was attempted. And sure enough, we see that we get 19 rows of data that were attempted to be deleted. And so this is a way that you could use a table variable inside of a trigger to get some additional information out if that was necessary for your specific application. I did want to very quickly show you how this does not work with a temporary table. As we discussed, only table variables are scoped inside of the trigger itself. They are not part of the containing transaction. So I'm simply going to redo this trigger, and I'm only changing it by selecting everything into a temporary table, not a table variable. Because the temporary table is scoped to the larger transaction, we should see that our information does not actually get logged as we saw previously. So I'm going to truncate that audit table log just so we can see from the beginning. And now I'll attempt that same delete, and you'll see that our trigger did execute, and it did roll back that delete. And now let's see if it logged that attempted delete using a temporary table. And it didn't. So that is why it's really useful to know that those temporary tables are scoped within the module that they are actually defined in.

Triggers in Moderation

As we near the end of this last module on triggers, I want to talk about two performance-related issues. The first is thinking about triggers and the effect they have on the work that's going on around them. There's a saying that says when all you have is a hammer, everything looks like a nail. And as you become more comfortable with triggers, trust me, you'll begin to see a trigger as that hammer and almost every requirement around you as a nail. As a DBA, you'll start to receive requirements, and you'll begin to think about how you could accomplish that in the database layer, things like constraints, maintaining data across tables. You'll immediately begin to think how you could do it by default in the database. And often, a trigger will come up as one of those solutions. Consider the audit logs that we've discussed in these modules. Even though triggers are a great way to maintain those audit logs, how often do you really review those logs? Do you have an archive strategy? If something really were to go wrong, would you actually even remember to go look at that logging table that you created? What about foreign key triggers? Do your databases do this extra work? Now many legacy systems, as I mentioned earlier, used triggers for foreign key integrity; however, SQL Server now does that on its own. It's done it for many, many years, and the query optimizer actually uses those foreign keys to do work. If you've robbed the optimizer of that knowledge, you're actually hurting yourself and doing more work than necessary. You also want to consider the effect of rollbacks that happen within your triggers. If you use triggers for all of these many ways to prevent data from being written or deleted, every time that happens in an AFTER trigger, remember the event has already happened, and then the trigger rolls it back, and that causes your logging to grow. And so if you have a very heavy system, you could really be impacting your performance because of a lot of rollbacks caused by triggers. And finally, triggers are generally transparent and hidden to developers. In most context, developers get used to looking at tables and views, but it's often difficult for them to remember that there are triggers that can modify the data across many levels within a database, and a solution to a problem that they're trying to develop might continually run up against this trigger that's remodifying this data or causing a performance problem that's very hard to debug. I want to show you as an example of this a couple ways that the work that's being done inside of triggers can impact the performance of your system.

Demo: Mitigating Extra Work in Triggers

In this next demo, we're going to look at ways you can identify the work that is being done within your trigger and some steps you might try and take to mitigate it. Now as I said, often triggers become that hammer for all the problems that you see as the nails, and so you need to examine the work that you're doing within your trigger to make sure that you're not keeping those transactions open longer than necessary when data is modified within your database. To give you a brief example of this, I wanted to show you what happens when you do something as simple as logging information about a DML action, something we've seen in the previous modules. If you aren't thinking about the potential of thousands and thousands of rows being modified, whether it's through an INSERT, an UPDATE, or DELETE, you may not realize that that logging action will actually increase the time of your transaction the more rows that are being modified. Let's look at how this impacts performance. I'm going to create an update trigger on Sales.OrderLines. All it will do at the end of this trigger is simply log a row into our AuditLog table for every row that was actually modified. Let me create that trigger. And then we'll check to see if there's any data in the application log table, so there's not. And then to demonstrate this, I'm going to turn on STATISTICS TIME settings. That will show us how much time was spent in CPU and the overall execution time of the SQL statement. And finally, I'm just going to run basically the same statement two times so that we can see if the time that the SQL statement takes is about the same. I'm simply going to increase the quantity by 1 where orderId less than 25, 000, and then I'll decrease the quantity by 1. The first time through, we'll see that this takes a little bit more than 3 seconds in total time. Let me decrease that quantity again, the same action just a different direction, and it takes about the same time, around 3 seconds. Now as more and more orders are added and more and more lines are created, if something like this had to occur, it probably wouldn't be the quantity column, but there might be something, maybe a setting that determines things that have been shipped or not shipped, and inadvertently someone were to update a column, if you were logging all of those modifications, the trigger's going to be holding open that transaction longer than you expected. One way to mitigate this is to simply add a check. Now as I've said in many of these demos, admittedly, this is not the best way to do this, but allows me to show you how simply removing work, avoiding the work of that log, can actually improve the response time of your trigger. So, I'm simply going to say if there has been any log for today that is for an update for this order ID, don't log and update again. I don't care that it's been updated multiple times. Again, this probably isn't what you would do in production, but I want you to see what it looks like when we do that check and the impact it has on our simple statement. So I'll modify that trigger to do that check. And now we're going to run that statement two more times. Again, we're going to increase the quantity by 1 and then decrease it by 1, and we'll see what the time is for each of those queries. Now remember before, it took about 3 seconds to execute that. This time, it takes a little under 2 seconds to actually finish the query. Let's do it a second time, and we get similar results, just under 2 seconds. So simply by checking to see if we need to do that additional work, we were able to save time that the transaction was held open on that table. While this specific example might not really make sense in the context of your application, hopefully seeing that just one simple change can have a dramatic, more than 30% impact on the execution time on my trigger will get you thinking about all the work that's happening inside of each of your triggers and how that's impacting the performance of your application. Above all, be diligent not to let triggers become that hammer for all of the problems you see within your database.

Improving Performance with Service Broker

As we finish our discussion and demonstrations on how to use triggers within your database, I want to discuss one option of improving performance with Service Broker. SQL Server Service Broker is an in-database messaging system, something you will often hear referred to as a queue in other systems. It allows messages to be delivered to queues which are acted upon at a different time and in a new thread. Triggers can use Service Broker to defer work until later, and this is especially helpful when triggers begin a chain reaction of additional work. Service Broker is a deep topic and something we cannot deal with in any depth within this module, but I wanted to give you a basic understanding of how Service Broker works and how it can help with your trigger execution. Service Broker works on the concept of a conversation, and generally there are at least two queues involved, one that initiates the conversation and one that receives that message and does something with it. So we have an initiator queue that sends a message to a target queue. That target queue picks up the message and does some work on it, usually in the form of a stored procedure. Once that target queue is done with the work, it prepares a message and sends that message back to the initiator queue, usually signaling that the work is done. We can use this to our advantage within the trigger. We can simply pretend at the beginning of this conversation to be the initiator queue by dropping a message and having that queue send it to the target queue. Our trigger has now gone and closed its transaction and moved on, but the target queue is able to pick that message up, do some work with it, respond to the initiator queue as if it is actually the one that wanted the work to start, and our trigger has continued on. Our thread is done, and we are no longer held up by the work that might have been done by a stored procedure or maybe a chain of stored procedures that needed to update information within your database. This is really freeing when you need asynchronous communication dealt with by a trigger. A couple of things to consider when thinking about using Service Broker for asynchronous trigger work. Any solution that involves asynchronous triggers must take into account those many rows. Now we've talked about this over and over, inside of the trigger, a network's on a batch of data. When you use a trigger to deliver messages, you need to know how to take those many rows and turn them into messages that Service Broker can actually deal with effectively. Sometimes when a statement affects thousands or millions of rows, the documents that are often XML that are passed back and forth between the queues can get really large, and sometimes, if you don't do it correctly, the implementation of the Service Broker trigger can actually be worse than the original implementation you were trying to speed up. However, if you implement asynchronous triggers correctly, depending on your workload, they can be a really great tool to improve the performance of the work within your database.

Demo: Using Service Broker in Triggers

In this last demonstration on triggers, we're going to see how Service Broker could be used to speed up a trigger that normally takes a long time because of other stored procedures and functions it calls. To see how Service Broker could be a help in some situations, I want to show you a contrived example, but one that I've seen in a number of databases over the years. Often, people will put stored procedures inside of triggers to be executed when a certain action takes place. Often this is used for something like a rollup table, something that needs to be updated as orders come in or orders are deleted. You want values in another table or maybe even in another database to be updated almost in real time. So that's what I've done in this sample database. I've created a sales table that is rolled up by stock item, by year and month, and simply gives the total by customer so that I could do in another application or maybe in something like Power BI some quick rollup information without having to query many, many tables at once. Now, to show you how this works, I'm going to use a tool from Microsoft that allows me to quickly insert many orders into this database. Currently, we have almost 84, 000 orders inside of this database. That will be important in just a minute as you'll see this take place. Now, to simulate a trigger doing a lot of work, I've put a very intentional 2 second sleep inside of the stored procedure. While this might seem unlikely, you'd be surprised, depending on the size and scope of your database, how quickly under load some of these stored procedures can take to execute, particularly when they're inside of a trigger, something that's getting called often when a database is very busy. You'll see this 2 second delay become very evident in a minute. So to start, this example is going to put a trigger on the sales order table that takes each OrderID and then runs it through a cursor. And for every order that comes in as part of that batch, it will execute the Sales.DoOrderRollup stored procedure. In this case, the tool inserts one order across many threads, so in reality, this trigger is firing once per order. And there is actually no multi-row batch here, but you'll see that having that delay simulating a lot of potential work that's slowing down the execution of each trigger will have an impact. I'm going to add that trigger onto that table. Now I'm going to use a tool from Microsoft that allows me to insert many orders at once. It does this across multiple threads. When I click the Insert button, it will begin to insert data, and it takes some time to get feedback from this application. Normally I get pretty quick response, but you'll notice this Average Order Insertion Time is really taking a while to update. And finally, we start to see it updating, but the numbers are pretty dramatic. Now let's see how many of those orders actually got in once I stop this. Alright, so let's go back to SQL and see how many orders we had. Now we started with not quite 84, 000 orders, and we'll see we really didn't get that many more. Now we were trying to insert across 10 threads, and it really took a long time. So now let's see what happens with Service Broker. The makeup of this trigger and the SQL inside of it is really far outside the scope of this Pluralsight course. What's going to happen is that I've already set up a Service Broker application within this database, and as new orders come in, I'm going to place an XML document containing the OrderID onto a queue like we saw earlier in those slides. The target queue will receive those messages and execute that same stored procedure in a different thread. It will still take 2 seconds for every one of those messages to finish, but it won't hold up my thread. As the application that's trying to insert these orders, I'll get an almost instant response time. Let me go ahead and replace the trigger with this new one from Service Broker, and I'm going to open that application back up again, and now let's go ahead and click Insert and see if we notice any difference in the Average Order Insertion Time. Oh my goodness. We get almost near response, and I'm at 13 ms. That means there's almost no delay in the insert of these orders and that trigger firing over and over again. The stored procedure's off doing its own work in a different thread, but that's not impacting my work here. Now I'm going to stop that, and let's go see what happened with the actual number of orders. Now if you remember, we only got a few more orders here with the previous example, so we're at about 83, 600 orders. I let that run for about 20 seconds, and let's see how many orders we have now. Wow! What a difference. That's 20, 000 orders almost that were inserted in that short amount of time, but with the old trigger that was delayed and waiting on that stored procedure to finish. As an example of a stored procedure doing lots of work, we got a very big impact in performance improvement. Now as I said earlier, Service Broker is not for the faint at heart, but if you're willing to put in the work and you have some of these sore spots, it can be a great tool to help you improve the efficiency of your triggers and the work that they're doing.

Summary

In this module, we talked about security and execution context of triggers and how you can use that to your advantage when dealing with security issues within your database. We looked at the unexpected behavior of triggers when you use a MERGE statement and the ways that you need to account for that. We discussed how you might be able to actually get some data out of a trigger outside of the transaction even when rollback occurs using table variables. We looked at not overusing triggers, thinking about all of the work that it's doing and how they interact with the system that you're developing in your database. And finally, we looked at how using Service Broker in some circumstances through asynchronous triggers really could help your workload and improve the interaction of your database with your application.

Reusing Code with Functions

Overview

Welcome back to Programming SQL Server Database Triggers and Functions. In this module, we'll be talking about reusing code with functions inside of SQL Server. We'll first talk about what functions are within SQL Server, both server-level functions and user- defined functions. We'll talk about why they're a useful to have within your tool belt. Then we'll discuss the difference between deterministic and non-deterministic functions, again, when they're valuable and what you must think about when creating and using them. And finally, we'll talk more about user-defined functions and group them into two main categories, multi-statement user-defined functions and inline table-valued functions. We'll discuss the pros and cons and eventually see how some newer versions of SQL Server can help your functions run more efficiently and be more valuable to you and your users. Let's go ahead and get started.

What Are SQL Server Functions?

So what are functions within SQL Server? Within SQL Server, functions refer to the reusable code specifically for querying information or performing repetitive actions on data. Within SQL Server, this is more like a functional programming rather than the reusable kind of composable subroutine methods you might be used to with object-oriented languages. They're very specific and narrow in their use cases, and they only work on the data passed into them. Now we use functions all the time when we're using T-SQL. We just might not realize that's what we're doing. The MONTH function that's built into SQL Server takes data in, a specific date, and passes back the month portion of that date. There are two kinds of functions within SQL Server. First, there are the built-in functions within SQL Server, things that we honestly use every day when we're writing SQL. And then there are user-defined functions, otherwise known as UDFs, and this is really going to be the focus of the rest of this course. So what are some built-in functions within SQL Server? Now, on the screen, you'll see just a handful of the things that you've probably used every day as you're working within the database, things like GETDATE, and CAST, and SUM, and making strings uppercase or lowercase. Those are built-in functions within SQL Server. Most of the time they either act on data we give to them, or they simply provide information and data back based on the context of what we're doing. As an example of context, something like EVENTDATA, a function we looked at earlier in this course, provides information within the context of DDL and logon triggers at the context at which they're happening. Now, user-defined functions are obviously created for application-specific purposes. You want to have some kind of reusable code, something you do over and over again and much easier to access. Functions can return a single scalar value or a table of data, and they must adhere to specific rules within SQL Server. The first rule is that every function must return a value. It doesn't matter if it's a scalar or a table value. It must actually have a return statement and a value that's returned from the function. You cannot create a function unless it returns a value. Now with functions, you can only take input parameters. You cannot provide data back out from the function through some linking output parameter like you can in a stored procedure. Functions can be used in SELECT statements, they can be used in JOINs if it returns a table, they can use table variables inside of them, and they can be bound to the schema to prevent modification of your database if a function relies on it. But there are a couple things that functions cannot do. First, they are not allowed to alter the state of the database in any way. There are no DML actions that can be run that will alter the database state. They can't alter their transaction. And although they can use table variables, they cannot use temporary tables. They cannot execute stored procedures. And finally, they cannot use a try/catch block within the function itself. Functions are meant to be a very specific way of manipulating data that's passed in so that it can provide results of something within that context. I also want to say that functions are not catch-all stored procedures. If you come from another database technology like Postgres, everything in Postgres is actually a function. Stored procedures run as functions, triggers run as functions, and functions run as functions. That is not the case in SQL Server. Stored procedures, functions, and triggers all run within various contexts and have very unique and specific rules that are applied to them. So recognize that functions are really about helping you use the data at query time to determine how to interact with the data and the database around you. Now within SSMS, functions are found within each database in the Programmability Functions folder. As we'll see in a bit, there are different kinds of functions, and you'll find individual folders for each of those function types.

Why Are Functions Useful?

Now hopefully it's almost immediately obvious why functions are very useful to the developer within a SQL context. As I showed earlier, things we use every day, something as simple as the MONTH function within SQL, provides a ton of value for us. If we had to include the SQL that did that every single time, it would be a lot of extra work that we wouldn't always get right. There might be a need that you often have to convert data and dates within your database into fiscal years. Maybe you have reports that run on a fiscal basis, so you have to take a date like December 31st of 2018 and convert it into the fiscal year of that system. In this case, the year ending in 2019. Now if you had to do this work over and over again and many queries, there would be multiple times that you would make mistakes, and you might not get it right, and every time that you made a change because of a new requirement, you'd have to do it many, many places. And that is the most obvious reason we want to incorporate common logic within functions. Now functions provide reusable query time logic. So that's exactly what I'm talking about. When I need to do that same kind of manipulation over and over again, it's a great use case for a function. It provides predictable query optimization when they're used right, not all the time as we're going to see. But when you use functions in the best possible way, you can often get very predictable query plans. And finally, you can almost be assured every time that the usage will be correct. If the function does the work and you teach your staff and your developers how to use those functions correctly, you'll get the results that you expect each time. Now functions can generally be used anywhere a scalar value or a table would be used. That means they can be used in SELECT statements and WHERE clause. They can actually be used in the constraint definitions on tables and computed columns. They can be used as a table to join to. They can be used within stored procedures. Even though functions cannot cause stored procedures themselves, within stored procedures, you can use functions. And finally, functions can call other functions. Recognize that just like triggers, when you nest functions, you are limited to 32 levels of nesting. Functions overall are a great tool for the SQL Server developer to know and use for all of these reasons and many more that we'll look at. However, much as I warned at the end of the modules dealing with triggers, functions are not the silver bullet of every single problem. If you don't know how to use them well and if you're not very deliberate about how to make them work the best with your system, your schema, your database, they can really impact the performance of your system and make it very difficult to debug. Use them wisely, and I guarantee they can be a joy.

Deterministic vs. Non-deterministic Functions

When creating functions, they can be either deterministic or non-deterministic. Knowing which they are does actually mean that SQL Server will interact with them differently. A deterministic function is one that given the same input, will always return the same result. If I pass the same exact year into the YEAR function, I will always get the same result back. Non-deterministic functions, however, do not guarantee the same result each time. Now there are numerous system functions specifically that are non-deterministic. Something like NEWID, regardless of what context it's called in, will always provide a brand-new ID. When you create user-defined functions, in general, your aim is to create deterministic functions. They allow SQL Server to do a little bit more planning if it can guarantee and know that the same kind of result is going to come back every time. There are two options that must be provided as a starting point. The first is to add the WITH SCHEMABINDING clause to your function definition. This prevents modification of the underlying schema and database and any function that references that information and schema. And the second thing that your function can't do is call other non-deterministic modules. If it does, then your function will not be a deterministic function. And it doesn't mean it's bad, it just won't be able to always take advantage of things like query plan caching that can help your server perform as best as possible.

Multi-statement vs. Inline Table-valued Functions

Before we actually start to look at functions within SQL Server and how they work, we need to talk about the core concept of multi-statement and inline table-valued functions. In reality, SQL Server currently has three specific kinds of functions. One is called a scalar function, the next type is a multi-statement table- valued function, and the third is a single- statement table-valued function. A number of years ago, however, in a book on SQL Server Internals by Dmitri Korotkevitch, I was first introduced to the idea that there are really two major kinds of functions, multi-statement and inline table-valued functions. It helps to think about it in these more simple terms for a couple of reasons. Now with a multi-statement function, we can have a scalar or a table variant, but multi-statement means any function that has a begin and an end statement inside of it is a multi-statement function. The scalar variant returns a single value of the declared type. A multi-statement table function also has a begin and end, but this returns a table. You specify the columns that are going to be returned, and somewhere within the actual function body, you populate those fields row by row to return to the calling statement. Whether using multi-statement scalar or table-valued functions, there are a number of reasons to use them with caution. Specifically, with scalar functions, they cannot be inlined prior to SQL 2019. This means that for every row that the function is being utilized on, it must be called over and over again. There's no way for the query optimizer to figure out a more efficient way to use that function. And so when there's non-trivial work happening within a scalar function, it can really have a major performance impact on SQL Server prior to SQL 2019. Now when dealing with table-valued functions, we have some of the very same problems. Up until SQL 2017, they also cannot be inlined to the calling DML statement, and again, that means the query optimizer really doesn't have any way to optimize what's going on there. Now they are only executed once per request, so that is a little bit different from the scalar functions. But as I said, the query optimizer doesn't know how to take the SQL that's within this function and make it better using the statistics of the table that's going to be returned. And because the optimizer can't do anything specific with the table that's returned, the team simply chose to give the same estimation of 100 rows to every single table-valued function. It doesn't matter if it returns one row or returns millions. The function always assumes only 100 are coming back. And you can see why when there's non- trivial work going on this can have a major performance impact, specifically in versions prior to SQL 2017. The second kind of function is an inline table-valued function. These are functions that return table data directly from a single SQL query within the function itself. It's a single statement that can actually be inlined. The optimizer can take the SQL that's within this function, replace the usage of it within the calling DML statement, and actually perform some optimization on that query. In general, you will get much better performance with inlined table-valued functions wherever you can use them. But you might ask, in all reality for the work I'm doing, does it really matter if I use one kind of function over the other? Well, as you've probably heard many times when talking about technology, it depends. Let me provide a real-life example of the impact that choosing one over the other can have within your work. The query on the top is a multi-statement scalar UDF. It takes the orderId that's being passed in, takes the modulus of it of 10, and if that returns 1, it returns the row. If I take the same work, but turn it into an inline table- valued function, taking in the orderId, modding it by 10, and returning a table with the values of 1, I can get the same information back, but the optimizer has a chance to take that SQL and inline it into this calling statement. So what does that look like if I call this on a real table? In the orders table with about 75, 000 rows, the top statement returns in about 300 ms on my computer. Doing the exact same work and returning the exact same rows with an inline table-valued function, returns in about 52 ms. Although the top query often feels a little bit easier to understand, the performance impacts can be very, very noticeable. Inlining it made the CPU usage on my computer better by 12 times, and the actual overall runtime was decreased by 6 times. In general, one of the reasons you'll see people tend towards the multi-statement functions is quite honestly they're easier to implement. The individual plans from the queries inside the functions can be cached if written appropriately. And when you have very complex logic, it's often easier to understand what's going on if you can break that up into multiple steps almost like a stored procedure. And multi-statement functions can have SCHEMABINDING, so you get that safety and control that the schema beneath this function that it relies upon can't be changed. In contrast, inline table-valued functions often do take a little bit more creative thought to implement. But again, SQL Server can inline the SQL that you're using into the calling statement, and that provides a nice path for optimization. It also provides a predictable batch workflow. You know exactly the kind of work and the table that's coming out, and the optimizer knows how to cache that and provide the best plans for the workload that you're giving it. And they can also have SCHEMABINDING applied to them, still giving you that same safety and control for the underlying schema that these functions rely upon. However, even knowing that it's often a little bit more work to implement inline table-valued functions, the prevailing wisdom of the SQL community is that you will get a better plan, you'll get better performance, and ultimately your system will perform better for your clients if you can make that work. Now there is good news. The SQL Server team has worked hard with SQL Server 2017 and 2019 to address some of these concerns specifically relating to performance and these multi-statement functions. They're not able to take care of everything, but it is another tool for you to be able to use better as newer and more performant versions of SQL Server come to market.

Summary

In this module, we gave you a brief overview of functions within SQL Server. We talked about built-in and user-defined functions, when and why functions are useful, specifically user-defined functions that help you do the work for your specific application. We talked about the basic rules that functions must follow as you create them, what deterministic and non-deterministic functions are, the difference between scalar, multi-statement, and single-statement table-valued functions, and ultimately how we can take those three kinds of functions and break them down into the two groups, multi-statement and inline table-valued functions to think about and remember that when you have multi-statement functions, they often have more performance concerns and something you have to be aware of.

Scaler and Table-valued Functions

Overview

Welcome back to Programming SQL Server Database Triggers and Functions. In this module, we'll discuss scalar and multi-statement table-valued functions. In this module, we are going to briefly review multi-statement user-defined functions. We will create some examples of scalar and multi-statement table-valued functions so that you can see the difference between the two. We'll demonstrate how you can use each of them within SELECT statements and, in the case of scalar functions, within WHERE clauses. We'll discuss the complications that multi- statement functions cause when it comes to SARGability, and we'll discuss exactly what SARGability is and why that's really important to understand. And we'll finish by giving an overview of some improvements that are coming in SQL Server 2017 and 2019 that improve multi- statement function performance. Let's go ahead and get started.

A Closer Look at Multi-statement Functions

Let's take a closer look at multi-statement user-defined functions. As we said in the previous module, functions are an essential part of having a productive SQL Server development environment. We use them every day. Many of the functions we rely upon are built into SQL Server. They're a part of all of our code. Remember that functions can only have input parameters, but they can have default values defined. All functions must return a value, and they can query and modify data locally internal to the function, but they may not use any DML or DDL statements that would modify the database or schema. And as we said in the previous module, there are two broad categories of functions, multi-statement and single- statement functions. And until recent versions of SQL Server, multi-statement functions cannot be inlined. As we go through the rest of this module, you'll understand why that's really important. First, let's discuss multi-statement scalar functions. Anytime someone discusses a user-defined scalar function in SQL Server, they really mean a multi-statement function. That's because there is actually a definition for a single-statement scalar function, but it's never actually been implemented. So when we say scalar functions, we do mean multi-statement. So all scalar functions must have a BEGIN and an END statement within their definition that makes them multi-statement. They must have one or more statements, and it must return a value of the specified type. And they can be used as a value in DML statements, some schema definitions, and in WHERE clauses. This is the most basic definition of a scalar function. As you can see, like triggers, in SQL Server 2016, we now can use the CREATE OR ALTER statement when defining a function. We are allowed to have input parameters or not. We must define the return type. As we discussed in the previous module, it's almost always advantageous to add WITH

SCHEMABINDING to your definition. And finally, you'll see that we have a BEGIN and END statement that returns a value. Don't let this fool you. Just because there's one statement doesn't make this a single-statement scalar function. It has a BEGIN and END statement; therefore, it is a multi-statement function. We just happen to have one. We could do the exact same thing by defining a variable, adding those two numbers together inside of that variable, and then returning that as the value. Again, it doesn't matter if there's one or many statements. Technically, this is a multi-statement function. Now we can use scalar functions in multiple places. We can simply pass in a parameter directly in a SELECT statement. We don't have to join to any tables or do anything like that. We can simply use that function to return a value. We can use a scalar function within a DML statement and utilize table data from that statement as an input parameter. And finally, we can use scalar functions in WHERE clauses as part of the predicates. Now this is not the preferred method of using scalar functions, something we'll see in a little bit as we discuss that funny term called SARGability. In contrast, multi-statement table-valued functions declare a table as the return type, and you must declare the actual columns that will be returned from the function. Just like the scalar function, it must have a BEGIN and END statement, and then it can have one or more statements that must populate that table and then return the declared table from the function. Multi-statement table-valued functions can be used anywhere a table input would be used. Something like joining to a table, you can join to a multi-statement table-valued function. It can also be used as a target of an APPLY statement. Defining a multi-statement table-valued function is very similar to the scalar function. The only difference is what you return. In this case, we define the table variable that we're going to return, the columns inside of that table, and then within the BEGIN and END statement, we insert those values into the table that needs to be returned before finally returning it from the function. Just like the scalar functions, we can use these in multiple ways. Because they act as a table, if you want to simply select from them, we can simply select from that table-valued function. We can use it as a join just as we would another table. And as stated earlier, we can also use it as a target for a CROSS or outer APPLY to do filtering.

Demo: Multi-statement Scalar Functions

In this demo, we're going to be creating and using multi-statement scalar functions within our SQL. To begin thinking about scalar functions, I want to show you a SELECT statement that simply uses all functions and doesn't actually even select any data from the database. We'll simply get the current month and current year of today. So without even selecting any data, I can use functions. Now let's create our first user-defined scalar function. You'll see, just as in the slides, this is our SuperAdd_scaler function. It will take two integers, add them together, and return that integer. Remember, we always need to return a value of the type that we specify. I'll create that scalar function and then simply select from it. Again, I'm not using the database tables or data in any way. And as expected, we get a result of 4. One of the nice things about functions in general is that we do get some sense of type checking. So as you begin to type the scalar function in your SQL, you will be prompted with the various parameter inputs and their types. It also allows us to get that type check when we pass in a bad value. You'll see, just as we would expect in any other context of SQL, we're told that we can't convert that character into an integer. Let's take it a step further. Let's actually now see the multi-statement nature of these scalar functions. I'm going to take in parameters for a date and what I'm calling a FiscalEndMonth. In a business calendar, we often talk about a fiscal year based on the month that it ends. So if your fiscal year ends in June and we're currently in September of a year, then our fiscal year is actually the following year. So this function will simply adjust that year and return the appropriate fiscal year. Let's go and create that function, and then let's see what happens. So, as I've said, I don't actually have to yet query the database for any real data. I can just keep using this function because it's something that I can query on. Now, in these five statements, I would expect to get a couple of them back as the year 2019 and a couple of them back as the year 2020. Those sales dates have advanced past the end date of the fiscal year, so they will actually report the following year as their fiscal year. And sure enough, I see a couple 2020s here that signifies that this is adjusting the date appropriately. All the dates I passed in were for the year 2019, but some of them got adjusted to next year. And now I can use this function on real data. So I'm going to select information from the Sales.Orders table. I'm going to use the order date from every row that's returned, use that as an input to the FiscalYearEnding function, and output the fiscal year. I expect, because I'm selecting information from the end of June forward, that some of my dates, as I get into July, will report the next fiscal year. Let's see what happens. So as you can see, I start at the end of June, and eventually, as I get to July, because I said that by default fiscal years end in June, the July dates are now part of the next fiscal year. You can also use scalar functions in your WHERE clauses, something we discussed earlier. I can say select the top 100 orders using the OrderDate into one function where the year of that OrderDate is greater than 2015. So this should return the top 100 orders whose order date is in the year 2016. We do get 100 rows, and we see that we're getting information for the year 2016. Now the fiscal year itself is not advanced to the next fiscal year because all of these records are simply for January of 2016. However, if I use the FiscalYearEnding function and pass in the same OrderDate, I will actually get rows out of this table that begin in July of 2015 because by default they are after June; therefore, they're part of the next fiscal year. Let's see if that's what happens. And sure enough, we start getting rows back for July of 2015 because their fiscal year is technically for the following year.

Demo: Multi-Statement Table-valued Functions

In this demo, we're going to be creating and using multi-statement table-valued functions and show how they differ from the scalar functions we created earlier. To start looking at multi-statement table- valued functions, we're going to create the same SuperAdd function, but as a table- valued version. It looks very similar to the scalar function we created with a couple of the differences we've already talked about. We defined the table variable that we're going to return and the columns, including the data type that will be returned for each of those columns. We then have the BEGIN and the END statement. We select into that table variable the columns of data that we want to be returned, and then we return that information. So this is going to look very similar to the scalar function. We're going to get the same result, but it's actually a table of data that we could join to if we needed. Next, you'll notice one other difference.

With the scalar function, we treated it generally as a column. We could simply select the scalar function. Because this is a table function, we need to treat it like a table and select the columns or all of the columns out that we want to get back. We'll pass in the same two integers, 2 and 2, and expect a value of 4, which we do get. The other thing you'll notice is that because we define that table, we actually get a named column back for that value. Now let's look at what a real multi-statement function might look like. In this case, I'm going to create a fairly simple table-valued function that's going to look at the sales order table and return the average min and max sale value for each customer within a start and EndDate time range. As part of this function, I'm going to add some logic. Specifically, I'm going to check if the EndDate is after the begin date. If for some reason it's not, I'll set the EndDate to 10 days after the StartDate kind of as a safety check. That allows me to show you that you can do all kinds of logic similar to a stored procedure inside of these functions. Then I'll take that SELECT statement with a CROSS APPLY, get all of the values I'm expecting, the min, the max, and the average for this customer for that date range, and I'll return that as the table that I specified. With that new function, I treat it like a table, passing in the appropriate input parameters, and I should get some values back. In this case, I'm going to use customer 10 for all three statements. The date range will be different for the values I expect back. For six months of time, I get the maximum, minimum, and average. The second statement is one day apart, so it will still pass that check. Notice these values, 261, 277, and 269. In this third statement, I'm actually passing in an EndDate that is before the StartDate; therefore, I expect those values to be very similar to the previous statement. And sure enough, they are. I can also use this as a table for the target of an APPLY statement. This allows me to iterate row by row over the customer table using that new function to return those values for a specified date range. And then in this case, I'm only going to show when the returned rows have a minimum sale of 100. So for all customers between January and June of 2015, who has a minimum sale greater than 100? And sure enough, I get this back. You'll see that all of our minimum sales are above $ 100.00. So that's how we can use a table-valued function to act as a table, do some special work that returns value specific to our application in need.

Challenges with Multi-statement Functions

There are a few challenges that you must be aware of when working with multi-statement functions. Specifically, multi-statement scalar and table-valued functions do suffer from known performance problems within SQL Server. That's one of the reasons you'll often hear SQL Server professionals caution and warn people not to use scalar and table-valued functions. However, that is beginning to change with newer versions of SQL Server, specifically SQL Server 2017 and 2019, something we'll discuss briefly at the end of this module. There are two main reasons why multi- statement functions inhibit the performance of your SQL. The first is what's called SARGability. SARG stands for searchable arguments, and this refers to the way in which SQL Server identifies the indexes it can use based on your search predicates and the WHERE clauses. When we use any kind of function on the left-hand side of a predicate in the WHERE clause, SQL Server can no longer use indexes. It identifies the item on the left as the column it's looking for an index on. When you now curate a random value by using a function, it no longer can identify a column and an index to use. If you often have situations where you need to identify data based on something like the month or the year and you are used to using that function on the left-hand side of the predicate, one option could be to use a precomputed data table of some sort that allows you to rework your predicates so that indexes can be used. And the second reason that multi-statement table-valued functions really become a performance inhibiter inside of your SQL queries has to do with statistics. Remember, what you get out of a multi-statement table-valued function is a table, but it's a virtual table in memory that never existed on disk, and no statistics have been or could've ever been created on it. Therefore, the SQL Server team had to make a decision. Beginning with SQL Server 2014, the team decided that every table- valued function would automatically be given an estimate of 100 rows regardless if that table-valued function returned one or millions of rows. And this can have a dramatic impact on the performance of your query. Making this generic estimate of 100 rows severely limits the optimizer's ability to correctly find the best plan for the SQL you've given it.

Demo: Estimation Problems in Multi-statement Functions

In this short demonstration, we're going to identify the impact that scalar and multi- statement table-valued functions have on query optimization and how you must address this in your own SQL queries once you understand what's happening. To demonstrate both of these problems that multi-statement functions can add into our queries, I want to start with a scalar function and how when we use it in a WHERE clause we actually impact the ability of SQL Server to do its job well. I know we've all done this numerous times. Dates are an easy way to demonstrate this because there's so many times where we want to get rows out of a table that are based on a specific month or year, and it's so much easier to simply use that month and year function to get those rows out. But I want to show you in a very simple way the impact this is having. Now again, in a small database like WorldWideImporters, it's kind of hard to show the grand impact this can have on a production system, but once you see exactly how these interact with the query planner, you'll start to think about how you're doing this in your queries and the impact it's having. To start, I want to show you that I did create an index specifically for this demonstration. It is on the OrderDate, and it includes the columns you see here in my query. When I use the index appropriately by specifying the dates, keeping the column name on the left, SQL Server is able to use that index, and you'll see in the execution plan it actually did a seek, and it knows exactly how many rows are coming out, 1, 830. It has that index. It knows how many rows are in the month of January for the year 2015. However, if I do the exact same thing, this should get me the exact same 1, 800 rows. But by using these functions as the left- hand side of my predicate, SQL Server can no longer go and use that index. Instead, what it will do is go to that index and do a scan of the whole index. Now depending on the size of the table and the kind of index it is, that might not seem very bad, but if you're using these functions on many different tables within a large query, your execution plan could really be impacted. In this case, simply by using that function, it actually read the entire index, almost 74, 000 rows simply to find the 1, 800 that it needed. That is how using functions as part of your predicates in the WHERE clauses could be dramatically impacting the performance of your SQL. Now the situation with multi-statement table-valued functions is slightly different. Remember that we said the SQL Server team decided to give the same estimate to every single execution of a table-valued function, 100 rows. It doesn't matter if this function returns one row for one customer or returns thousands or millions of rows. When I execute this function and this statement, you'll see in the query plan that even though one row was actually returned, it estimated that 100 rows would be returned. This could have a dramatic impact on the execution of your SQL statements. The same thing happens even with this CROSS APPLY. You'll notice that the execution plan is a little bit more complicated because it has to do the CROSS APPLY, but it has no idea how many rows are going to come out for each customer for this time range, so every single time it estimates 100, regardless of how many truly come out. You'll notice it takes some time to execute this statement, and part of that is because the ultimate filter that gets into this table-valued function still estimates 100 rows even though it was actually executed 663 times. There is no way for the query planner to take the statements inside of that function and make them part of the containing SQL. There's no optimization that could occur. Hopefully you can see how these two things could really impact the SQL that you're writing, and you'll begin to think about how you can improve your queries and find those rough spots that are causing your system to be slow.

Improvements in SQL Server 2017 and 2019

One of the great things that's occurred over the last number of years with SQL Server is the pace at which Microsoft and the SQL Server team continue to release improvements to the core software behind everything that we do. I'd like to discuss two changes in recent versions of SQL Server that you will be able to take advantage of as you have the opportunity to upgrade or potentially to use the online version inside the Azure ecosystem. The first is interleaved executions from multi-statement table-valued functions. This was introduced in SQL Server 2017. This change allows SQL Server to identify that there is a multi-statement table- valued function inside of the query. It pauses the query momentarily to execute that multi-statement table-valued function. It gathers the proper statistics and uses those rows to determine if it wants to change the execution plan. One of the great things about this change is that the multi-statement table-valued function execution plan is actually cached. It does not need to be run every time that calling query is made. This means that on each subsequent query, if that multi-statement table-valued function is identified, it can use the cached plan. This brings all the benefits of Query Store. For instance, if SQL Server chose a new plan for that multi-statement table-valued function that was less efficient, you could actually use Query Store to force an older plan. Now there are a couple of things that need to happen in order for your current multi- statement table-valued functions to be eligible. First, you must be using SQL Server 2017 or greater, and that means that the compatibility level of your database must be set to 140 or higher. Even if you import an older database into a new version of SQL Server, if you don't actually update that compatibility level, you will not get all the benefits of that new software. Any referencing statements must be read- only, and they cannot be part of any data modification operation.

The other current limitation on interleaving these multi-statement table- valued functions is that they cannot be used as a target of CROSS APPLY. Scalar UDF inlining is one of the most exciting things for old-time query tuners. This feature is very similar to the interleaved execution, but this time for scalar functions. Much like the interleaved execution, it identifies scalar UDFs and, where possible, transforms them into an expression that can actually be inlined into the calling query. This essentially removes the scale or UDF operator that you'll see inside of query plans. It hides all the details of what that scalar UDF is actually doing. Now you'll get to see the work inside of the overall execution plan. Much like the interleaved execution feature, inlining scalar UDFs does come with some limitations in this current iteration. You must be using SQL Server 2019, and your database must have a compatibility level of 150 or greater. The SQL inside of the scalar UDF cannot use any time-dependent functions such as GETDATE or NEWID. As we talked about in the security and permissions of triggers, the execution context of the scalar UDF must be CALLER in order for inlining to be possible. There are a number or restrictions around GROUP BY and ORDER BY and where the function is used. And honestly, the easiest way to determine, once you upgrade your database, to see if your functions can be inlined, simply select from the sys.sql_modules table, and find your scalar functions, and look at the is_Inlinable column. If it's true, there is a good chance that SQL Server will attempt to use your function inlined. And because this is such a new feature that has not even been officially released at the time of this recording, I've included the link to the documentation as it currently stands on the Microsoft website. These features, specifically dealing with SQL Server 2017 and 2019 relating to multi- statement functions, is very exciting and allows a lot of opportunity for immediate performance gains as you upgrade to newer versions of SQL Server.

Summary

In this module, we've reviewed what multi- statement functions are. We discussed how to create scalar and table- valued functions within SQL Server. We discussed the problem of using functions as predicates and how it impacts the SARGability of your queries, something you need to be aware of as you are trying to make your queries more and more performant. And finally, we discussed new improvements in SQL Server 2017 and in 2019 that are beginning to give us tools for making these scalar and multi-statement functions more performant with newer versions of SQL Server.

Improving Function Performance with Inline Table-valued Functions

Overview

Welcome back to Programming SQL Server Database Triggers and Functions. In this last module, we're going to discuss inline table-valued functions. To begin, we'll review single-statement, otherwise known as inline, table-valued functions. We'll create a simple inline table-valued function to demonstrate how they're different from scalar and multi-statement table-valued functions. Once we see the value that they can bring in contrast to those scalar and multi- statement functions, we'll look at ways that you could convert a multi-statement function into the more performant single- statement inline table-valued functions. And we'll finish by talking about one potential problem with inline table-valued functions known as parameter sniffing. Let's go ahead and get started.

A Closer Look at Inline Table-valued Functions

In the previous module, we looked at multi- statement table-valued functions. Let's examine how single-statement table- valued functions are different and often preferable. First of all, single-statement table-valued functions are often referred to as inline table-valued functions, or simply ITVFs. They have no BEGIN or END statement, and they only contain one SQL statement that returns a table of data. You cannot have multiple separate statements chained together inside to produce a result. They can be used anywhere a table input would be used within your SQL query just like multi-statement table-valued functions. The definition of an inline table-valued function is very similar to our other definitions. The big difference is that we simply return a table. We don't have to define the table variable or the columns that we're going to return. And then you'll notice there's no BEGIN and END statement. We simply RETURN, and inside of those parentheses is the SQL statement that would return the result we are expecting. Just like multi-statement table-valued functions, ITVFs can be called independently just like a table. They can also be used inside of DML statements, utilizing them just like another table. And finally, ITVFs can also be used for all of our APPLY statements, often used with CROSS APPLY to do additional filtering. You may be asking though, why do I keep making such a big deal over the difference between multi-statement and single- statement table-valued functions? They don't look all that different. The real value comes down to this. The query optimizer can take the actual T-SQL inside of that single-statement function and include it in the calling query. This means that there's no more 100 row estimates. Because that SQL is included as part of the actual query, the optimizer can take that into account when finding the best plan. This ultimately means that single-statement table-valued functions are often the clearest and most performant function type within SQL Server regardless of the version.

Demo: Creating Inline Table-valued Functions

In this demo, we'll be creating and using inline table-valued functions. In this first example of creating an inline table-valued function, we're going to take our same very simple SuperAdd and create it now as the third type of function, an inline table-valued function. You'll see that the only difference between this and the multi-statement table-valued function is that we simply declare we are going to return a table. We don't have to name the columns, we don't have to name the table variable that we're returning, and that's because we only have one opportunity to craft a select statement that will return the table of data we expect. So in this case, we're simply going to return a SELECT statement that adds two integers together. And even though we didn't name a table and its columns, we can name the columns that are being returned as part of the SELECT statement to get that context. Let's create that function. And you'll see, just as in our previous demos with our SuperAdd function, this will do exactly as we expect. It will add these two numbers together. We get the SumValue column with the number of 4, and we'll get the same type checking as we have in the other examples, the same error message about the conversion. With a simple example like that, it can be hard to imagine how would I craft a more difficult, more complex SQL statement. In this case, as I mentioned in the slides, a popular way to do this is with a Common Table Expression, otherwise known as a CTE. In this case, we take the SumValues, and we do that inside of a SELECT statement that's one of the CTEs, and then we use that CTE to select from. So let's create that version of the same function, and let's see if it acts the same as the others. Again, we get a value of 4 and a column of SumValue. And just as we'd expect, we get the same error messages about type when we pass in an incorrect value. So those are two ways that you can craft an inline table-valued function as you begin to think about your workload.

Converting Multi-statement to Inline Table-valued Functions

The more time you spend with SQL Server and within the SQL Server community, you'll often hear the recommendation to convert your multi-statement functions into inline table-valued functions. I'd like to briefly discuss some ways you can do that and then demonstrate the value of converting a multi-statement function can have on the performance of your SQL. When converting a multi-statement function into an inline table-valued function, you'll often hear from the SQL community that creativity is key. Sometimes it simply won't be possible because there's so much that happens inside of that multi-statement function that it's quite honestly impossible to reproduce inside of a single statement. As you do look to convert these multi- statement functions, however, you'll often find CTEs are a common go-to tool for dealing with parameters and the data that was previously multi-statement. Remember, however, that it is often worth the effort to convert these multi-statement functions if it's possible so that you can get that inline query and get the best possible plan from the plan optimizer.

Demo: Converting a MSTVF into an ITVF

In this demo, we'll convert a multi- statement table-valued function that we used earlier into an inline table-valued function to see the difference and the potential impact it can have on your SQL execution. In the previous module, we talked about multi-statement functions, and one of the first functions I created as part of the demonstration was this FiscalYearEnding function. And I explained at the time that one of the concepts of the fiscal calendar is often your dates might fall into the following calendar year because your fiscal year ends at some point in the next year. Now, in that example, we passed in a parameter for that ending month so that you could change that value. What I wanted to demonstrate between a multi-statement function and an inline table-valued function is the work that the function is doing. So I've converted that function into a multi-statement table function, and within this, I'm actually doing a SELECT statement to get that fiscal year end month out of a new table I've created inside of this database. That allows the month to be something you can set at the database level, and you don't have to worry about passing it in each time. So otherwise, this function is exactly the same, and we're going to do the same SELECTs. There's nothing new to understand here. I'm going to create that function. So again, the only difference in this function is that we're now getting the setting out of the database. And in this case, I'm going to select more than 100 rows. In the previous example, I limited it to the top 100 rows. And I also want to show you that I've now included the execution plan, something we've only looked at once or twice. So let's go ahead and select that. We're going to get our sales, and again, in this case, I said anything after that OrderDate of June 28th, so I expect it at some point to have that function return the next fiscal year, which it did. The difference here is that this execution plan doesn't include any of that work. We've talked about this before. All we see is that we have a table-valued function that says it's going to return 100 rows. We know that there's a lot more. In this case, almost 64, 000 rows had to go through that function, but all we know is that the database thought it would get 100 rows out of this table-valued function. The reality is, for every sale, it's going to get one row, but it doesn't know how to estimate that. So let's go ahead and change that function into an inline table-valued function and see what difference it makes, specifically in the execution plan. In this case, we know that we have to get that value out of this table. I can't pull it into a variable because I don't have multiple statements that are independent. I have to craft one SELECT statement. And so a popular method, again, is with a CTE. So what I can do is select that value into a common table expression that could have one or more columns and use that common table expression, that table, as the source of my SELECT statement that's going to come out of this function. I don't have to name the table. I don't have to name the column specifically. So let's create that second version. You'll see I've called this FiscalYearEndingDB_ITVF. So it's doing the exact same thing. It's getting that setting out of the table, but now the query planner can at least understand that it's going to have to go do that work, and it might do it in a different way. Let's see what it does. So we get the same result. Believe it or not, that was actually a little bit faster. This is a very simple query, so it doesn't take long in either case. But the difference here is in the execution plan. Two things are noticeable here. One, it was better able to estimate exactly how many rows are going to come out of that table. It knew how many. But you'll notice there's no reference to that multi- statement table-valued function. It just doesn't exist in this query plan. Instead, the engine knew that it had to go get that setting one time. It could scan that table, find the setting once, and then use that in a nested loop for every row that comes out of the index seek. It had a better way to go find this information and make a better guess at the query it was going to write. And that is exactly why this is worth the effort. We enabled the query planner to do its job and do it well. Now this is obviously a very simple statement and much more simple than what you are dealing with I'm sure within your databases, but understanding that you could give that advantage to the query optimizer is really key to how you think about your development. Now, yes, newer versions of SQL Server are going to start to help us with this. Remember though there are some limitations, and so if you really need performance, even today for most of our workloads, you still need to think about the option of an inline table-valued function to have the best performance possible.

Demo: Improving Performance with ITVFs

In this demonstration, we'll take the two versions of our table-valued function, the multi-statement and the inline table- valued function, look at their execution plans to see how they differ, and why it might be worth the effort for you to convert your multi-statement table-valued functions to improve the performance of your application. So now that we've seen the positive impact you can have on how your queries are executed by converting them to an inline table-valued function, that the query planner gets to use that information in a new and better way, let's look at how we could convert something slightly more difficult. This is the function we used in the previous module, this is the multi-statement function, and the tricky part in this function is that check on the EndDate. Remember, we did an IF DATEDIFF logic to see if the EndDate is before the StartDate. We assume that's an error. That gives us an invalid range, and so we bumped the EndDate ahead. How do we do that IF THEN logic inside of a single SELECT statement of some sort to return the same information? Well, let me show you one example of how this could be done. This is one example of how you could convert that multi-statement table-valued function that had the logic and multiple statements and variables into an inline table-valued function. In this case, we have the single-return statement, and this time I'm using the same Common Table Expression idea as I did in the previous demo. I select the CustomerID because I'm going to need to join on it. Now in this case, I select the variable StartDate into just another column StartDate. I don't really have to do that, but you'll see in a little bit maybe why you'd want to. And then we can do the date logic here through a CASE statement. So we get the same ability without it being multiple statements. We do the same logic. If the EndDate is before the StartDate, we're going to bump it ahead; otherwise, we'll use the variable that's passed in. And I use that Common Table Expression, that CTE, to join to later on in this query. I do that with the ParameterData table, which I've aliased to PD. And I do the CROSS APPLY, which gets me the data out based on the order and so forth. And this is the key. Because I've joined to that Common Table Expression, I can use the values that come out of it in my WHERE clause. So now I get the EndDate down here in the WHERE clause that's variable, that does that check and adjusts the EndDate just like I would have in the other statement, but this time the query planner will be able to use all of the SQL within the planning stage of the outer query and get a better plan, and you'll see exactly what happens. Let me create that function. And you'll notice in all three cases, including this one case where my EndDate is before my StartDate, I still get the values I expect. I don't get any errors. I'm getting the same logic. So I have successfully converted this function from a multi-statement function with the same functionality into an inline table-valued function. Now I asked that question in the slides, does it really matter? Well, this is a demo database. It doesn't have load, it doesn't have nearly the scope and size of most of your databases, so let's see if we can even glean some performance advantage here inside of WideWorldImporters. Here's what that function looks like by selecting all of those same values as I did in the previous module using the multi- statement function. Now these values are exactly the same as the previous module. You'll have to take my word for that. I don't expect you to remember them. But the difference is that this query plan knew exactly how to go get the data based on all the indexes for what it was going to do. There's no reference in this plan to that table-valued function with a static estimate of 100 rows. So how do we see if it really has any impact? Well, let's go run these against each other. So this time I'm going to again set the statistics time on so that we can see the actual time it takes. So let's run this inline table-valued function first to get a sense using those statistics of how long it takes. Alright, in this case, it took about 250, 260 ms. We see that very similar execution plan. Does it really make a difference converting this to the inline table-valued function? Now remember, the query planner is going to assume that for every customer it's going to get 100 rows out of this function, whether there's one or thousands. So let's go ahead and see how the original multi-statement version compares to the new inline table version. So the first one was a little over 250 ms, and this one runs in just over 2 seconds. So it's not quite, but almost a tenfold increase in the time it takes. It's nearly 10 times slower. That means that simply putting a little effort into this function across my whole database, if this was being used a lot of places, would give me that performance advantage for almost no additional cost to the rest of my development team. This is a huge issue in SQL Server and has been for many years. Being aware of this opportunity and finding your hardest queries that use these functions and making these changes can have a dramatic impact in what you're doing and how your application is performing.

Avoiding Parameter Sniffing in ITVFs

When it comes to functions, inline table-valued functions really are our best choice in many cases. If we have to find something that could go wrong with them, one thing is the possibility of parameter sniffing. In order to show you what parameter sniffing looks like, I want to briefly discuss what it is and a couple options you might consider. Again, there's a lot written about parameter sniffing and things that you could do in your situation if it is causing you a problem. I just need you to be aware that this is one of those things that can happen usually in the usage of inline table-valued functions because of how they're used and how they're written. So parameter sniffing is caused by SQL Server caching the plans that it generates for each of the queries we write. It does that because generating those plans takes a lot of work. And often, this is a great benefit for us. It saves SQL Server from having to do that compile and understand how to make up a plan for that query over and over again. The key, as it were, if you think about this as a key-value store, the key of that plan is the actual SQL text that you've written. Sometimes the plan it chooses won't be efficient in all cases. If I write the exact same SQL and I give it a different value, often I would expect it to find the right plan for that value. But if I parameterize that query and I pass a value in as a variable, it doesn't know that when it generates the plan, and now it has a key that doesn't have an actual value, it has a placeholder for a value. So when another query comes along and it looks like the same plan, it chooses the one that it has cached, but it might not be the best one for the new value that you've just passed in. You'll see parameter sniffing most often with any query that's parameterized. Now this is usually stored procedures and functions. However, there are times that SQL Server actually tries to parameterize queries that you're generating over and over again as well. But being aware of the data that you're querying inside of your functions can really help you focus on the potential problem when you see performance issues with functions. I will say that this is often confused in a lot of threads and conversations with the same symptoms as outdated statistics on your indexes. The query was running fine most of the time, but all of a sudden the query isn't working as well with the different values, and so sometimes that can happen if the statistics on your indexes and the table get out of date, and SQL Server doesn't actually know how to come up with a good plan. That's not what parameter sniffing is. It's simply saying I think I know what the best plan is because I have this text that matches. It just turns out that it's the wrong plan. So what are some ways, if you find that this is actually what's happening with your function usage, that you could consider fixing it? Honestly, the advice you'll see most often is query hints. There are a number of them. I've listed three on the screen. And in all honesty, OPTION (RECOMPILE) is becoming more and more of an acceptable option. When it makes sense, sometimes you have to tell a query that it should generate a new plan every time. Don't cache a plan for this query. In years past, we were much more aware of CPU usage and that generating a planned cycle. If it's a really key query and you really need to ensure that SQL Server generates the best plan, it might be worth the extra hit to compile the right plan for the right value. There are some other options that you can see here, and you'll find others, but consider at least looking at your query and seeing if a query hint could help. Another option you'll see maybe a little outdated is writing dynamic SQL. Quite honestly, this allows you to kind of write your own parameterized version, but what SQL Server ends up running is an individual query for every single value you're passing in, so the text is actually that value. This has some really unfortunate side effects. It can really bloat your plan cache, that's another problem, so you really need to examine if this is the right option for your situation. You can actually disable plan caching through trace flags or through some other options. Again, this is pretty extreme and probably not what you actually want to do for most of these circumstances, but I want you to know the option does exist. And then a lot of people will say if you have a really problematic function query or stored procedure query, is there a way that you could, in those critical situations, inline that code in the place where it's being called so that SQL Server can actually use the right plan each time?

There's a lot of ways to approach improving parameter sniffing when it's an actual problem that's affecting your production systems. I would start at the top and work your way down, particularly if updating the schema and the logic of your database isn't a good option to begin with. Usually that's where the ultimate problem is actually at and probably where you need to start attacking this problem.

Demo: Identifying Parameter Sniffing

In this demo, we're going to look at how parameter sniffing could be impacting the execution plan of your inline table-valued functions and the queries that use them. To demonstrate parameter sniffing, I'm going to create a simple inline table- valued function. It's simply going to sum all of the transaction amounts for a particular customer. Because this is an inline table-valued function, we'll actually get to see the plan that SQL Server creates for this one SELECT statement within context of the execution plan. I will create that function, and then I will do two things to prepare for this demonstration. First, I'm going to clear out all of the plan cache so that if there's something already stored for what I'm about to do, we will erase it, and we start fresh. So there's nothing from history that's impacting what we're about to see. Second, I'm going to turn on statistics so that we can start to compare how much work, how many pages SQL Server is reading in process of doing the SELECT statement. Alright, so now let's just do a very simple SELECT. I'm going to use two values, customer ID 401 and customer ID 976. So let's go ahead and run this statement for customer ID 401. Now this is a literal statement, so that exact SQL text is what the plan cache would attempt to use with that 401 as a literal. When I run this, we'll see two things. First, SQL Server had to read over 1, 000 pages of logical data to get this information out. Now that's because there's a little over 23, 000 rows for this customer in that table. Now it actually did more work. It read almost 100, 000. For some reason, it had to go through and find it, multiple iterations, but we got about 23, 000 rows, read a little over 1, 000 pages of data. Now when I run it for a different customer ID, 976, we'll see that SQL Server first did a lot less work. It only read about 500 pages, about half as much work. The reason is there are only 250 rows of information for this client in that table. So SQL Server could take that literal value and go do the right amount of work based on the indexes and what it knows about this client. But in development, one of the reasons we have these functions is we're trying to reuse text and reuse variables and be as efficient as possible. So now if I parameterize that query and I pass in a variable instead, that text is what's going to get cached. And so the plan that it generates the first time around is what it will try and cache, and if it reuses that later, we might not get the best plan. Let's see what happens again. Remember, customer ID 401 had a little over 1, 000 pages read and about 23, 000 rows of data returned. And we get about the same, a little over 1, 000 pages, and we have that same 23, 000 rows returned. Now let's see what happens for that second client. It's the same client ID. I'm using the same function. The only thing that's changed is the actual text of the SQL. Now I'm passing it in as a variable. Let's see what SQL Server does. We get the same amount, but here we see a difference. Over 1, 000 pages were read from the internal memory cache of data, not the 500 we saw earlier. And the execution plan is actually the same one for the customer ID that had 23, 000 rows. And look at that. We actually read out almost those same 100, 000 rows just to get the 250 that we know about. So this is what parameter sniffing looks like. Again, there are many ways to fix it, and I would simply encourage you to examine the multiple options, do a little bit more research a little bit beyond the scope of this specific module and course, but let me just show you what does happen with the OPTION (RECOMPILE) that you can add to queries. In years past, we were often cautioned against using this except in extreme circumstances, and that's because this will cause SQL Server to have to develop a new plan every time. So use it only when it's necessary, but as computers have gotten faster and faster, this is often a less concern than what we've had in previous years. So now when I run this with the OPTION (RECOMPILE) statement, you'll see that I still get the same plan, I still get the 1, 000 pages read out of the logical disk, and I still get the 23, 000 rows of data read for this client. I expected that. That worked as I expected. But what if I do OPTION (RECOMPILE) here? Does it reuse that plan cache, or does it generate something more specific for this client ID? And there we go. We did get the 500 pages, so we're back to doing less work for this client, and we got that same plan that we originally had with this client ID. That is one way to consider approaching parameter sniffing and fixing it in your code. Again, you have to choose what the best options are based on your data. You control it. You can determine and make this better for your system and your clients.

Summary

In this last module dealing with inline table-valued functions, we've reviewed what they are and how they're different from the other function types we've looked at. We created a simple inline table-valued function to demonstrate how that definition is different from our other multi-statement functions. I then showed you how to convert a multi-statement function into a single-statement inline function, and we discussed why this is really important to the performance of your SQL queries. And finally, we discussed parameter sniffing and how it can impact inline functions in ways that you could identify that it's happening and how you could mitigate it.