Course Overview

Course Overview

Hi everyone. My name is Jared Westover, and welcome to my course, Capturing Logic with Stored Procedures in T-SQL. I'm a SQL architect and Microsoft-certified solutions expert in data management and analytics. Most importantly, I love working with data. If you've worked in a database environment where ad hoc queries were the primary method of executing code, you know how problematic that can be. It's easy to run obsolete scripts, especially if you're working on a large team of developers. No one wants their email to be the primary source of truth when it comes to executing code. In this course, we're going to explore all of the possibilities which exist for executing your queries using stored procedures. Having knowledge of SQL Server and prior experience writing queries will be helpful as you go through the course. Some of the major topics that we will cover include the benefits of using stored procedures, incorporating input and output parameters, establishing a baseline for performance, and finally methods for optimizing your stored procedures. By the end of this course, you will have a solid foundation to start creating and optimizing stored procedures. From here, you should feel comfortable diving into other SQL courses on Overcoming Parameter Sniffing, Advanced Query Tuning, and Plan Cache Analysis. I hope you'll join me on this journey to learn stored procedures with Capturing Logic with Stored Procedures in T-SQL course, at Pluralsight.

Exploring the Benefits of Stored Procedures

Version Check

Introduction

Hello. My name is Jared Westover, and I'm recording this course for Pluralsight. This course is Capturing Logic with Stored Procedures in T-SQL. In this module, we will be covering Exploring the Benefits of Stored Procedures. I'm going to start this module out by looking at what exactly a stored procedure is and when you might use them versus inline T-SQL. I personally love using stored procedures, especially when you need to perform some sort of insert, update, or delete on one or more tables. I'll continue on by comparing a SQL view to a stored procedure. It's hard to do a direct comparison between different object types in SQL Server, since they perform different functions. It's certainly not an apples to apples, but I think a view is fairly close to a stored procedure in terms of returning data and how developers may use them. Next, we'll take a look at the primary benefits I see to using stored procedures over inline T-SQL code. These are going to be mainly related to code directly in your client application or if you're using an ORM like Entity Framework. The one I believe to be the most compelling is the maintainability of the code. I'll be going into more details in a bit. Stored procedures also offer us some nice security features over using inline T-SQL. Last, but certainly not least is performance. Performance can be a touchy subject since when you use Dynamic SQL with sp_executesql, you'll have about the same performance. We do have benefits when using stored procedures over random queries though. Finally, I'll take a look at some of the disadvantages that can come along with using stored procedures. I love them, but thought it would be unfair not to mention why developers or organizations in general may not want to move all their SQL logic to procedures. I see three primary disadvantages. The first one is an organization having a lack of SQL expertise, the second being it can be difficult to group or organize stored procedures, so they can get easily lost within the code. I'll touch on some ways in which we can try to alleviate that as we go through the course. The last one would be SQL Server is not the best at looping or iterating over data. In most languages such as C# or JavaScript, looping is extremely common. We have a lot of great information to cover in this module, so let's get started.

What Is a Stored Procedure?

Let's start off with a basic definition of a stored procedure. The easiest way is if you're already familiar with methods or functions in another language such as C# or JavaScript, they're basically a group of statements which perform some sort of action. For example, if you had an application which needed employee records created, you may have a stored procedure that creates the new employees. Keep in mind the employee information could be referenced in more than one table. Perhaps the employee has a distinct home address, good assumption, that could mean adding a row to the address table. That's where a stored procedure comes in handy. To say it again, a stored procedure is simply one or more SQL statements which are batched together. You will often hear stored procedures called sprocs. I'll even do that throughout the course. It's nothing more than an a shorthand, plus it sounds a bit cooler to stay it that way. One of the best things about sprocs is that they can be reused multiple times, and you simply need to reference the stored procedure versus all the statements inside of it. You may even need to call a sproc multiple times in a single application. You can even have multiple applications which call the same stored procedure. With stored procedures, you can perform simple or complex logic, for example, a select statement that's hooked up to a frontend, which may be returning data based on an employee's email address. You have the ability inside the sproc to use a variable similar to a .NET language, or you can do something more complex like performing calculations related to a sales order. Something I do quite often is utilize temporary objects such as temp tables or table variables to hold result sets. This can be useful if you don't want to add unnecessary locks to the underlying tables. We'll be looking at it more in the next module, but a common thing you see is that stored procedures accept one or more input parameters. In the example of a query to return employee information, the parameter would be the email address. The possibilities are nearly unlimited in terms of what you can do with sprocs. In the next slide, we're going to take a quick look at some of the common ways folks may return data back to a client in their application.

Comparing Data Fetching Options

I want to be clear that it's almost impossible to compare the various objects used in SQL Server to each other, since they perform different functionality. It would almost be like comparing a hammer to a screwdriver, even though I've tried to use a hammer on some screws in the past when things didn't quite fit; however, the end result is pretty similar in that they are likely trying to fasten a couple pieces of wood together. Let's start with SQL views. I'm a huge fan of views. I couldn't even count the number of ones I've created over the years. I especially like using them as a dataset in SQL Server Reporting Services. A view, if you're not familiar, is essentially a saved query. You can take a complex query and make it appear simpler when presented to the end user. Next we have stored procedures, which provide a bit more capability than just your standard view, but can primarily be used to return data back to the client. You're more likely to see them being used when the data needs to be manipulated before presenting the results. Finally, we have inline T-SQL. This could be SQL code that's being generated by an ORM, or code that was built dynamically, or is simply hardcoded to return some results. This one is likely the easiest to implement at first, especially if whoever is implementing this has limited T-SQL skills. You can see that there is not one that is necessarily better than the other, just like our modes of transportation, they all have the same end goal of getting you to a particular destination. One may be better for a given situation. We all know you wouldn't take a plane to pick up some groceries. Let's spend a couple of minutes reviewing the differences of using stored procedures. They each have their pros and cons like we'll see. There is no way to say one is superior than the other. It would be like saying Mario is better than Luigi. Well maybe that's true. First, like I mentioned before, sprocs can accept parameters. You can pass in one or several in of different data types. You can even pass in table types as a parameter. We'll cover those a bit later. We can also have logical statements in sprocs, for example, we can use if statements. You often see this before executing an insert or an update, the developer is checking first to see if that particular value exists before proceeding. A big one for me is that sprocs can use temporary tables. If you're working with larger datasets, temp tables can be awesome. You can even add indexes to your temp tables for better performance, But something like that has to be considered on a case by case basis. You cannot, however, reference a stored procedure in a table join. You also can't materialize the results of a stored procedure unless you're physically saving the query results to a table, which is something you might see if someone is building a table based on a group of calculations, and perhaps the data doesn't change that often. Another great thing you can do is have error handling built into the sproc. You can use RAISERROR and THROW to catch exceptions or to throw your own. Now let's contrast our comparison points with views. First, views don't accept parameters, you can, however, pass a filter in a where clause, just like if you're performing a normal select statement. You also cannot use logical statements like if exists or if not exists. You cannot use a temp table or table variable inside of a view. The best you can do is outside the view destination in your query, is place the results into a temp table. You can also create a CTE, or Common Table Expression, which sort of mimics a temp table at least from a cosmetic perspective. You can, however, reference a view in a table join. You need to be careful though about nesting views inside of views, you can run into some performance problems. The results of a view can be materialized. There is a lot that goes into a materialized view, but if the query isn't too complex, for example, you're not doing a bunch of left joins, it's pretty easy to construct one, and they can be a huge performance gain as well. A downside is that there can be no direct error handling in a view. Similar to a temp table, you would need to perform the try catch or raise error outside of the view in the query. I hope this comparison gave you a good idea of situations where you might want to use a view versus a stored procedure. For any calls to the database which are modifying data, I would always use a sproc. For reports in Power BI, Tableau, or SQL Reporting Services, I might lean towards a view, but as in a lot of situations, the answer is it depends.

Primary Stored Procedure Benefits

I've organized the primary benefits into three different categories, which I see as the main reason for using stored procedures over inline T-SQL code. There's certainly more, but these are my three favorite. Our first benefit is maintainability. For me, this is the biggest one. At the core of it, all of the SQL code is located in the same place. When we need to make a change to it, you do it at the database level, perhaps alleviating the need to redeploy the entire application. Next would be the added security benefits which come along with using sprocs. I'm definitely going to touch on these a bit more in the upcoming slides. It's really nice because you can get pretty granular in your control. Finally we have performance, which some could argue is the biggest benefit, but when compared to dynamic SQL being executed via sp_executesql, it's going to be about the same. In older versions, that would not have been the case. Compared to ad hoc queries, you'll notice a performance gain. Now let's take a look at a scenario at ABC Corp. and some changes which are underway. First, let's meet Carl. He's the new IT director at ABC Corp., and it's come to his attention that most of the T-SQL code is directly in the application, and that the same ad hoc scripts are ran over and over again during deployment and support fixes. Well at his last company, they had almost everything inside of sprocs, and he wants to see the same thing here. He's well aware of the security and maintainability, which come along with stored procedures, his biggest motivator is the consistency which stored procedures bring, reducing the likelihood of copy and paste errors. Now meet Susan. She was recently hired to work on migrating all the inline SQL code to stored procedures where it's relevant. She is in agreement with Carl that this refactoring needs to take place as soon as possible. She's a bit apprehensive about moving everything to stored procedures though. She has an appreciation of views. Before Susan gets started, we're going to take a closer look at the three primary benefits I listed earlier, which come along with utilizing stored procedures.

Demo: Creating Test Environment

In this demo, we're going to be setting up the base for our test environment, which we'll use for the rest of the course. We may be adding to it later on, but this will give us a great place to start. We're also going to ensure line numbers are turned on. Here we are in SQL Management Studio with our script for setting up the demo environment. I've already executed this query beforehand; however, I wanted to touch on some of the main elements. Just to note, this query took a few seconds to run on my system, but your time may vary depending upon how beefy your machine is. You can see on line 2 we're starting out with using our master database, and then a little bit further down, starting on line 6, we're checking to see if there is a database called ABCCompany, and if there is, we are dropping that. And then on line 14, we're recreating the database. This is so if you want to start over at any point in time, you can just come back to this initial script and execute it, and you'll have a clean slate. And if we go a little bit further down, we can see that on line 23, that we're creating a schema called Sales that's going to hold all of our sales tables. And on 26, we're creating a table called SalesPersonLevel, and most of these tables, they're going to have an Id column with an identity, we're seating a 1, 1, which is going to start out at the number 1, and it's going to increment as rows are inserted by 1, and then we're also creating PRIMARY KEY constraints on the Id column for the clustered index. And then you can see on 34, we're inserting a few values into it, president, Manager, and Staff. And then if we go down just a bit, on line 40, we're creating our SalesPerson table, and this is going to contain a lot of information related to our sales folks, for example, the FirstName, the LastName. On 44, you can see we have the Salary, 45, the ManagerId, and then we have some additional columns as well if we scroll down just a bit. On 51, I'm creating that PRIMARY KEY, 52, we're creating a FOREIGN KEY to our SalesPerson level table, and then 53, we're creating a FOREIGN KEY to our ManagerId, which is referencing on 45 the same ManagerId in this table. And if we scroll down just a bit, we can see on line 56, we're inserting some values into it, so that we have some data to play with. Let's keep going down a bit. And then starting on line 72, we're creating our SalesTerritoryStatus table, and we're then on 80, we're inserting a few statuses into it. Let's keep scrolling. And on 86, we're creating a SalesTerritory table, and then on 97, we're inserting some values into it related to the TerritoryNames, and the Group, and the StatusId. And let's keep scrolling. On 112 line number, we're creating our main table, our SalesOrder table. We can see we have, on 114, the SalesPerson, a SalesAmount, SalesDate, a SalesTerritory, and then an OrderDescription, and then if we scroll down some, you can see between 122 and 123, we're creating some FOREIGN KEY constraints back to our previous tables. Then on line 126, we're inserting some values into our SalesOrder table so that we have some data to play with, and we'll probably be adding some more data to that later on. Let's keep scrolling. And on line 142, you can see that our final thing we're doing is creating a stored procedure that we're going to be using at least in our next demo, and it's fairly basic, we're just pulling some information from the SalesOrder, and the SalesPerson, and SalesPerson level table, we're summing up the SalesAmount. It's a fairly basic stored procedure. Let's open up Object Explorer over here on the left, and if you don't see it, just go up to View in the Menu bar, and then it's F8, the first option, Object Explorer, and we can expand out our server, our Database, and we can see we have an ABCCompany database with all of our tables in it. One thing I want you to ensure is that you have your line numbers turned on. You can see mine over here on the left grid. I love referencing the line number, I do it quite often during the demos, so let's make sure we have it turned on. Up in the menu bar, if you click Tools, then go down to Options, and then over on the left, underneath of Text Editor, we'll just want to expand it, and then we have our Transact-SQL node, let's expand that guy, and we can see it's going to show up under General or the top-level node. Over there on the right, I already have it checked, but if that's not checked, go ahead and do that now, and then once you've got that checked, go ahead and click OK. To summarize this demo, essentially what we did was just set up our basic environment we're going to use, and we also added a stored procedure for the next demo, and we ensured that our line numbers are turned on so that we have a great reference point as we go through these scripts.

Demo: Minimizing Modifications

In this demo, we're going to take a look at an example of utilizing a report in SQL Server Reporting Services, and now if you saved the query in line, we would need to redeploy the report if something changes in the where clause. The same concept applies to if we have a .NET application with inline T-SQL. Again, in my experience, making changes to the database logic generally leaves a smaller footprint. I understand though that this may not apply to all situations. I'm in Visual Studio using SQL Server Data Tools, and I created a pretty simple report, which is pulling back some sales data information. If we go in and look at our dataset, we can see I am using as the Query type as Text, and this is your classic inline T-SQL code. And the query is pretty simple, it's just doing a sum on our SalesAmount, we're pulling back our SalesPerson LevelName, and then concatenating our LastName and FirstName to get our FullName. There has been a request by the business to exclude the president from our result set. Easy to do that, what we would do is just after the left join, we would add in a statement where we're saying where the LevelName is NOT IN President, and at that point, we would click OK, and then we would need to save and redeploy the report again to the report server. A different way of handling this would be with using a stored procedure. I have another report over here called SalesReportSproc, and if we look at the dataset for it, you can see for the Query type, instead of using Text, I'm using Stored Procedure, and as our GenerateSalesReport procedure. Now let's go over to SQL Server Management Studio, and what I have on the screen is just a query that is going to alter that procedure for our GenerateSalesReport. And you can see on line 13, I have the WHERE condition where I'm saying where the LevelName is not in President. The nice thing about doing it this way is that if we made the change to the stored procedure, we wouldn't have to change the report file and redeploy it. Now if you needed to add an additional column, you would need to redeploy the report, but if you're making a change that doesn't call for adding an additional column, this can be an easier way to modify the business logic. In summary, using stored procedures as the query type in applications can lead to less steps when making changes, which inevitably happens.

Exploring Security Benefits

The next benefit I'd like to explore is the added security, which comes along with using stored procedures. It's generally considered a best practice in SQL to provide the least amount of security that's needed to perform any given task. For example, if I only want to allow users to view results in a table, I wouldn't allow them to modify the data. If part of their job function was to run some stored procedure to modify the sales order information, I can provide them access to execute the stored procedure without giving them access to the underlying tables. Providing the user access to only run the sproc and not perform CRUD operations, which stands for create, update, and delete, can alleviate some security concerns. Sometimes you'll see DBAs set up specific SQL or domain accounts, which have limited access to only a handful of stored procedures, which are needed to execute in the application. This can limit the amount of damage an attacker might do if they gain entry. Whenever you execute a query through an application, the code is sent over the network if it's not wrapped in a stored procedure. This could potentially expose your table names. It can also be an excessive amount of data sent between the frontend and the backend. With a sproc, you're only sending the call to the stored procedure, which is going to be less physical text, and you're not exposing the underlying table names. The last item with regards to security is that using stored procedures versus dynamic SQL has the potential to limit SQL injection. SQL injection is where someone with bad intentions attempts to insert commands into existing SQL code. For example, if you're logging into a system, if there's limited security measures in place, and you're using dynamic SQL, there is a potential that someone could drop tables or insert malicious data. There's a fairly popular example coined by Kimberly Tripp called Little Bobby Tables. There is actually a great course on Pluralsight called Ethical Hacking: SQL Injection by Troy Hunt. It's not specific to Microsoft SQL Server, but if you want to learn more about preventing SQL injection, I would highly recommend it.

Demo: Limiting User Access

In this demo, we're going to see how we can limit a user's ability to modify data in a table. However, we'll provide them access to execute a stored procedure. I'm going to start off by creating a new login on our SQL Server. You can see on line 5, I'm creating the ReportingSales login, and I'm setting the password. Then on line 8, I'm using my ABCCompany database, and I'm creating a user that is specific to the database called ReportingSales, and I'm assigning for the LOGIN ReportingSales. Then on line 14, I'm choosing the GRANT EXECUTE on my stored procedure, GenerateSalesReport TO the ReportingSales user. My hope here is that the user reporting sales will have no access to the underlying tables, that they'll only be able to execute the stored procedure. Let's go ahead and execute this grouping of code, and let's go back to the editor. Now let's scroll down just a bit, and to do my test here, I'm going to have to change the context, so let's click the Change Connection button up in the toolbar, and let's choose SQL Server Authentication, and for my login, I have ReportingSales, and I copied my password, so I'm just going to paste that in there and choose to Connect. Let's make sure we're using our ABCCompany database. And you can see on line 21, I'm attempting to set the SalesAmount from the SalesOrder table equal to NULL. Let's see if I'm able to do this and execute that. And I'm getting The UPDATE permission was denied on the object, and that's what I want to see. And let's see on line 24 if we can even do a simple SELECT* from the SalesOrder table. And I'm getting a SELECT permission was denied, and that's what I want to see. And on line 27, I'm basically just executing the GenerateSales stored procedure. Let's see if I'm able to do that. And if I click Execute, I can, and I have my results back that I'm expecting. Now that we're done with this test, I want to go ahead and clean up after myself, so I'm going to change the connection back to my Windows Authentication and my admin account on the server. And if I scroll down just a bit, I have a few commands where I'm dropping the LOGIN and then dropping the USER. So let's go ahead and execute those. To sum this demo up, we saw how we can limit a user's ability to even perform a SELECT on a table, let alone modifying the data; however, we granted the ability to execute a stored procedure.

Exploring Performance Gains

Everyone wants their freshly-ordered packages to be delivered as fast as possible. It's human nature. If my Netflix show is slow to stream, I usually let out a sigh of disappointment. If my 2 year old has to wait 5 seconds for his snack, he's not a happy camper. One of the common benefits you hear about with using stored procedures is the performance gains that come along with them. The primary performance gain is because stored procedures are compiled once, and the execution plan will be reused over and over again, that is unless the stored procedure has changed. Now compare this to just running an inline T-SQL query. The plan will likely need to be recreated, which is going to take extra time, and you may end up suffering from plan cache below from all the extra execution plans. Kimberly Tripp has written a lot of interesting blog posts related to this. The performance gains do add up over time, especially if you're running the same sproc over and over again in your application. The performance gains kind of go away if you're comparing to parameterized dynamic SQL being executed with sp_executesql. We'll look at sp_executesql in a later module. This is where you'll see a lot of debate online, where in any modern version of SQL, you'll have equal plan reuse. The same thing goes for the most part if you're using a local variable to hold the parameter. If you're running complex ad hoc queries without local variables, any time the smallest change is made in the syntax, for example, extra white space, the execution plan will be recreated. Another area where performance gains will be touted is in regards to the reduction of network I/O between the application and the database. If you think about it, the application may need to send hundreds of lines per query versus one or two lines call to a stored procedure. It's certainly is true that this will reduce network I/O, but in most situations, this is going to be minimal unless you're sending tons of SQL scripts between the client and the database. In summary, while performance is a benefit that comes along with utilizing stored procedures, I personally just wouldn't say it's the main one.

Demo: Execution Plan Reuse

In this demo, we're going to compare executing a stored procedure with a parameter to the same inline query. We'll see how if we change the query in some way, a new plan will be generated, creating excess plans and cache. Here we are back in SQL Management Studio, and the first thing I want to do is ensure I'm using my ABCCompany database. On line 7, I have a statement CREATE OR ALTER PROCEDURE, and that's my GenerateSalesReport procedure. And on line 8, I'm using a parameter @LevelName, and I'm making that nvarchar 500 to match the actual column. Then on line 16, you can see I'm passing in that parameter against our SalesPerson level table for the LevelName column. Let's go ahead and execute this guy, so he will get altered, and go back over to my editor, and I'll scroll down just a bit. Starting on line 22, I have a command DBCC FREEPROCCACHE, and what this guy is going to do is clear out all my execution plan cache. And I have a note on 21 that says Please do not run this in production or your DBA will likely come after you. I'm going to execute him so we can get a clean slate. What I have next on line 27 is essentially the same query which is inside the stored procedure, but you can see on line 34 that I'm passing in a literal President into the LevelName. Now I'm going to execute this guy, and we can see we're getting our results back, and let's scroll down a bit, and here is the same query again, but I'm using Manager as the literal. And let's go ahead and execute this guy, and one more time, and this time I'm passing in Staff as the literal, and let's execute him. And let's scroll down just a bit, and what I have starting on line 64 is a query which is going to be pulling from our cached_plans DMV, and I'm using a CROSS APPLY to the sql_text, so I can pass in the plan_handle. On 69, it allows me to do a filter on the sql_text, and I'm using that query as the sql_text. And let's go ahead and run this guy and check and see what's in cache, and you can see I have three different row values for my plans that were created. So essentially three separate plans were created, and the three separate plans were created because I was passing in different parameter names. Now let's go back over to our editor, and if I scroll down just a bit, and I have another one starting on 74, and this guy is exactly like the last one, except on line 75, I have an additional space in there. Let's go ahead and run him. Let's go back over to the editor, and let's check our cache again and see if we have three or four, and we have four this time. So it added an additional one, and it added it because of the space that we had in there. Let's go back over to our editor. Now let's scroll down just a bit here, and what I have starting on line 97 is an execute command for my GenerateSalesReport procedure. You can see that I'm passing in President for the LevelName parameter, and I'm doing the same thing on 100, line 100, and then line 103, I'm passing in Manager and Staff. Let's go ahead and execute this guy here, and I'm getting my results back. Let's scroll down just a bit, and here I want to check my cache again to see what is showing up in there. And we can see for this guy, we just have 1 plan in cache, and you can see the Execution Counts shows as 3. Let's go back over to our editor, and just for fun, let's go ahead and execute this guy again three more times. And once again let's check our cache and see what we've got. Ah, we've got 6, so it's doing what we want here. So we can see that we're getting our plan reused. To sum it up, in this demo, we were able to see that if you're running ad hoc queries and passing in different filter values, a new plan will be generated. You hear this referred to as single use plans. With our stored procedure, one plan was created and reused, cutting down on the bloat and needing to create a plan each time.

Disadvantages of Stored Procedures

Throughout this module, we looked at a number of advantages which come from using stored procedures. I would feel a bit dishonest if I didn't at least mention some of the disadvantages. I've compiled three of the most popular ones I've seen for us to review. The first would be a company simply having a lack of T-SQL knowledge during the development process. If you've only got .NET developers who have little to no experience with SQL, then I can completely understand where you're coming from with this. However, SQL is one of the most popular languages out there, and there's so much training material available. You could argue and say well, stored procedures are hard to create, but when you break them down to their core, they are simply a grouping of statements. Sure, they can get complex, but the same logic could be applied to almost anything. The second would be SQL Server lacks optimizations for looping with while or cursors. A database system such as Oracle is more optimized for row-by-row operations, whereas SQL Server is optimized for set-based logic, where you're performing actions against a set of data versus one row at a time. On the other hand, this is where other languages excel. I understand you can't always remove while loops or cursors, but in most modern versions of SQL Server, I would say 2016 and above, there is almost always an alternative, especially when we have the ability to store output in temp tables to use later on in the batch. The last disadvantage would be the lack of organization which can come with using stored procedures. Unless you have a good system in place for naming sprocs, things can get out of hand quickly. If you start with naming your procedure Bob's procedure, you're bound for a world of pain when you accumulate several hundred from having multiple developers creating them. In the next module, I'll talk a bit more about the naming convention. Adding to this disadvantage, if your application has hundreds and hundreds of different database calls through an ORM, it may not be advantageous to actually create hundreds of stored procedures. There are certainly more disadvantages that I haven't mentioned here, but these are three valid ones I see brought up often. In summary, there are some disadvantages to solely using stored procedures. I don't know if I've ever been in an ideal situation where there is a cut and dry solution. A lot of times my answer is it's depends where you use a combination of stored procedures and other means of querying the data.

Summary

In this module, we started out by looking at what a stored procedure is, along with when you would primarily want to use them. At its simplest level, a sproc is a grouping of one or more statements, which are executed as a batch to perform some action. A simple example would be to insert some data into a table. I compared a stored procedure to a view, calling out some of the pros and cons of each, keeping in mind we can't do an exact comparison between the two. Next, we looked at the three primary benefits of stored procedures. The first being in the maintainability of the code. This includes the advantage of having all the business logic at the database layer, and often ease of making changes to the data layer versus redeploying the entire application. The second benefit we looked at was the added security that comes along with using stored procedures. This includes the capability to limit users' ability to modify data directly into a table, permitting it only via sproc, giving us some added control. The last benefit we talked about was the performance gains, which come from using sprocs versus inline T-SQL, and how our execution plan will be reused over and over again. Finally, I discussed three popular disadvantages which are often brought up when using sprocs, and how a hybrid approach may provide a better solution. Please join me in the next module, where we will be creating our first stored procedure.

Creating Our First Stored Procedure

Introduction

Hello. My name is Jared Westover, and I'm recording this course for Pluralsight. This course is Capturing Logic with Stored Procedures in T-SQL. In this module, we will be creating our first stored procedure. In our last module, we modified an existing stored procedure, but here we're going to cover all the elements which go into making a fully-functioning sproc. As Lewis Carroll said, we will begin at the beginning. I'll show you one of my favorite syntax changes, which was added in with SQL 2016, making altering and creating procedures much easier than before. In prior versions, you needed to check to see if the procedure already existed, and if so, perform an alter on it. Along the same lines, I'll demonstrate how to drop stored procedures, along with generating create scripts. Allowing SQL to generate the scripts for creating or even executing can be a big time saver, especially if you have a bunch of parameters in place. Naming any type of database object, whether it's a table, column view, or stored procedure can be a touchy subject. Most data professionals have their own unique way of doing it. This makes best practices a fairly subjective matter. I believe there are some defined ways in which you would want to name your sprocs, and we'll cover a few. Next, I'll touch on input parameters and different ways in which we can pass them into a stored procedure. I'll also touch on output parameters and how they're often used to pass values into another stored procedure. We'll also take a look at the built-in debug in SQL Server Management Studio, and how we can use it to step through our stored procedures. It's similar to debugging an application in Visual Studio. I find a lot of people have never used it before or don't even know it exists, but it can help you tremendously when troubleshooting discrepancies. The last thing I'll touch on is the concept of best practices when it comes to stored procedures. This will include some primary things you should or shouldn't do inside of them. Often you'll find conflicting information online, where one expert tells you to do one thing, and another individual tells you to do the exact opposite. In situations like these, I've found it's best to test and go with what works the best for you in your environment. Now let's get started with reviewing some T-SQL code.

Creating Our Stored Procedure

I thought I would start out with some syntax, which might be slightly familiar. What we have on the screen is about as basic as it comes for creating a sproc. If I were to execute this script in our ABC company database, I would creating a procedure which is simply returning back the FirstName and LastName from our SalesPerson table. We're not currently passing in any parameters, parameters are going to come a bit later. If I execute this twice in a row, can you guys guess what would happen on this second execute? If you guessed I would get an error message indicating that the procedure already exists, you would be right. Let's go ahead and walk through each line. First we have our actual CREATE syntax with the schema name. I'm personally a big fan of schemas. If nothing else, they help organize things almost like using folders. Having hundreds of stored procedures without a name schema would be kind of like keeping all your files on the desktop. You could also use the actual word proc, P, R, O, C, instead of spelling out procedure. I'm just not a fan of shorthand, but it's there in case typing a few extra letters is completely out of the question. On our second line, we have the AS keyword. We need this or the CREATE will fail. On our third line, we have the actual syntax that we're trying to accomplish. Here I'm simply selecting a couple of columns from the SalesPerson table. Finally on line 4, I'm ending my batch with a trusty GO statement. You also commonly see BEGIN and END statements, the BEGIN would obviously go after the AS, and the END before the GO, but in this case they're certainly not required. I added a note on here which is saying please don't start your procedure with sp_. The primary reason for that is sp_ is reserved for system stored procedures.

Exploring New Syntax and Executing

I've mentioned it before, but this new create or alter syntax is the best thing since sliced bread for me. You no longer need to check if a procedure exists before trying to create or alter it. This functionality was in my top-10 reasons I wanted to upgrade to SQL 2016. Let's walk through this one really quick. Starting on line 1, we have the new CREATE OR ALTER syntax. Guess what would happen for some reason if I attempted to execute this three times. I'm going to try and read your mind, and you're right, it would return back completed each time without an error message. On line 2, we have our same AS statement, which is needed. On line 3, I'm using a BEGIN statement, which isn't necessarily required, but it's a good way to outline the specific set of statements you're attempting to execute. You will definitely see BEGIN and END syntax when you're performing logical statements, for example, if exists. On line 4, just like before, we're executing a simple SELECT statement, then on line 5, we're using our END statement to indicate the end of our block. And then finishing off on line 6 with our GO statement. I personally always use the CREATE OR ALTER syntax when I'm working with stored procedures. The only time I wouldn't is if I'm working on pre-2016 version of SQL. In older versions, a common way you see people modifying procedures is to drop and then recreate them. This can work unless you have specific permissions assigned on a stored procedure. In that case, you would want to check and make sure it exists first, and then alter it. Now let's take a look at the standard syntax for executing stored procedures. Starting on line 1, we have your typical EXECUTE command, along with the schema name and the stored procedure name. One thing we want to make sure to include is the schema name. When you're executing these, even if it's the standard DBO, if you leave off the schema name, SQL will search in other locations before using your predefined schema. This will make your sproc run just a bit slower, plus if you're working with other people, it's just common curiosity to try and include as much information as you can. On line 2, we have our handy GO statement. On the third line, I wanted to include the shorthand for execute. I've mentioned it before, but I'm not a huge fan of shorthand, even when I'm typing out an inner join, I use the actual word inner. This will definitely execute, but unless I'm typing something incorrectly, you will always see me spell out execute.

On our last line, we simply have another GO statement. When we get to the demo section, I'll present a few different examples of executing procedures. Before that, let's spend a few minutes talking about naming stored procedures.

Naming Stored Procedures

Naming database objects in general is a touchy subject. For example, there's always a debate over whether to use singular or plural with table names. Stored procedures are really no different. You can likely perform a search in your favorite search engine and come up with hundreds of recommendations, each saying their way is superior for one reason or another. There is one thing that most people will agree on, and that is not placing a sp_ as the prefix to the name. Microsoft actually recommends not using the prefix in your naming convention. On their website, they specifically say that it's reserved for system stored procedures. Will your procedure still run if you use that as a name? Sure. If you do use sp_, SQL will first look in the master database to see if it exists in there, costing you a tiny bit of performance. Plus a SQL developer friend will talk with you about why you should never do it. A popular prefix you see is usp_, which stands for user-defined or just user stored procedure. I personally don't use this prefix. For me if I'm using a named schema, and the procedure lives there, I know it's a user-defined and not a system one. Most recommendations do call for some type of verb and noun combination. For example, if you have stored procedure which inserts customer rows, you might name it InsertCustomer, where insert is the verb and customer is the noun. Another example would be UpdateCustomer or UpdateClient, if you're simply updating the row values. I think you get the idea here. Maybe just stay away from using your name or a date as the suffix. When I see a date, I'm always wondering is that when it was created or is that when I should delete it. Another technique you see is with using the primary table as the prefix. For example, if we have a stored procedure which is going to insert rows into our SalesPerson table, I might use SalesPerson_InsertPerson. The only downside for this is that most procedures don't just affect one table, things would get pretty long if you listed all the tables affected in a larger environment. Likely the most important thing is that you define some sort of naming system that fits your specific environment and remain consistent with it. If you want to start simple, go with the verb-noun combination. Then do it until it stops working.

Demo: Creating and Executing

In this demo, we're going to be creating a stored procedure from scratch. I'll be touching on some items you may want to avoid. We're starting out fairly basic with just selecting some data from our sales schema. We are then going to execute the stored procedure. Here we are back in the SQL Management Studio, and the first thing I wanted to do is ensure I'm using my ABCCompany database. Starting on line 6, I wanted to show you guys the syntax that you would use to alter a stored procedure pre 2016. So what I'm doing is, on line 6, is if OBJECT_ID, and the OBJECT_ID is a system function, and I'm passing in the name of the procedure, the Sales.GenerateReport, and I'm also supplying P for procedure, and I'm checking to see if it's NULL, so if it doesn't exist on line 7, I'm executing the CREATE PROCEDURE, and I'm just supplying a simple SELECT 1 from it. Then on line 9, I'm performing an ALTER PROCEDURE. The reason you would do this is because if you just try to alter the procedure and it didn't exist, you would get an error message. Let's go ahead and execute this guy, and my procedure should exist now. Another method that you would use for altering a procedure is simply just deleting it or dropping it. On line 18, I'm checking to see if it exists again, and if it does, I am dropping it on line 19. Let's execute this group of code, it says it's completed. For line 26, I'm using my CREATE PROCEDURE, and I'm just giving the name, and I'm adding on 28 just a simple SELECT 1 for our demo here. Let's go ahead and execute this guy. And that completed successfully because in the prior statement, I dropped it, but if we try to execute this again, we're going to get a message indicating that there is already an object named GenerateSalesReport in the database. Let's go back over to the editor and scroll down just a bit. So that we don't have to worry about this whole ALTER OR CREATE, checking to see if the object exists, on line 35, I'm using the CREATE OR ALTER PROCEDURE. So if I select this guy, and I execute him, and if I do it again, and again, and again, it's going to continue coming back saying commands completed successfully, and that's exactly what I want to see. Let's scroll down just a bit. Starting on line 46, I'm alerting my procedure, and you can see on line 49 that I'm adding a bit more meat to it. I'm basically pulling the LastName and the FirstName in concatenation, and on line 50, I'm using the Email, and 51, I'm pulling the LevelName, and on line number 52, I'm using the Manager LastName and FirstName because you can see on 54, I'm doing a self JOIN back to the SalesPerson table based on the ManagerId. Let's go ahead and execute this, and that completed successfully, and let's scroll down just a bit. Let's spend just a minute looking at some different ways to execute the procedure. On line 63, I'm using EXECUTE and GenerateSalesReport, and you're probably wondering, something's missing here, and it is, I'm not including the schema as a prefix. Will this even work? Let's see, let's go ahead and execute, and yes, it did work, it pulled back our results, so we didn't get an error message back, but it is a little bit confusing, and you probably won't notice it very much, the performance of it, maybe if it gets called hundreds and hundreds of times and hours, something along those lines, you would, but let's scroll down just a bit, and starting on line 70, it's essentially the same command, but I am passing in my Sales schema. So this will indicate much better where this procedure lives at, if I can get my same results back from this. On line 77, you'll notice all I have is the name of the procedure, and my statement terminator, then a GO command. And now I'm wondering will this even work? Let's go ahead and try to execute it, and it did, it pulled back my sales information. Let's scroll down just a bit, and you can see starting on line 84, I have a SELECT 1 at the start, and then 85 I'm just passing in that Sales.GenerateSalesReport procedure. Let's see if this guy works. The red squiggly line is probably an indicator. And you can see I'm getting an Incorrect syntax near Sales. So, if you don't have a statement before whatever you're trying to execute the procedure, you can get away without using execute, but I don't suggest doing that, so let's try to avoid line 77 without using the execute command. Now let's scroll down just a bit and look at our last command, and you can see on line 92, I'm using the shorthand EXEC, there's nothing wrong with doing this. If you prefer using shorthand, please continue doing so. If I execute this guy, I'm going to get my same results back. In summary, we looked at a few different ways in which you can create or modify a stored procedure, including the pre-2016 syntax. We also saw how easy it is to execute one, and one method in which you may want to avoid.

Demo: Scripting Stored Procedures

In this demo, we're going to see how easy it is to script out a stored procedure. A lot of the times you would want to do this is if you're trying to implement it in another environment, or if you have some sort of version control like a Git repository to store your code. It's also a breeze to script out the execute portion of the sproc. This can come in handy if you have a lot of parameters, which are required. What I would like to show now is a couple of different ways that you can script your stored procedures. A few reasons you may want to script stored procedure, the first being if you need to apply the stored procedure to a different environment, maybe you, hopefully you have set up a development environment, perhaps a QA, and then a production environment, so that way you can take it through each of the environments. And along the same lines, you can also save it off into a Git repository that allows you to do versioning control, maybe even something like Azure DevOps. First let us check out what it's like to script out a single stored procedure. If we expand out our database ABCCompany, and then Programmability, and then underneath of Stored Procedures, and I can see I have my two different procedures here, I'm going to right-click on the GenerateSalesReport, go over to Script Stored Procedure as, and I have a few options over here, I can CREATE To, ALTER To, DROP To, I can choose a DROP and CREATE, or an EXECUTE To, and then if I CREATE To, I'm just going to put it to a New Query Window, you may want to save it to a file. So when I choose New QUERY WINDOW, SQL automatically generates the script that will create the procedure. The only downside here is that SQL Management Studio doesn't have an option that does the CREATE OR ALTER automatically, so that is something that I generally, if I'm scripting out one procedure, that'll be the first thing, for example, like on line 11, I would go in and add that CREATE OR ALTER. Let's go ahead and close this query window, and let's check out what it's like to execute the procedure, and let's go over to Script Stored Procedure as, and down to EXECUTE, and then choose New Query Window. We can see this is the syntax for executing our stored procedures. This can come in super handy if your stored procedure has a couple dozen parameters in it, especially the table type parameters. Let's go ahead and close this window. Now I'd like to show you a way if you need to generate a lot of stored procedures at the same time. If we go up to our database, ABCCompany, and do a right-click, and then go down to Task, and then choose Generate Scripts, and this Generate Scripts option just isn't for stored procedures, you can have other objects like functions, tables, views, I'm going to choose the option that says Select specific database objects, and then underneath there you see I have Tables, Stored Procedures, and Schemas. So I'm going to choose the Stored Procedures one, and if I expand out that node, I can see my stored procedures. And if I don't want to include one, I can deselect it from here, I'm going to choose both of those though, and then click Next. And here is where you can say where you want to save the actual .sql file to. I'm going to just choose, for this demo, to Save it to a new query window, and then I'm going to choose Next, and then Next. Everything looks good. And then it'll run through the wizard, and it'll pop up over here, and I can see starting on line 8, I have my CREATE PROCEDURE from my GenerateSalesReport, if I scroll down just a bit on line 25, I have my CREATE for my SelectSalesPerson. And again this is something, if I'm going to be saving this into a Git repo, I would go in here and alter these to where it's using the CREATE OR ALTER PROCEDURE because if I'm applying it to a different environment, in case the procedure already exists, I don't want to run into an error message, especially if I'm using a deployment tool that's popular with databases called DbUp. Now let's go ahead and close out our query window. In summary, in this demo, we looked at a couple of different ways that you can generate scripts from your stored procedure, just a single script for a single stored procedure, or scripting out multiple stored procedures at the same time.

Exploring Parameters

I couldn't go along without talking about parameters. At their core, they're basically tiny commands which tell your procedure what to do. You can think of your stored procedure as a program, and the parameters are the instructions it needs to follow to acquire your desired outcome. A lot of times you'll see them used in a similar manner as a predicate in a Where clause. Like other coding languages where a method may accept one or more arguments, stored procedures can accept one or more parameters. Microsoft has a pretty interesting document on their site indicating the maximum number for basically anything you can imagine, for example, the maximum number of columns in a Select statement or the maximum number of columns in an OrderBy clause. They list the maximum number of parameters to a stored procedure at 2100. I think we should be fairly safe with that number. I'm sure some folks have tried a way to exceed it. You can have both input and output parameters. The input is what you see used the most. There are certainly instances where you would want to use output parameters. Let's say, for example, you're wanting to return back a specific number of rows, which were updated, and you need to pass that value into another stored procedure. You can also use the output to determine if a stored procedure executed successfully. Most of the time though that's done with a return statement. A lot of the times you'll see developers creating local variables inside of their sprocs, which hold the input parameters, allowing them to do any additional validation, for instance, ensuring there are no single quotes in a string value. This goes back to making sure we're free from SQL injection. Another thing you want to keep in mind is naming of your parameters. If I'm using a parameter to basically pass in a predicate on a Where clause, I'll name it the same as the column. For instance, if our procedure is returning back data based on the employee address, I would likely name the parameter EmployeeAddress, or something along those lines. You do want to stay away from using super generic names like parameter1 or letters in the alphabet, really no different than avoiding bad column or table names. Now let's take a look at some T-SQL code to add in our parameters.

Demo: Adding Parameters

In this demo, we're going to look at using input parameters in our stored procedure. We are also going to look and see how to set up a default value, and then override it. Finally, we're going to take a look at having a single value returned with an output statement. Here we are back in SQL Management Studio, and the first thing I want to do is ensure I'm using my ABCCompany database, if I'm not. Starting on line 7, what I have is a script where I'm adding an additional column to our SalesPerson table called IsActive, and it's essentially there to indicate whether the SalesPerson is active or inactive, maybe they no longer work for the company, and I'm using a bit data type. And then on line 8, I'm setting the DEFAULT as equal 1, and I'm using the WITH VALUES command, so all the current rows will be populated with 1. And then if we scroll down just a bit more, on line 14, I'm updating the SalesPerson, and I'm setting that IsActive equal to 0, WHERE the employee Id is equal to 10, and that is for Tammy Smith, I'm basically saying she doesn't work for the company anymore. And let's go ahead and select these guys, and execute, I get my 1 rows affected, and let's scroll down just a bit. What I have, starting on line 19, is my CREATE OR ALTER PROCEDURE for the GenerateSalesReport. You can see on line 20, I am using an input parameter of ManagerEmail, and I'm setting the data type as nvarchar 500. Then on line 21, I'm suing another input parameter called IsActive, the bit data type, and I'm setting the default value for it equals to 1. Now if we go down to line 32, I'm using the ManagerEmail parameter to match whatever is in the SalesPerson table, and then I'm also saying the IsActive parameter is equal to the IsActive in the SalesPerson table. Now let's go ahead and execute this guy. We'll see it should say Command completed successfully, and it did. Now let's scroll down just a bit, and what I have starting on line 39 is an EXECUTE for our GenerateSalesReport procedure, you can see I'm using my schema name obviously, and I'm also explicitly using the ManagerEmail parameter, and I'm setting that equal to our Sally.Smith. Let's Execute this guy and see what results we get back. We can see it essentially brings back the Select statement that we had. Now on line 44, we're basically using the same command, but a different parameter value, but you'll notice that I'm not explicitly naming the parameter @ManagerEmail =. The only issue that I have with this is if you have a bunch of parameters like say more than like 7, 10, maybe you have way more than that, it can get really difficult to determine which one is which. You essentially have to put them in the order that they need to show up in. So if I had like 5 right here, and I have to know specifically the order, but it will execute, let's go ahead and do that, I get my 2 rows back, I always recommend explicitly naming the parameter just to avoid confusion. Now if we scroll down just a bit more, you can see starting on line 49, I'm executing the procedure again, but on line 50, I'm setting the IsActive is equal to 0, so it's overriding my default value of 1, and you'll notice in the prior executions of this, I didn't have to specify a value because it was just using the default that we set up when creating the stored procedure of 1. If I execute this guy, you can see I get my 1 row back of Tammy Smith. And let's scroll down just a bit, and you can see on line 56, I'm creating another stored procedure called ReturnSalesPersonId, and I have a parameter at 57 where I'm taking in the EmployeeEmail, and then on 58, I have an additional parameter for EmployeeId, I'm setting that as the data type of int, but you'll notice that I'm using the output keyword, and essentially what I'm doing, you can see on line 61, we're selecting into that Employee parameter, the Id from the SalesPerson table. Then on 63, we're using that input parameter for the Email. Let's go ahead and execute this guy, and let's go back over to the editor and scroll down just a bit. Now I want to demonstrate using that output parameter. So what I have, starting on line 70, I'm declaring a variable called EmployeeId of the data type END. Now on 72, I'm executing my ReturnSalesPersonId, and I'm using the input parameter for Sally.Smith, and then I'm assigning that output parameter back to the variable that I declared on 70. And then on 75, I'm simply selecting that variable. And let's go ahead and execute this guy, and you can see it's returning back 2, and that is the Id from the SalesPerson table. Let's go back over to the editor. I've mentioned it before, but a common way you would see for that output parameter is instead of selecting the EmployeeId, I would pass that into another stored procedure that was requiring that Id. In summary in this demo, we saw how to create a parameter, we also saw how to set the default value of a parameter, as well as creating an output parameter.

Exploring Debugger

I find so many times that developers don't even know Management Studio has a built-in debugger, which is similar to the one in Visual Studio. If you're not familiar with the debugger in Visual Studio, it basically lets you step inside of your code line by line while it executes. It's great because you can see what values are being placed into variables. You can also see when certain statements may or may not be returning the desired results. If I'm writing a large stored procedure where there are several variables involved, I'll more than likely run it through the debugger at some point in time. One of the cool things you can do is add break points in your code, which the execution will stop on. This is helpful if you don't want to walk through the entire thing each time you go into troubleshooting. You can also add in a condition for a break point, for example, if you only want the break point to occur, if a variable has a value of 10, you can easily add that condition in. You will see it in the upcoming demo, but you have a local window, which displays all of your current local variable values. A couple of things that you do want to keep in mind when using a debugger is that I would only recommend doing so on a test instance of your database. Microsoft actually recommends using the debugger on a non-prod instance, since it's easy to leave a transaction open and apply unnecessary locks on your tables. You will also need to have fairly high privileges as being a member of the system admin role. Some people either hate the debugger or love it. I would suggest at least trying it out to see what you think. It's not going to be something you would necessarily run for every sproc you create; however, it just might be the right tool to help you fix a nagging issue.

Demo: Debugging Code

In this demo, we're going to be using the debugger to step through a block of code that utilizes a random variable and see how easy it is to find out those values in the locals window. We're also going to see how easy it is to add a break point. I'm going to be using a couple different scripts in this demo, the first one here, you can see starting on line 4, we're going to be creating a new stored procedure called GenerateRandomMessage. On line 5, we are using an input parameter of RandomNumber of the integer data type. If we scroll down just a bit, we can see on line 9, we are declaring a local variable called PlaceHolder, also an integer data type. Then on line 11, we are setting that PlaceHolder value equal to the RandomNumber -3, and I'm just doing that because I can. Then on line 13, we're going to start some conditional statements, and we're checking to see that PlaceHolder value is les than 3, and if it is, on line 16, you can see we're going to get a printout that says You are the lucky winner. And going down just a bit, on line 20, we're going to see if the PlaceHolder is greater than 3, and PlaceHolder is less than 21, so if it's between that range, we're going to get on line 23 a message that says Not bad but not great. And then finally on line 27, we're going to see if that PlaceHolder is greater than 21 and less than 31, we're going to get a message that says that was not good. And let's scroll up to the top, and let's go ahead and execute this guy, so we can create him, go back over to the editor, and let's go over to our other query. Now for our second query, on line 4, we're declaring a variable called RandomNumber of the integer data type, and the magic really happens on line 6, we're selecting that RandomNumber from 2 functions that we're using together, and this is going to essentially give us a random number between 1 and 30, and this is a popular little set of statements that you can find online for generating random numbers. Then on line 8, we're going to EXECUTE our RandomMessage procedure, and we're going to set the input parameter equal to whatever that RandomNumber is. Let's go ahead and we'll execute this guy a few times, and see what we get back. Hey, on the first, we're a lucky winner, let's see again, Not bad but not great, and we get another Not bad but not great. Let's go back over to the editor. A procedure like this is a great opportunity to use the debugger, and to use it, I'm going to set a break point on line 8 where our EXECUTE procedure is, and then over in this gray bar, if I just click on that, you can see we get a round circle, and that is, a red round circle, and that is an indicator that we have a break point enabled. I'm going to go up and click on the Debug, and when I do that, it'll start at the very first line, and then in our Debug toolbar, I'm going to click the Continue. You can also see there are keyboard shortcuts for that. And since I have a breakpoint enabled, that's where the continue is going to stop on, and you can see down in our Locals window that I have my variable, RandomNumber, that currently has a value of 12, so that was the random number generated. And I'm going to use the step into, or F11, and at this point it's going to jump inside of my stored procedure, and I can see on line 9 that I'm setting that PlaceHolder value, and if you look down in the Locals window, you'll see that we have a new variable called PlaceHolder, currently isn't, has a val, doesn't have a value, so let's click step into, and we can see it took 3 away, so we have 9. And then on line 11, we're going to be checking that PlaceHolder is less than 3, and it is not, so it's not going to print that. And here we're checking to see if it's greater than 3, less than 21, so that is where it should fall into, and we can see it fell in there, and we're going to get a PRINT Not bad but not great, and it's going check the last one, and it's not going to apply, so we get a message back that says Not bad but not great. Let's go back over to the editor. The debugger can come in really hand, especially if you're performing a lot of logical statements. If you're using While statements or cursors, the debugger excels at troubleshooting your code. If I scroll down just a bit, I included an article from Microsoft, which goes through all the details of the debugger. If you're interested, I would give that a read. I would encourage you just to play around with it like I did here to discover all the options available.

Reviewing Best Practices

Growing up, I used to love watching Kung Fu movies. My favorite was Enter the Dragon with Bruce Lee. I have a quote on the screen, which is Adapt what is useful, reject what is useless, and add what is specifically your own. I think this is a great attitude to have when we start to talk about best practices and how you should approach adopting them. I've outlined a few different best practices that you commonly see when working with stored procedures. Some of these could apply directly to ad hoc queries as well. Let's spend a few minutes going through them. These five have the most meaning for me. I strive to incorporate them into each sproc I create. Our first best practice is using the set nocount on statement. This one I'm totally in agreement with, since it will reduce some of the network traffic which goes between the database and the client. When enabled, it will basically not return the count of rows affected or the done and proc message each time a statement completes in your procedure. This likely won't make or break your procedure, but it could help performance overall. Our second best practice is something we've talked about before, and that is not using the sp_ as the perfect to the name of your procedure. Just remember that the sp_ is reserved for system stored procedures, and you may see a tiny performance hit if you're using it. Now if you're creating a procedure which will live in the master database that's performing some type of server-level functionality, you may use the sp_, but for the most part, try to avoid it. Next, when you call the stored procedure, remember to use the schema name. You will get a tiny bit of performance out of it. I think more importantly if you use schemas heavily, it just avoids confusion when other individuals are looking at your code. There are definitely some instances where you need to use a cursor, especially performing maintenance tasks. You see the biggest performant hits when it comes to iterating over a lot of data. I would try my best to find an alternative. Since a procedure is something you're likely to use over and over again, and multiple people could be calling the same sproc, you want it to execute as fast as possible. It's easy to fall into the cursor trap, specifically if you're coming from a .NET background and are custom to using loops in JavaScript or C#. The last one is the naming of your stored procedure. We talked earlier about making sure you name them appropriately. I suggest some type of reference to the action the stored procedure will be performing. You can never go wrong with the verb-noun combination. These are really just the tip of the iceberg in terms of best practices you can follow for stored procedures. As we go through the course, I'll be throwing a few more in.

Summary

We started this module out by looking at the syntax which is required for creating a stored procedure. It's actually pretty simple at its core. We also looked at the new create or alter syntax, which makes creating and especially modifying a procedure a breeze. I touched on how easy it is to drop a procedure, you may want to make sure you're really intending to do so, since all the permissions will vanish along with it. We also saw how you can execute a sproc by using the execute command. If you prefer using shorthand of exec, I won't hold it against you. I then walked through scripting our stored procedures out with Management Studio. This comes in extra handy when we're saving them to a code repository or running them to another environment. Continuing on, we looked at how to add parameters to our stored procedures, and how they are basically instructions as to the sproc's behavior. I touched on when you may want to use output parameters, especially when you need to pass instructions from one sproc to the next. We spent a few minutes looking at the debugger in Management Studio, stepping through and analyzing each line of our code. I honestly like to think of it as a light version of Visual Studio. We saw how to add a break point and how our variable values show up in the Locals window. Finally, we looked at some common best practices. I strongly encourage you to create your own best practices by borrowing from what Microsoft recommends and from your own experiences. Remember, you don't necessarily just want to do something because someone online told you to. Please join me in the next module, where we will be Setting up Queries in Stored Procedures.

Setting up Queries in Stored Procedures

Introduction

Hello. My name is Jared Westover, and I'm recording this course for Pluralsight. This course is Capturing Logic with Stored Procedures in T-SQL. In this module, we will be Setting Up Queries in Stored Procedures. In this module, we're going to start out by discussing the two common reasons which would necessitate adding queries to a stored procedure. At its core, adding a query to a stored procedure isn't really different than adding one to an ad hoc script. Situations can call for adding several queries to your stored procedure, which build upon each other. This can make things complex pretty quickly. One element which can help reduce the complexity is using temporary objects. Those include temporary tables and table variables. Temp tables and variables are great in that they can be useful for holding intermediate result sets, which can then be built upon as you go through your stored procedure. You can then take the values placed into the temp object, and use those to modify your underlying data. Using the temp objects, especially if you're working with larger datasets, can reduce unnecessary locks on the underlying tables. I find myself often using temp tables if I'm performing several aggregations over a large number of rows. Next, we're going to spend some time looking at table-valued parameters. They've been around for a long time, introduced in SQL 2008, but are often overlooked by developers. A primary use of them is to pass in a large set of parameter values to your stored procedure. Think of a website which has a checkbox with several options for you to pick from. Without some way to pass in multiple values, our end user wouldn't be very happy with the limited website we set up. Along those lines, we will be looking at creating a user-defined table type, since a table type is required when working with table-valued parameters. Additionally, I'll touch on some of the limitations we have when it comes to table-valued parameters. Continuing on, we'll look at the built-in system procedure sp_executesql, and how you can use it to execute dynamic statements. I'll also touch on an example of when you might need to use dynamic SQL. I'll also compare sp_executesql to the execute command you commonly see out in the wild. With this information, we can draw a conclusion on which one is best to use. Finally, we'll touch on something which can be a major time saver when you're troubleshooting discrepancies with your stored procedures, especially when you have several queries in your procedure. I know it's personally saved me a lot of time over the years. I have just always called it a debug flag, but feel free to come up with your own unique name. Let's get started by spending a couple of minutes talking about the purpose of queries in stored procedures.

Building Queries in Stored Procedures

I remember being asked once by a developer what the purpose of a stored procedure is. It's not as complex as trying to come up with an answer as to the meaning of life, but it took a question and made me think. After careful pondering, I came up with a couple of purposes. The first is to simply query data. We saw on the last module how a simple select statement can be added to get some results back. Maybe the results are for a report or they feed a grid or box on a frontend application. If we remember back to a previous module, some of the reasons we do this in a stored procedure include the potential for execution plan reuse and having all the code in a central location. The second purpose would be to modify data, which could include performing inserts, updates, and deletes. You often hear these referred to as CRUD operations. The reason to perform these in a stored procedure could include all the reasons for querying, as well as having a standard for how our rows are manipulated. We'll see in one of the upcoming demos how beneficial having business logic in a sproc can be. Queries in a stored procedure can become complicated quickly, especially if we're performing multiple transformation steps. One way in which we can reduce the complexity is by adding in temporary objects to hold our intermediate result sets. There are going to be situations that call for determining information about our rows, for example, checking for a particular rank based on a sort order. Now you could do all of this with a cascade of derived tabled, but making further enhancements can be complicated, but that's where temporary objects come into play.

Reviewing Temporary Objects

Something that I seem to use on a daily basis when it comes to stored procedures are temporary objects in some way. The first one we'll be talking about are temporary tables. You might ask why use a temp table in the first place. The best answer I can give would be that they allow us to save a subset of data and then reference the data multiple times throughout our stored procedure without needing to scan the underlying table several times. An example might be if you need to perform some sort of aggregation on a set of data. Perhaps your application is calculating cost, and that cost is fairly intensive. You also have the ability to add indexes to your temp tables, hopefully speeding up performance. You do need to weight the cost of creating the index versus its performance gains. Adding an index wouldn't be something I would perform each time. Creating and dropping temp tables is better than ever starting with SQL 2016. You basically have one line of syntax, which you can check for the temp table and drop it if it exists. In the upcoming slides, we'll see how easy that is. In the next demo, I'll show you the pre-2016 way of performing this. Next, we have table variables, which are sort of like the little brother of temp tables. They're a favorite among developers because of how easy they are to create and insert values into. For smaller datasets, they require less overall resources to create than temp tables. You do want to avoid putting a larger amount of data into them. If I'm controlling the number of rows flowing into them, and it's, let's say, under 500, I'll often use a table variable over temp tables. If I can't control the amount of data which is going to be flowing into them, I'll likely use a temp table. The performance gains introduced with temp tables fundamentally relate to the statistics which are automatically created. Statistics are used by SQL to determine how to access the data in the table. More importantly, it can affect which kind of physical join operator used. If you're joining a table variable to a user table, and SQL assumes there is a low number of rows in the variable, a nested loop will likely be used, which may not be the best choice for a larger dataset. As most everything in life, temporary objects do have a few downsides, the first being they're temporary, just like a bowl of ice cream. Each time you run your procedure, which builds the result set, it will need to be recreated. You also need to make sure that your Tempdb database and files are configured properly as to the Microsoft recommendations to get the full performance out of them. Let's take a look at some of the syntax used to create these temporary objects.

Creating Temporary Objects

Let's start by examining each line of our syntax for creating a temporary table. On line 1, I'm checking to see if the temp table currently exists, and if it does, I'm dropping it. The temp table will exist as long as my session is open or until I decide it's no longer needed. If we try and create it again while it currently exists, we'll get an ugly error message saying an object named SalesOrderTemp already exists. On line 2, we have our GO statement. On line 3, I'm creating my temporary table. Notice you designate that it's a temp table with the pound sign as the prefix. You can also have global temp tables, which are accessible to other sessions, and you define those with two pound signs. On line 4, I'm declaring the data structure by adding a column called SalesAmount of decimal data type, and one called Id of int data type. Finally on line 5, we're ending with a GO statement. I have a note below stating like variables in stored procedures, you want the data types of your columns to match whatever the underlying data type to the table is. For example, I wouldn't create both of these columns as nvarchar max, I'm not saying you can't, but performance might be affected down the road. Now let's take a moment to look at the syntax for creating a table variable. Starting on line 1, we're declaring the table variable as SalesOrderVar, and setting the type as TABLE, similar to how you would declare any type of variable. On line 2, I'm defining my data structure with my SalesAmount and Id columns, identical to our temp table. On line 3, I'm finishing off with a GO statement. I do have a node at the bottom indicating that you don't really need to worry about dropping a table variable, since they are removed when the batch is completed. That's about as simple as it gets. A limitation with table variables is that we cannot add columns to them as we go along. For example, if further along in the procedure, I wanted to add a big column, I wouldn't be able to do that. It all needs to be done up front.

Demo: Adding Queries to Stored Procedures

In this demo, we are first going to expand out our ABCCompany data, making it a bit more realistic by adding some sales people and orders. Next, we're going to look at taking an existing query and placing it in a stored procedure. I have a fairly compelling use case from our SQL developer Susan for doing this. Here we are back in SQL Management Studio, and the first script I would like to show you is for expanding out our ABCCompany database. Just to note, this did take about 3 to 4 minutes to run on my system, the execution time can vary. Starting on line 5, I'm setting my NOCOUNT ON, the reason I'm doing that is so I don't get back a bunch of rows affected messages because starting on line 9, I'm inserting about 1000 rows into our SalesPerson table. So I don't have to scroll down a bunch, I am just going to click on line 5 to collapse all of those INSERT statements. Starting on line 1014, I'm INSERTING INTO my SalesOrder table the current contents, and then you can see on 1021, I'm doing that 17 times. So this'll give us about a million rows inside the table, so it's not extremely huge, but it'll be easier to show some examples with it. And let's scroll down just a bit. Now once I have all my data in there, I want to have some actual ranges for dates and numbers. You can see on line 1025 I'm declaring a StartDate, and on 1026, I'm declaring an EndDate. Then on 1029, I'm performing an UPDATE on my SalesOrder table to set the SalesDate equal to a random date in that range. And the same thing for 1032, our SalesPerson, I'm using a random range as well, and the same thing on 1033 for the SalesAmount. And then finally on line 1037, I'm going to shrink our log file just because of all of those INSERTS and UPDATES are going to kind of expand it out a bit, so this will shrink it down. Now let's go over to our second query, what we have here starting on line 5 is a script that Susan, our SQL developer, runs on a fairly regular basis for adding folks to the SalesPerson table. We have some business rules in place. On line 7, we have a note that we need to remember not to use a period between the first name and last name of the email. Line 8, we also have another one that says if we don't know their start date, just use today's date. And then our third one is if we don't know the levelId, just use 3 for staff. And then you can see on line 11, we're performing the INSERT into our SalesPerson table with some values, but we need to keep in mind those business rules that Susan has indicated above. The major downside for doing this in an ad hoc script is if Susan ever leaves, maybe there's new SQL developers that are added, unless she has shared this with everybody, nobody else is going to know what these business rules are. A better approach would be to place this in to a stored procedure, and perhaps we can accomplish some of the business logic in the procedure. Now let's, well before we scroll down, let's go ahead and use our ABCCompany database, and then we'll scroll down a bit, and take a look at the stored procedure. Starting on line 17, I'm creating the procedure called InsertSalesPerson, and you can see on 18, I'm starting to declare some input parameters, our FirstName, on 19 our LastName, our Salary, the ManagerId, and then on 22, we have the LevelId, and if you remember back to one of the business rules, if we don't know it, we just wanted to put a 3 in, so we're setting the default value to a 3. And then on 23, I'm declaring one for StartDate. And let's scroll down just a bit. And you can see starting on line 27, I'm setting the NOCOUNT ON because that was one of our best practices discussed in a previous module. Then on line 29, I'm beginning in explicit TRANSACTION. Line 31, I'm declaring a local variable called SalesPersonEmail, setting that to nvarchar 500. Then on 32, I'm declaring another local variable called SalesPersonStart, and using the date data type. Then on line 34, I'm selecting into that SalesPersonEmail variable the concatenation of our FirstName, LastName, and then also including our @ ABCCompany.com. Then on line 35, I'm setting the SalesPersonStart, I'm checking to see if it's null with the ISNULL, and if it is, I'm going to use GETDATE function, which is our current date. We could've also done something like we did with the LevelId above like setting the default value equal to GetDate. Then on line 37, I'm going to be inserting into our SalesPerson table all of our values that we've created so far from our variables. And then on 40, we're going to be committing the transaction. Let's go ahead and I'm going to create this guy, and let's go down a bit, and we can give him a go. And starting on line 48, I'm executing the procedure, and I'm passing in some parameter values. And let's go ahead and execute here and see what we get. Command completed successfully. Now let's check to see, in our SalesPerson table, we're going to check for that email for Bruce Wayne. And you can see that we inserted the record, we used the LevelId of 3, even though we didn't pass one into it, and we also have the correct email format without the period, and also the current date as of the time of this recording for our StartDate. Let's go back over to our editor. In this demo, I think we came up with a fairly compelling reason to take some business logic that was in an ad hoc script and place it into a stored procedure, one so that we can kind of automate the business logic a bit, so we don't have to remember to change our ad hoc script, and also the procedure is going to be available to all the developers versus the ad hoc script, where it may just be on somebody's My Documents folder.

Demo: Using Temporary Objects

In this demo, we're going to look at implementing temporary tables and table variables to hold some intermediate result sets. Then we're going to look at using a temp table in a stored procedure to cut down on the complexity. Here we are back in SQL Management Studio, and the first thing I want to do is ensure I'm using my ABCCompany database. Let's go back over to the editor. Starting on line 8, what I have is the syntax that you would use pre-SQL 2016 for checking to see if a temp table exists. And then on line 9, if that returns back true, you can see that I am dropping it. Let's go ahead and execute this guy. And basically the reason you would do that, that you would want to check to see if it existed first, if you just tried to drop it without it existing, you would get back an error message. And if we scroll down just a bit, starting on line 16, we have the SQL 2016 and onward syntax, one line, DROP TABLE IF EXISTS. And let's execute this guy. Let's go back over to the editor. Starting on line 23, this is some syntax we've seen before. We are creating a temp table called SalesOrder, and we have a couple of columns in it as well, so we're defining our data structure there. And let's scroll down just a bit. Starting on line 31, we're going to be inserting into our SalesOrder temp table a couple of columns, SalesAmount and Id, from the SalesOrder table, and we're setting in a date range, so essentially everything from the year 2018. Let's go ahead and execute this guy. We can see it inserted about 505, 000 rows, and let's scroll down just a bit. Starting on line 42, another common way you see individuals create temp tables is selecting into them from another table, for example, on line 43 from our SalesOrder table. So we're not defining the data structure for that SalesOrderDemo2 anywhere, and I've done this myself many, many times, so there's nothing at all wrong with it. And on 38, I have in there a DROP TABLE IF EXISTS, so essentially I'm checking and dropping it if it does exist because if it did, I would get back an error message. Let's execute that, and we can see we have our 505, 000 rows, and let's scroll down just a bit. Starting on line 51, we have our syntax for creating our table variable, and you can see just like our temp table, we're using a SalesAmount column, an Id column, then on 54, we're inserting into that SalesOrder temp table the SalesAmount and the Id from the SalesOrder, and we have our same SalesDate filter in place. Let's go ahead and execute this guy, and I have a note on line 50 that it's important because with a GO statement, essentially after this would run, that table variable would no longer exist because it only lives up until the end of the batch that's completed, and the GO statement is our batch terminator. Let's execute this guy, and we can see we have our same 505, 000 rows inserted. And if we scroll down a bit, if you remember just a few steps ago, we created a temp table by inserting into it, and I have

Will this work, starting on line 63, that we're going to try to insert a couple of columns into the table variable, and the red squiggly line is probably giving it away for us, but this will not work, you cannot create a table variable on the fly by selecting into it, you have to define it upfront. Now if we scroll down just a bit, what I have starting on line 71 is that I'm creating a procedure called ReportSalesCommission, and I have a couple of input parameters, we have a StartDate and an EndDate, and then our SalesPersonLevelId, let's scroll down just a bit more. You can see starting on line 78, I'm setting the NOCOUNT ON, 80 I'm checking to see if there is a temp table called SalesOrderData, if it exists, and if it does, I'm dropping it. And then on 82, I am creating that SalesOrderData temp table. If we scroll down just a bit more, starting on line 89, I'm inserting into that temp table that I just created the SalesPersonId, SalesAmount, and then a column called WeekNumber, and you can see I'm selecting that from a few different tables, I'm getting the SalesPerson and the SalesAmount, I'm performing a SUM on 91, so I'm aggregating that. Then on 92, I'm just getting the WeekNumber. Then on 96, you can see I'm using that StartDate and EndDate to have a filter range, and then at the end of that line, we're using that SalesPersonLevelId. Now if we scroll down just a bit, you can see on line, starting on 100, that I'm updating that temp table, and I'm setting the Commission based on a few different case conditions. Essentially on the WeekNumber, I'm checking the first one to see if it's between 1 and 12, so if it's between the first week of the year and the 12th week of the year, we're going to set the commission equal to the SalesAmount times 0.01, so we're going to get a 1% sales commission in that case. And you can see I have a few other conditions. And the very last one basically is if it's the last week of the year, they're going to have a 10% sales commission because we're going to assume that that is the hardest week of the year to make sales. And then you can see on line 107, I'm selecting the SalesPersonId, the WeekNumber, and then the Commission, and then I'm performing a ROW_NUMBER windowing function, and I'm doing a partition by the WeekNumber and ordering it by Commission descending. So this is basically going to rank our sales folks for the week number. So in the first week of the year, if we have 20 sales folks that make sales, this will tell us which one is number 1. And let's scroll down just a bit, and let's go ahead and execute to create this guy. And execute there, and if we scroll down a bit, starting on line 117, we are executing our ReportSalesCommission procedure, we're passing in a couple of parameters, our StartDate, EndDate, and then our SalesPersonLevelId. Let's go ahead and execute this, and you can see it gives us back a bit over 16, 000 rows, and we have our SalesPersonId, our WeekNumber, the Commission, and then the WeeklyRank, and I suppose at this point the business folks may want to take and put this in Excel and do some more stuff with it. You likely could make an argument and say well, we didn't really need a temp table to do this, and we could've done it with either a common table expression or a bunch of derived tables, and that's probably true, but we need to keep in mind if we have a bunch of transformation steps that we're doing here, it could be a lot easier for someone to understand if we break it into smaller steps versus having one humungous query. We can also reduce unnecessary locks on our underlying SalesOrder table by first placing it into the temp table, and then performing whatever aggregation and transformations we need on the temp table.

Exploring Table Valued Parameters

We examined in the last module how to pass in a parameter to our stored procedure. We used the SalesPerson email address as the parameter, and then applied that value as the filter in our Select statement. What if we wanted to pass in more than one value though, specifically more than one email address. That gets at the core of why to use table-valued parameters. Imagine if you're on a website and filling out information about the type of products you want returned back. Perhaps you're shopping for a new laptop and have several ideas in mind. Well if you were to select 10 different options from the filter list, those values could be going into a table-valued parameter. And when your procedure runs to return back data to the client, it can use all those values in a Where clause or a join. To use table-valued parameters, you need to start off with what's known as a type, specifically a table type. One of the cool things about a table type is that it can be used by multiple different stored procedures. I've worked on systems in the past where there was a table type defined which accepted report parameters with a simple integer column as the data type. When you create a table type, you define what sort of data structure it will contain. Pretty much the same concept of when you create a standard user table or a temporary one. User-defined types are also great when you want to limit the types of objects you want your developers to use. Generally in your application, you would programmatically insert data into the table type, and pass that as a parameter into your stored procedure. A couple of things to keep in mind with table types is that you cannot add or modify the data that's in them once they are executed in the stored procedure. We'll see in the upcoming slides that you use the read-only option when creating them in your stored procedure. SQL also doesn't keep statistics on table-valued parameters. Similar to table variables, it may not be a great idea to pass in an excessive number of rows to them. The way I've worked around this in the past is simply to take the contents of the table-valued parameter and place those into a temp table if you know the row number is going to be large, and you need statistics created. What I have on the screen now is a syntax for creating a user-defined table type, and then referencing it in a stored procedure. Let's walk through it quickly. On line 1, I'm creating a new type called SalesTableType, using Type as the suffix may be a bit redundant, but I wanted to get the point through. Then on line 2, you can see I'm declaring it as TABLE with a single column called SalesPersonEmail, and the data type is nvarchar 500, matching what's in my SalesPerson table. Then we have the GO statement on line 3. On line 4, I basically have the beginning of a CREATE OR ALTER for my procedure. specifically on line 5, I'm creating an instance of my table type and calling it SalesPersonEmail. You'll notice that I'm using the READONLY keyword at the end. I do have a note here that says once you create the table type, you can't alter it. In terms of altering, I mean you can't add or change a column, you must drop and then recreate it.

Demo: Table Valued Parameters

In this demo, we're going to take a look at creating a user-defined table type, along with defining the data structure. We will then use that table type for our table-valued parameter. I'll demonstrate how we can pass in multiple values versus a single one. Here we are back in SQL Management Studio, and I want to use my ABCCompany database to get us started. On line 8, I'm creating a type called SalesPersonId, and you can see I'm specifying AS TABLE, and the data structure is called SalesPersonId, and it's a single integer column. Let's go ahead and create this type here. And go back over to the editor, and if I scroll down just a bit, starting on line 14, I'm creating a procedure called SalesPersonDetail, and on line 15, you can see I'm using a table-valued parameter as my SalesPersonInput, and I'm specifying an instance of my SalesPersonId, and I'm using the READONLY keyword. Then starting on line 19, this is essentially just selecting some data from my SalesPerson table, but on line 23, you can see I'm performing an INNER JOIN to that SalesPersonInput table-valued parameter. And I'm just joining on the Id is equal to the Id in the SalesPerson table. So let's go ahead and execute this guy to create the procedure. Go back over to the editor, and let's go down a little bit. You can see starting on line 31, I'm declaring a variable called SalesPersonInput of that SalesPersonId table type. I'm essentially creating an instance of it. Then starting on line 34, down to line 38, I'm inserting 5 values into the variable. Starting on 34, it's 1001, and these are essentially my sales folks' Ids. Then on line 41, I'm executing the SalesPersonDetail procedure and using that table-valued parameter of SalesPersonInput. Now let's go ahead and execute this and see what we get back, and we should have rows back for five of our sales folks, and we do, you can see we're just getting our FullName, SalaryPerHour, and then the StartDate. Let's go back over to our editor and scroll down just a bit. One of the things about table types is that, I mentioned it a little bit earlier, is that is you cannot alter them. You basically have to drop and then recreate them. Now this can be problematic if you're using them in a lot of stored procedures because if we tried a drop starting on line 47, our table type, SalesPersonId, we'll get an error message back saying we can't drop it because it's being referenced by our stored procedure, SalesPersonDetail. Let me go back over to the editor. The way to work kind of around that is you would need to drop all the associated procedures with that. It's just something to keep in the back of your mind that it's better sometimes just to create multiple table types versus just having one that does everything.

Examining Sp_executesql

I find that people generally have one or two reactions when they hear dynamic SQL. The first one being awesome, we have no rules now and unlimited power. The other being no, SQL injection is coming, and the internal security team is on the way. Better prepare my resume. If you ask me, there is some truth in both of those statements, but in life, most things are hardly ever that black and white. Sp_executesql is a system-stored procedure that executes a string which is presumably a T-SQL statement. A super simple example would be select* from table name. You may be asking yourself, well, why would I need to build something so simple and execute it in this fashion? I would argue that you don't. I actually encourage developers, if at all possible, to stay clear of it. For one reason, unless it's part of a sproc, you run into the whole maintainability of the code. It can also be extremely difficult to troubleshoot. You may ask yourself well, why would I ever need to use dynamic SQL? Let me give you a good example. Let's say you have an application which allows users to define a particular sort order for items they may be viewing on a grid. At the time we create the database portion for this application, we don't know how the end user would perform the sort. Likely having a perfectly prepared query is out of the question. That's where dynamic SQL will come into play. We would dynamically build the query based on the user's selection. You may want to sort by price, no problem, the same thing could be applied to columns being returned back or even the number of rows. A great thing about using sp_executesql is that you can get plan reuse similar to a stored procedure. It's really going to depend on how complex you make the string part when you execute it. I find this mostly applies to simpler queries where you're not changing elements outside of predicates in the Where clause. You might see dynamic SQL being executed with exec, or as I call it E, X, E, C, or execute. There are really two reasons to use sp_executesql over exec. The first being it's considered to be a more secure method, and you're less likely to be a victim of SQL injection when users are passing in values. The second is along the same lines and that is sp_executesql accepts parameters, which reduce the likelihood of injections taking place. You ideally wouldn't be concatenating the command together, since it accepts parameters, it also encourages the plan to be reused in the same manner as our sprocs. Bottom line, if you're going to use dynamic SQL, I would highly recommend using sp_executesql over exec.

Demo: Using Sp_executesql

In this demo, I'm going to show you how to build a query string dynamically, then execute it first with sp_executesql, and we can determine if the execution plan is being reused. I'll also show you the pitfall with using exec. Again, we're not going to go deep into SQL injection, and I reference the course in a previous module that covers it in depth. I'll mention the course again at the end of this demo. To get us started here, let's go ahead and use our ABCCompany database. And let's go back over to the editor. Starting on line 6, I have our DBCC FREEPROCACHE command, and that's going to clear out all of our execution plans, and I have a note in here that says Please do not run this in production. Let's go ahead and run this guy to get it cleared out. And starting on line 11, what I'm doing is declaring a variable that's called @SqlCmd, and that's going to be nvarchar 1000. On line 12, I'm declaring another variable that's the SalesPersonEmailInput, and I'm setting that to be nvarchar 500. Then on line 13, I am defining, I'm setting that SalesPersonEmailInput equal to be an individual from our SalesPerson table. Then starting on line 15, I'm setting my SQL command variable equal to a string, but you can see on 21 that I'm able to pass in where it says WHERE Email = @SalesPersonEmail, so I'm able to use a parameter in this string. And if we look on line 23, I am executing the sp_executesql, and I can pass in that SqlCmd, and I define that SalesPersonEmail parameter, and then I'm setting that SalesPersonEmail parameter equal to the input that I had from line number 13. Now let's go ahead and execute this guy, and we'll do it a few different times, 1, 2, 3 times, and if I scroll down a bit, and this is the exact same syntax starting on line 29, except I'm using a different email address. Let's go ahead and execute him, we'll do it a few different times, three times. Let's scroll down just a bit. One of the benefits I mentioned before about using sp_executesql is that you get the plan reuse just like you would in a stored procedure. Let's go ahead, starting on line 47, we've seen this query before, but this is checking our cached plans, and I have a WHERE on line 53, we're looking for something that is remotely familiar in our query, and let's do an execute. And you can see the second row is for our prepared statement, and you can see that we were able to reuse the execution plan 6 different times on it, so we have plan reuse. And let's go back over to the editor. Let's scroll down just a bit. What I have starting on line 59, similar to before, I am declaring a SQL command, nvarchar 1000 variable, and a SalesPersonEmail variable, and I'm setting that SalesPersonEmail equal to an individual from our SalesPerson table. Then on 63, I'm setting that SqlCmd equal to a string, and I'm just concatenating in that SalesPersonEmail, so it's not actually, it's not going to be used as a parameter because you can't really have parameters when you're using the execute command. So it's just concatenating that together. Then on line 71, I'm executing that SqlCmd. Let's go ahead and run this guy, and we'll do it a few different times. Let's do 3 times, and then starting on line 77, is the exact same situation I had before, but I'm using a different email address. And let's execute this guy a few different times. And let's check our cache again and see what we've got going on. And you can see on rows 1 and 2 is our ad hoc statement. It was able to reuse it for the three times we put it in, but it wasn't able to use the parameter like it was for our sp_executesql, so it's essentially, if we would run it again with another email address, we would get another execution plan from it. Let's go back over to our editor and scroll down just a bit. I wanted to include a Microsoft article on SQL injection, and I have the URL on line 107, and then on line 109, there's a Pluralsight course called Ethical Hacking SQL Injection, and that's by Troy Hunt, and it has a lot of great information that you can use to prevent injection, I definitely recommend you checking it out.

Implementing a Debug Flag

We looked at the debugger in the last module, and it's a great tool for seeing which literal values are being used related to variables. It also excels at showing us which logical statements are being executed. I think it falls a bit short when we are needing to see intermediate result sets. It's also not recommended to run in a production environment. If we're working with a dataset in our development space that's vastly different than production, this can leave us making bad assumptions. Ideally we need something that gives us insight as to what's happening related to the statements which are being executed in our sproc. If you're creating stored procedures which use temp tables or table variables heavily, it can be a real pain to troubleshoot. Sometimes you just need to know what results you're getting back after each statement is being executed. You likely need to know this before the procedure is complete. I honestly can't even remember where I saw the idea of using a testing or what's sometimes called a debugging flag or parameter. I can say it's helped me troubleshoot discrepancies more times than I can count. There's also more than one way to go about adding the debug flag. For example, if the debug flag is enabled, you can select from each temp table as you go along in the sproc. I've also seen it to where you perform all the selects at the end. I wouldn't say there's a right or wrong way to do this. Pick whichever works for you in your environment. What's really cool is that you can leave it in all the way to production, since it's only executed when we give the command. Keep in mind running it in production is going to be dependent upon what you're actually doing in the procedure. I totally admit this does take some time to set up at the get go, but it can absolutely be worth it in the long run. You also don't need to do this for every stored procedure that's created. Most of the time I would only implement on sprocs which build a result set over multiple statements.

Demo: Adding a Debug Flag

In this demo, we're going to look at how we can go about adding a debug flag to our stored procedure. I think you'll be surprised at how simple it is. The utility of it outweighs the setup cost in my opinion. We are then going to execute our procedure with the debug flag enabled and check out the results. We're back in Management Studio, and I'm going to use my ABCCompany database to get us started. On line 7, I am altering a procedure that we've used a couple of demos back, and that is for our sales commission report. You'll notice on line 11, I've added an additional input parameter called Debug, and the data type on it is a bit, and I'm setting it as default equal to 0, so that way unless I specify it equal to 1, it will not be a 1. Let's scroll down just a bit. And you can see on line 26, that's where I was performing the insert into our SalesOrderData temp table. If I scroll down just a bit more, starting on line 36, I have an if condition, and I'm checking that Debug variable, and if it's set to 1, if that is true, then line 38 will execute, and that's going to give me a select* for my SalesOrderData temp table, so I'm going to be able to see what my result sets are at that point. And if I scroll down a little bit more, I am on line 42 transforming the data some more, and on line 49, I have another condition that I'm checking to see if the Debug is equal to 1, and if it is, it's going to do another Select on the SalesOrderData. If I scroll down a bit, I added just on line 54 a TOP 10 of that table. Let's go ahead and execute this guy, get him created or modified, and let's scroll down just a bit. Starting on line 66, I'm simply just executing that ReportSalesCommission procedure, you can see I have my StartDate being 1/1/2018, my EndDate being 1/31/2018, so it's going to pull everything for January of 2018, I'm still passing in my SalesPersonLevelId, but I'm also passing in that optional parameter, the Debug, and I'm setting that equal to 1. So this is going to execute those Select statements. Let's go ahead and execute this guy, and you can see I get back 3 different Select statements, my first one, my second one, and then my final results, which is simply going to have 10 rows in it. So if there's something that doesn't equal up in the final result set, I can go back and look at the other two result sets and kind of figure out the key to the puzzle. Let's go ahead and go back to the editor. This was a really simple implementation, but this can be so effective in troubleshooting data discrepancies inside your stored procedures. So I invite you to come up with your own ways of implementing it.

Summary

In this module, we looked at adding different types of queries to our stored procedures. This includes Select statements, as well as statements which perform updates and inserts. We saw that adding a query to a stored procedure isn't really different than an ad hoc script. I demonstrated how we can create temporary objects to hold intermediate result sets with temp tables and table variables. We also touched on a few differences between the two, and when you would want to use one over the other. Next, we looked at using table-valued parameters to pass multiple values into our stored procedures. I would highly recommend using table-valued parameters over trying to store the results in a varchar max or XML field which needs to be parsed out. Please remember that you need to create a user-defined table type. I mentioned a couple of the limitations that come along with table-valued parameters, including the lack of statistics. We then looked at executing dynamic SQL with sp_executesql. I touched on the reasons you may want to use dynamic SQL in the first place. For the most part, I personally try to avoid it, but it can be invaluable for specific situations. I also mentioned it's considered a best practice to use sp_executesql over exec. I would say the primary reason is due to the risk of SQL injection. Finally, we looked at implementing a debug flag in our stored procedure, and how this can be helpful for us to review intermediate result sets. There is definitely some setup time to enabling this, and I wouldn't necessarily do it for every stored procedure. Please join me in the next module, where we will be Optimizing Stored Procedure Performance.

Optimizing Stored Procedure Performance

Introduction

Hello. My name is Jared Westover, and I'm recording this course for Pluralsight. This course is Capturing Logic with Stored Procedures in T-SQL. In this module, you'll be Optimizing Stored Procedure Performance. In this module, we're going to start out by ensuring you have some sort of method for establishing a baseline for the performance of your stored procedures. If you don't know how they're currently performing, how can you expect to improve them? We will do this by exploring a dynamic management view, which provides a lot of great information around the performance of your sprocs. The data is aggregated up, so individual executions are not captured; however, this view gives you a nice window into the current behavior. When you're dealing with procedures and ad hoc queries performing poorly, the cause often comes down to returning or updating a lot of pages. SQL Server has a helpful built-in tool, which allows you to see how many pages are being returned back from either disk or cache. From there, you can try and reduce them. It's also important to understand that we need to perform extensive testing of our stored procedures before releasing them to production. This includes more than anything using multiple values for our parameters. Based on the selectivity of the data, you can get vastly different results back. I also want to talk about a couple of claims you commonly see online related to sproc performance, which may not actually hold up to testing. The first being that you should avoid temporary tables in your sprocs, since they will eliminate plan reuse. In the last module, I talked about how great temp tables can be for reducing complexity and alleviating excessive table locks. The second is something you commonly see in sprocs, and that is checking for the existence of some value before updating or inserting rows. The common suggestion is to use select 1 versus select *, since the select * will return back more data. Do you think this is accurate? The simplest way you can answer that is to test it out. Finally, we're going to touch on a topic which is every SQL developer's favorite interview question, and that is about parameter sniffing. I'm going to talk about what exactly it is, spoiler, it's expected functionality in SQL Server. If you're not familiar with parameter sniffing, at first glance, you may find references online painting it in a pretty bad light. It can definitely have negative effects on your stored procedure performance in certain situations. But it is in no way something which should be avoided overall. Now if it does affect your stored procedures in a negative way, we need to have a plan in place for overcoming those situations. I'll be touching on a few ways in which I commonly do that. I'll also mention a Pluralsight course, which goes into details about parameter sniffing if you're interested in learning more. Some of the techniques we're going to look at could also be applied to ad hoc statements. But since stored procedures are likely executed repeatedly, the effects are going to be more apparent.

Creating a Baseline

I have a great quote from Zig Ziglar that speaks well to how you should approach optimizing the performance of sprocs. I'll admit I've been guilty of not having objective performance goals of my sprocs in the past, but you should never stop trying to improve. Something that I dread hearing is hey, can you take a look at my procedure and make it faster? Believe me, I would like nothing more than to do that. I likely counter with the question, are they slower than what they used to be? How do we know our stored procedures are slow? It's almost like anything in life, you need a baseline. A simple example, if you lift weights and say I want to add 50 pounds to my bench press, that's great, but unless you have a starting place, it's going to be difficult to measure your progress over time, and if that's even a feasible goal. Unfortunately in the real world, you often know your procedures are slow because users start complaining that the application or reports are running slowly. Now this may be your procedure's fault or it may not be. If you don't routinely rebuild or reorganize your indexes, that could cause the procedure to run slowly, but that's not something you could easily solve by tweaking your sproc. Another way in which we may find out that they're not performing well is that the DBA visits your desk or sends you a nasty email kindly copying your boss saying hey, this sproc is causing blocking, we need this fixed as soon as possible. A way to alleviate some of this headache before your sprocs are put into production is to ensure you're performing adequate testing using a wide variety of parameter values. SQL Server does contain a dynamic management view, which provides us some insight into how our procedures are performing. Unfortunately it doesn't provide us details for every execution, but at least contains information like the max and last duration, and the number of pages being read. We'll take a look at it in the upcoming demo. You can also utilize an awesome built-in tool to gather some information related to the number of pages being read by SQL. In my experience, reducing the number of pages you're reading back from disk or cache is one of the most impactful ways to increase performance. These do need to be captured when you're in the development phase for your procedure, so that you can make tweaks before releasing to production. We want to do everything in our control to ensure our procedures perform at their best out of the gate.

Demo: Gathering Baseline Stats

In this demo, we're going to take a look at a dynamic management view, and how you can use it to determine the performance of your procedures. Next, we're going to check out a method for gathering some stats by using a command, which allows you to return the number of pages which are being read from disk or cache in your stored procedures. Here we are back in SQL Management Studio, and the first thing I want to do is ensure I'm using my ABCCompany database. Starting on line 9, I have a DBCC command for FREEPROCCACHE, and that's essentially going to clear out everything from our procedure cache, and I do have a note on line 7 that says Please do not run this in production. Let's go ahead and execute that, and scroll down just a bit. Starting on line 16, I'm executing a procedure we've seen before, our ReportSalesCommission, and I'm passing in a StartDate of 1/1/2018, and then an EndDate of 1/31/2018 with the SalesPersonLevelId of 3, so this'll give me everything for January of 2018. So let's go ahead and we'll execute this guy three different times. And let's go back over to the editor and scroll down just a bit. Starting on line 26, I have a query which is pulling from our DMV dm_exec_procedure_stats, and this is the DMV which is going to have a lot of great information about our procedure performance. You can see on line 36, I have a WHERE condition, and I'm using the object_id, and then our procedure name, ReportSalesCommission. So you could pull all your procedures back if you wanted to. And let's go ahead and I'm going go to execute this guy, and I do have a note on line 25, which is saying that procedures which are in process, essentially executions which are currently going will not show up until they complete. So let's go ahead and execute this guy and see what we get back. And our first column, our Cached column is when the execution plan was placed in the cache. And then we have our Execution Count of 3, the last time it was executed. The Last Logical Read, so we have around 72, 000, the Max Logical Reads, which is a bit over the 72, 000, Last Logical Writes, keeping in mind that we are doing some stuff with TempDB here, and then our Last Elapsed Time, 290 ms, our Max Elapsed Time is 329 ms, and then the Min Time is 275 ms. Now, one thing I would probably do here is if I'm troubleshooting a procedure is I would likely take and just copy and paste this information, and place it into Excel or Notepad++ with a listing of the parameter values I used, just so I could keep track. Let's go back over to the editor, and let's scroll down just a bit. Starting on line 43, I'm executing the same stored procedure, but here I'm passing in a wider date range, essentially for the entirety of 2018, a lot of times developers may only use a limited amount of data when they're testing out sprocs or reports, but a lot of the times the business will want to come in and see everything for an entire year or multiple years together, so it's best to try to use a wide variety of parameters. And let's go ahead and execute this guy three different times. And go back over to the editor and scroll down a bit. Now let's go ahead and execute our query to get our procedure results, and we can see we now have an Execution Count of 6, and our Last Logical Reads went up to 89, 000, and I expect that because we're pulling back more data, so we're going to have more data pages. And our Max is a bit over, it's right at that 89, 000, the Writes is 87, and then so on for our time has increase as I would expect it to since we're pulling in more data. Let's go back over to our editor, and let's scroll down just a bit. A great tool we can use to see the actual page reads while we're running the individual execution of the procedure is statistics io. You can see on line 74, I'm setting it ON, and then on line 82, I'm turning it OFF. And I also want to ensure I have my actual execution plan enabled, it's either Ctrl+M or up in the menu bar, you can click on the icon. Let's go ahead and run this, and once it returns, we can go over to our messages. In here we're going to get a listing of all of our tables that are involved in the query of how many scans were done, how many logical reads, and logical reads is a really important one to pay attention to, and I can just see here the table, which is pulling back the most is certainly our SalesOrder table, so we're pulling back the 70, 000 pages from it. And then let's go over to our Execution plan, and you can see right off the bat our first query that's taking up 100% of the batch, and that is also showing us that we have a missing, it's giving us a Missing Index hint, and it's giving us, hey, go create an index on this table, include these columns in it, and that will likely speed up the performance of the query. And the index it's suggesting is essentially on the SalesDate, and it certainly makes sense. Now let's go back over to the editor, and we'll scroll down just a bit. Starting on line 90, I have the syntax for creating my non-clustered index on that SalesPerson, and then SalesDate, and including the amount in it. I'm going to go ahead and execute that, and it should just take a few seconds. And if we go scroll down just a bit more, let's go ahead and execute the procedure again. First we'll check the messages to get our page count, and we can see the Scan count went up, but the logical reads went down dramatically, we're only at a bit over 2000. And if we go over to our Execution plan, we can see we're no longer getting that hint that says hey, add this index. And the reason our scan count went up is because we're having a different physical join operator that the optimizer is using. Now let's scroll down just a bit more. On line 115, I have a URL to a Microsoft article on the DMV dm-execute-procedure-stats, there's a lot more columns that you can use to pull back information on it, so I highly recommend that you spend a few minutes reviewing that.

Busting Performance Myths

I used to love watching the TV Show Myth Busters. My favorite episodes were when they would present some sort of story from a show like MacGyver or Star Trek, and through practical experiments, they would determine if something was fact or fiction. We can do the same thing with SQL Server by putting a few claims that I once believed to the test. In the last module, I talked about how temp tables can be helpful for breaking down complex logic into smaller steps and alleviating excessive locks. However, a common best practice you hear is that you should limit temp table usage in stored procedures, since they'll likely cause the plan to not be reused. I remember being told this several years ago. For a long time, I didn't even question it until someone suggested using a temp table in the procedure I was developing, and I said something to the effect of don't those affect execution plan reuse? I honestly can't find a source where someone has demonstrated this. Perhaps on a prehistoric version of SQL Server or even a different database engine all together, it may have applied. Needless to say, we are going to put this claim to the test in our next demo. One area in which temp tables can reduce performance is with a lack of data being sorted. You can reduce this in certain situations by adding indexes to your temp tables. You do need to test and make sure creating the index isn't taking up as much resource as you would be gaining down the road by having the data sorted. A common technique you see in sprocs is to check the existence of a value before performing an insert or an update. Typically this is accomplished using an if exists or if not exists. Once inside the logic block, you would perform a select 1 or a select * on the table. Several sources say to use select 1, since select * is often frowned upon. Do you think you'll get better performance using select 1 versus select *? The answer may surprise you. Something I wanted to touch on briefly relates to the idea that breaking up one long stored procedure into multiple ones can increase performance. As a general rule of thumb, I've never experienced any performance gains from doing this. Again, there could be some edge case out there where this suggestion holds true. It's just not with anything I've worked on. There are reasons to have one sproc call another, but performance is not one of them. This is another situation where you could simply put the claim to the test.

Demo: Temp Table Myth

In this demo, we're going to test out the claim that adding a temporary table to our stored procedure will stop the execution plan from being reused. Being able to test this out will be fairly simple as you'll see. Here we are back in the SQL Management Studio, and I want to ensure I'm using my ABCCompany database. Starting on line 8 is a command we've seen a few times before, and that is clearing out my plan cache. I'm going to go ahead and execute this, remember Please don't run that in production. If I scroll down just a bit is a procedure starting on line 15 that we've seen a couple of times before, our ReportSalesComission. I just wanted to pull it up to reiterate the fact that we are using temp tables. And on line 25, you can see I am dropping the temp table if it exists. Scroll down just a bit more. On line 27, I am creating the temp table. On line 34, I'm inserting into the temp table some aggregated data from our other tables, and if I keep on scrolling down, I'm performing some transformation on line 50. And then finally if we scroll down some more, on line 62, I'm selecting some values from it. Well let's go ahead and execute this guy, ensure we're all on the same page, and then let's scroll down just a bit. What we're wanting to test here is if we execute this procedure, if the myth holds true, then we should not see the Execution plan being reused on multiple executions. So you can see starting on line 76 that I'm executing the procedure with a couple of different parameter values, and let's go ahead and we'll execute this guy, we'll say three different times. And we'll go back over to our editor, and starting on line 85 is a query we have used before that is pulling from our DMV dm_exec_cached_plans, and we're performing a CROSS APPLY on line 90 to our SQL text, and we're just checking to see where we have that temp table. And let's go ahead and run this guy and see what shows up. And we can see we have an Execution Count of 3. We are reusing that execution plan even though we're using a temp table. Let's go back over to our editor, and we'll scroll down just a bit. And to assure that we're not having any trickery here, let's go ahead and use a couple of different parameter values. And we'll execute it 3 more times, 2, 3, and we're going to check that cache again to see how many execution counts we have, and you can see that we have 6 here. So the temp table is no way not causing the execution plan to be reused. And let's scroll down just a bit. Now one thing which will definitely cause the execution plan to not be reused is if we perform an alteration on it. So starting on line 120, I am doing exactly that. If we scroll down just a bit, really the only change I've made here is down on line 167 where I'm saying SELECT TOP 5 versus TOP 10. Let's go ahead and execute this guy, so we can perform our alter, and let's scroll down just a bit. And let's run our procedure a few times here, let's say 1, 2, 3, go back over to our editor, and we're going to check our plan cache now, and what I expect to see is 3 versus additions to the previous one. And let's execute that, and that's exactly what we get. We get an Execution Count of 3. Let's go back over to the editor. To say again, this is likely a situation where this may have occurred on an older version of SQL, or perhaps it was on a different database engine all together, but it's so easy to just put these little claims to the test to see if they pan out.

Demo: Select 1 Versus Select *

In this demo, we're going to take a look at an example of checking a value's existence before issuing the update or insert statement. I want you to help me determine if there's any performance gain by using a select 1 versus a select *. Here we are back in SQL Management Studio, and the first thing I want to do is use my ABCCompany database. Starting on line 7, I have a new stored procedure called InsertUpdateSalesPerson, and basically what we're doing here, it looks similar to a previous stored procedure we worked on, but we're taking in some input parameters, and if we scroll down just a bit, the main difference is on line 27, I'm saying IF NOT EXISTS, and I'm performing a SELECT 1 FROM SalesPerson, where the Email is equal to whatever SalesPerson we passed in. So if that SalesPerson email doesn't exist, on line 31, I'm performing an INSERT. And if we scroll down just a bit, on line 36, I have an ELSE, and so if the SalesPerson does exist, we're going to update their salary. You can see that on line 40. And then on 42, we're printing out a message saying Salary updated. Let's go and run this guy so we can get him created, and let's go back over to the editor. And I'm simply just going to execute this on line 56, I'm going to pass in a few parameters that I know where the email is already going to exist, and that's for our Bruce Wayne employee. Let's execute this, and we get a message back saying Salary Updated. And let's scroll down just a bit. Now I used a SELECT 1 above when I wanted to check for the existence of the SalesPerson, and you can see on line 69, and I usually do select one just out of habit because it's the one I always have done, so I'm just used to doing it, but what we have here on 69 is we're doing IF EXISTS SELECT 1 FROM SalesPerson, and we're passing in that Email address. We're going to PRINT out Bruce is already here. You'll notice on line 67, I'm setting my STATISTICS IO ON, and then I'm turning it OFF on 74. If we scroll down just a bit, our second example here is the exact same query essentially, but instead of SELECT 1 on line 82, we're performing a SELECT *. And we're going to run both of these guys together, and we want to ensure we have our actual execution plan. So if that is not enabled, make sure to enable that. And let's execute this. And we can see our statistics io data is telling us on both of these queries that we're having 20 logical reads and 1 scan, so those look exactly the same. And if we go over to our Execution plan, you can see these two queries together, that they're each 50% of the batch, and we're essentially doing the exact same plan for both of them. Let's go back over to the editor, and we'll scroll down just a bit. Now one thing I want to do quickly is to just test another table. Since our SalesPerson table is pretty small, we're going to use our SalesOrder table this time. On 97, we're performing the SELECT 1, and then on line 110, if we scroll down just a bit, we're performing the SELECT *. Let's go ahead, and we're going to execute this, and our execution plan should still be on. Let's execute, we can see we're getting back the exact same logical read count of 3, and if we go over to our Execution plan, they look exactly the same. Let's go back over to the editor. It doesn't matter if we do a SELECT * or a SELECT 1 in these situations. SQL still needs to read the pages for the data. So this is definitely something that does not hold up under testing.

Exploring Parameter Sniffing

You already know one of the ways in which SQL Server optimizes performance by creating an execution plan once and then reusing it multiple times. If we run a stored procedure for the first time, the plan will be cached, and the parameter values will also be cached. This is expected and desired behavior from SQL Server because it allows you to bypass recreating the execution plan, which can be an expensive operation. To recap, if you pass in a particular SalesPersonId to use in one of your Where clauses, the plan is built around that particular Id. Let me give you an example of how SQL creates the plan using this one-size-fits-all mindset. Imagine for a minute you're helping a friend move from their parent's house to a studio apartment. You arrive at their parents, and they only have three boxes. No big deal, you put the boxes in your Prius and deliver the packages in one trip. What if you go to help your friend move again a few years later, but this time they have accumulated loads of furniture and a few dozen boxes. Well that small Prius wouldn't be the best choice of transportation now. This is the same concept which can cause issues with SQL sniffing out and reusing the original parameter values that were passed in. What if in our SalesOrder table there was only two rows for the original SalesPerson? For another, there could be 100, 000. Do you think that same plan is going to be a good fit? There are ways in which we can go about correcting this behavior when you find it negatively affecting performance. Hopefully you're including these kinds of scenarios in your test plans. You have the option to recompile the stored procedure before each execution. This is kind of like the sledge hammer approach, but it's often the most recommended you'll see. We can also recompile the individual statements in the stored procedure. Keep in mind you're stored procedure can be made up of one or more SQL statements. In the upcoming demo, we'll take a look at a few of these. There is a great Pluralsight course called Identifying and Fixing Performance Issues Caused by Parameter Sniffing. Gail Shaw is the author and goes into everything there is to know about parameter sniffing. If you would like a deep dive, I would highly recommend checking it out.

Demo: Parameter Sniffing

In this demo, we're going to take a look at an example of when the cache parameter values negatively affect future executions. We'll be able to see this by checking out the properties of our execution plan. Once we have identified that parameter sniffing is affecting us in a negative way, we'll explore a few different ways to work around it, including different ways to recompile the procedure. Here we are back in SQL Management Studio, and the first thing I want to do is use my ABCCompany database. Let's go back over to the editor. Starting on line 8, we have a command we've seen a few times before, and that's to our DBCC FREEPROCCACHE, and that is going to clear out our plan cache. Let's go ahead and execute that. Scroll down just a bit. Starting on line 14, I have a CREATE OR ALTER for a PROCEDURE we've seen before, the GenerateSalesReport. I've made some modifications to it. On line 15, I have a StartDate parameter, and then on 16 I have an EndDate parameter. This query is essentially just pulling from our SalesOrder, SalesPerson, and SalesPersonLevel table, and we're performing a SUM on the SalesAmount. And you can see on line 26, I'm using my parameter values to get a date range from a StartDate to an EndDate. Let's go ahead and execute this guy, so we can make the modification, and let's scroll down just a bit. Starting on line 35, I'm executing my procedure, and I'm using a StartDate of 01/01/2018 and an EndDate of 01/31/2018, so it's going to pull everything for January. Now let's execute this, and I want to ensure I also turn on my actual execution plan, and let's click Execute. It returns super fast, 999 rows in less than a second. Let's look at our execution plan, and we can see if we scroll a bit over to the right, most of the cost is coming from the Index Seek that we have on our SalesOrder table, and it's to be expected, it's the largest table by far in our query. And let's go back over to our editor. And let's scroll down just a bit. On line 45, I'm executing the procedure again, this time though for the date range, you can see I'm using a quite larger time span, it's 01/01/2017 to 12/31/2019. Let's execute this guy and see what we get. And I'm going to go ahead and pause the video. You can see that the query took right at 28 seconds to complete, and it's returning back 31, 000 rows, a bit slower than what I would expect for that row count. Let's go over and look at our execution plan, and see what we have. Let's scroll a little bit over to the right. One thing that I noticed kind of like right off the bat is this is the exact same execution plan that we used for the previous execution. Well, that's to be expected because we're getting plan reuse. But before, we were only using such a tiny date range, and now we're using a larger one. We can still see that we're doing an Index Seek on that non-clustered index for our SalesOrder table. Let's pull up the Properties window, and I'm going to pin it here so that we have a good view point on it. Let me point out the actual number of rows that we're seeking on is 1.3 million, so that's quite a few. And if we look down just a little bit to the estimated number of rows, it's at 43, that's a huge difference. And let's also scroll a bit over to our left, and we can see, if we click our SELECT statement, and back at our Properties windows, we have an option for Parameter List. So we can expand that node. And we have our StartDate and EndDate, and let's look at our StartDate first. And we have areas that are called Parameter Compiled Value, and this is of course our 01/01/2018 because that was what it was originally done as. And then our runtime value is 01/01/2017, and we have the same thing basically for our EndDate, where our Compiled Value and our Runtime Value are vastly different. But SQL is still using the same execution plan, and this is where you run into issues with parameter sniffing. Let's go back over to the editor, and I'm going to hide my Properties window, and let's scroll down just a bit. Starting on line 54, I have another execute. This time, and this is my, one I'm using my larger time span on, but you'll notice on line 56, I have WITH RECOMPILE. What this will do is recompile the stored procedure at runtime. Let's go ahead and execute this guy, see what we get. Oh wow, so that ran much faster, 31, 000 rows in about a second. If we look at the execution plan, we even get an index hint, but the important thing to notice is that we're doing an index scan now versus our seek that we were doing before, and we're using hash match. Let's go back over to our editor. So that is one way of doing it. And let's scroll down just a bit. Another option that we have is to recompile every time the procedure runs, the entire stored procedure. You can see on line 67, I have an option that says WITH RECOMPILE. Let's go ahead and execute this guy to alter him. We can also, if we scroll down just a bit, we can also just recompile individual statements. In this stored procedure, it doesn't really matter because it's only one statement, but if we have multiple statements and we're finding one of the statements is the culprit, we can use what I have on 99 at the statement level, OPTION RECOMPILE. So I'm not going to choose that one. And let's run this on 105 because now I'm recompiling every single time, and let's execute. And we can see really fast on that guy, and here's our large time span on line 114, and let's execute. Wow, and you can see how much faster it is. And let's go back over, and let's scroll down just a bit. Starting on line 122, I have the Pluralsight course, Identifying and Fixing Performance Issues Caused by Parameter Sniffing, by Gail Shaw. It's a great course, there's several other options that you can do to remediate parameter sniffing issues. There's an Optimize for hint, which is saying hey, I'm going to optimize it for a particular value, maybe you want a middle of the road kind of plan created. You can also declare local variables, which are essentially just going to recreate the execution plan, or if you find this is just like a nagging issue once a month, you can clear out single execution plans for using DBCC FREEPROCCACHE, and passing in the actual execution plan Id. Again I highly recommend checking out her Pluralsight course.

Summary

In this module, I touched on how important it is for you to have some sort of baseline in place when approaching optimizing stored procedures. It's hard to know how to make something better unless you know what better is. To create that baseline and performance markers, we looked first at a built-in dynamic management view with aggregated stats. For individual executions, you used statistics io, which shows you how many pages SQL is reading back. Trying to reduce pages being read from either disk or cache will likely give you the most bang for your buck in getting your sproc to run faster. We then looked at a couple of common myths you routinely see for improving sproc performance. The first being that temp tables cause execution plans to not be reused. You know how to easily bust this myth now. The second being that you should avoid a select * when you're checking for the existence of some value. If you don't know why you're doing something, always feel free to test it out. Along those same lines, I talked about how important it is that you use multiple values when testing your sprocs. Finally, you learned what parameter sniffing is, and that it's an expected and desired behavior in SQL Server. There are situations where parameter values being cached can negatively affect your stored procedure performance. I introduced you to a couple of different methods for dealing with parameter sniffing issues, one being to recompile the entire sproc, the other is just to focus on recompiling the affected statement. Please join me in the next module where you'll be Building Stored Procedures in the Real World.

Building Stored Procedures in the Real World

Introduction

Hello. My name is Jared Westover, and I'm recording this course for Pluralsight. This course is Capturing Logic with Stored Procedures in T-SQL. In this module, you'll be Building Stored Procedures in the Real World. My hope with this module is to provide you with a recap of the main points from this course. You know how to create stored procedures and add elements like parameters and a debug flag. This module will serve as a reinforcement of all your newly-acquired skills. To start us out, I'm going to talk about something which can easily be overlooked, and that is creating a template for your stored procedures. Having a template can save you time down the road, and more importantly, it can ensure all your developers are on the same page. I'll introduce you to a few different elements you may want to include in this template. For me, an important one is error handling. Since you're going to have multiple stored procedures which perform different tasks, you'll likely want to have more than one template. You may not want to apply error handling to a sproc, which is performing a simple select statement. I've seen developers in the past try to make one template fit every situation. Next, we're going to review some of the key points discussed throughout this entire course. I've compiled together a listing of what I would consider to be the top three in no particular order. This will include a review of why you should even create a stored procedure in the first place. Also, one of the items which can really make or break how your stored procedure performs is defining some sort of baseline during the development phase. Finally, I'll share with you a few different Pluralsight courses which go into various aspects of stored procedures. Remember, it's extremely important that you take what you learn here and put it into practice to get the most benefit. We have some great information to cover in this module, so let's get started.

Defining Stored Procedure Templates

Years ago, I worked in support, and we would see a lot of the same issues over and over again. One thing we did to streamline our tickets was to create templates for particular error messages or tasks which needed to be completed in SQL Server by the clients. Our SQL developer Susan can do the exact same thing when it comes to stored procedures. If you perform a search online, you can likely find a few different examples, which can be used. I've compiled together a listing of three main things you may want to consider including. The first is having some method of error handling. I generally only use error handling when performing some sort of insert, update, or delete. SQL has a couple of options, the older one, which has been around for several versions, is RAISERROR. Microsoft actually recommends using the Throw command introduced first in SQL 2012. To go along with error handling, you'll want to try to include try catch blocks, which allow you to be in control of when the errors do pop up. The next item I would suggest you include is explicit transactions. They're defined using the begin tran and commit tran, and if things don't go as intended, rollback tran. Using explicit transactions allow you to be in control of the flow of your stored procedure. Who doesn't want to be in control of when things go sideways? The last thing I want to touch on is something simple, but you find varying opinions online, and that is adding comments to your sprocs. Some developers are a fan of them, others say well, you should be able to read the code and know what's going on. I'm personally a huge fan of adding comments, if for no other reason than to have some notes of what I'm actually doing in the procedure. If you create a lot of procedures, it's easy to forget when you go back to make changes a few months later. I think it's always a good idea to provide as much information to other developers who will be working on your sprocs. You giving them detailed information will ensure continued success across your team.

Demo: Creating Templates

In this demo, I want to show you a basic template that I put together which should give you a great starting place. I'll walk you through some of the basic elements which you can incorporate. More than anything, I want you to take this template and make it your own. What I have on this screen is a template that I put together for a procedure where you may do an insert, update, or a delete, and starting on line 2 down to line 6 is just some comment information, which is likely to be helpful. We have the author's name on 2, 4 has just detailed description of what the stored procedure is doing, and then if we make any updates to the procedure, we may want to capture them here, especially if there's a major update to the functionality of it, we may want to capture it here, and whoever made that change, so if there's issues, that we'll be able to go them to seek help. Starting on line 9, I have the CREATE OR ALTER syntax, and then to the right of that, I have some template parameters. The first one is called Schema Name, and that is of the sysname data type, and then the default value for it is Sales, and then to the right of that I have the Procedure_Name, sysname, and then the default value of that is Update. And then below that, we have our actual input parameter, and that's @param1 of the sysname data type, and the default value is just going to be @p1. And then our default data type for that parameter is going to be integer. And we'll see how the template parameters come into play here in just a minute. Now let's scroll down just a bit and see what we've got going on in this procedure. Starting on line 14 is something I generally always include, and that's just my SET NO COUNT ON, so I don't get messages back. On line 16, we are performing a BEGIN TRY because we want to be in control of if an exception occurs. And then on line 18, we're beginning an explicit transaction, 20, I just have, that's where the meat of our code goes, and then if everything goes well on 22, we are going to commit the transaction. Let's scroll down just a bit more. In 24, we're going to END the TRY, and now if we do have an exception, on line 26, I'm beginning that CATCH, 28, we're checking to see, hey, are there any open transactions, and if there are, on line 30, we're rolling them back, and then 32, we're going to throw that exception, and that will terminate the procedure at that point. Let's scroll back up to the top. Now I want to take a minute and show you something you don't see that often, and that is the Template Explorer or the Template Browser. If we go up into the menu bar under View, and then choose Template

Explorer, over here on the right, and I'm going to pin this guy so he doesn't disappear, we have the Template Browser. And basically these are folders that have any type of script that you can imagine underneath of it. Now if I scroll down, and I've already expanded it here, but it's under Stored Procedure, these are all the templates that SQL Server has automatically installed. Now I created a custom folder just because I didn't want my templates to be intermingled with all the standard ones. Now I also added in a Custom template that was called Create Procedure Update. And it's very simple to add a folder, you basically just right-click and choose Folder, or if you're adding a template, you choose Template. Now I want to edit this template because currently it is blank. So if I just right-click, go to Edit, now I want to take what I have over here in my script, and I'm going to copy that guy, and let's go ahead and paste him. Now I'm going to save this, and I'm going to close it, and let's go back over here. Now if I want to use that template, I can simply just double-click on it, and it'll open up a new query window with that template in it. And now we want to populate those parameters, and to do that, we're going to go up to the menu bar and click Query, and there's an option that says Specify Value for Template Parameters. So when we choose that, we are presented with a window to populate those values. So the first one, the Schema_Name I already have is Sales, so that's fine, the second one is Update, you probably would want to include a little bit more detail with that, also for our parameter 1, the name, and then the datatype is fine as integer. So I'm just going to click OK. And you can see that those are updated automatically. Now let's go ahead and go back to my original script, and I'm going to scroll down just a bit. What I have starting on line 42 is a Microsoft article about the Template Explorer that you can go and check out, it has a few more hints and tricks on there. I think a more detailed article though is on line 45, and that is by SQL Shack, I definitely recommend going and checking out both of those if you want to learn more about using templates in SQL Server.

Exploring Key Takeaways

I wanted to spend just a couple of minutes going over what I would consider the key takeaways from this course. The first would be the reason for creating a stored procedure in the first place. If you have a lot of ad hoc queries in your production environment, going through and creating hundreds of sprocs can be extremely daunting. Just like if I was trying to get you to replace your windows in your house with a different brand, I would need to have a pretty good sales pitch for you to make the investment. If you remember back, the primary reason for me is having the maintainability of the code, especially if you're working with multiple developers. It can be a bad idea to have SQL scripts floating around in people's email or even up on a shared drive. Having this code saved in a sproc on the server seems like a better solution. Then we have the added security, which comes with using a stored procedure, in that you have more granular control over user's access to tables by just allowing them to execute the sproc to get their job done. I mentioned it before that you see performance touted as a benefit, and don't get me wrong, you will notice a performance gain from your using the execution plan because creating one each and every time is an expensive operation, but it's also nice because you have the reduce network traffic by just having the call to the sproc versus a couple hundred lines from an ad hoc query. Your second important takeaway is that you want to establish some sort of performance baseline with your procedures, whether you come to an agreement with the business that report sprocs need to execute in less than 30 seconds, establishing this baseline will ensure success from the start. We saw how you can use a DMV to track the performance at an aggregated level, but you can also use statistics io to capture the individual executions. Again it's important to use a wide variety of parameter values. As we saw in the last module, you don't want to make your sprocs perform well artificially by sandbagging the parameters. I would suggest you take the mentality of trying to break your sprocs when testing it out. It's better that you break it versus an end user in production. Finally, I would highly recommend that you put together a set of best practices. You can use some of the ones I talked about in this course, such as ensuring set nocount is on, or making sure to implement a debug flag when dealing with temporary objects. Whatever best practices you choose, start by writing them down or placing them in a Word document that you can share with other developers on your team. Perhaps you can work in a group to come up with them. Making sure they are simple to implement allows you to be consistent with practicing them.

Continuing the Journey

I wanted to leave you with a few different Pluralsight courses that you can use to continue on your path to creating and optimizing stored procedures. The first is optimizing stored procedure performance, and there are two parts to this one. These were put together by Kimberly Tripp. They're fairly long, around 7 hours each, but are more than worth the time commitment. I think part one was the first Pluralsight course I ever watched. I promise you'll learn so much from these. Kimberly is such an awesome educator, you can always count on learning something new from her. The next would be a course I mentioned in a previous module, and that is Ethical Hacking: SQL Injection. SQL Injection is a hot topic for good reason, and it's something you hear a lot about in the IT world. If you're using dynamic SQL, I would suggest at least becoming familiar with it. This is one part of Troy Hunt's Ethical Hacking track on Pluralsight. Another one I mentioned in the last module is Identifying and Fixing Performance Issues Caused by Parameter Sniffing, and that's by Gail Shaw. If you work a lot with stored procedures, you will, without a doubt come across parameter sniffing issues at some point. The last one is called SQL Server: Temporary Objects, and that is by the great Joe Sack. Since you will likely be using temporary tables or table variables in your procedures, this is a great course to check out. It's fairly short, but goes into great detail about temporary objects. I highly recommend any of the courses by Joe Sack, especially his Performance Tuning ones. I hope you enjoyed this course and had as much fun watching and learning as I did with making it. I look forward to talking with you again.