Course Overview

Course Overview

(Music) Hi everyone. My name is Pinal Dave. Welcome to my new course, Using Memory-optimized Tables and Native Stored Procedures. I am a SQL Server performance tuning expert at sqlauthority.com. In-Memory OLTP technology provides us the ability to optimize the performance of the last transaction processed in a fraction of the time. This course is a quick introduction to two of the major components of SQL Server's In-Memory OLTP technology, memory-optimized tables and natively compiled stored procedures, commonly known as native stored procedures. You just need a basic understanding of SQL Server for this course. Some of the major topics that we will cover are creating memory-optimized tables, optimizing performance of in-memory tables, creating natively compiled stored procedures, collecting execution statistics for natively compiled stored procedures, and finally, a summary. By the end of this course, you will have a solid foundation to get started with In-Memory OLTP technology. I have included a very interesting demonstration in the last module, which I call Convince Your Boss. I hope you will join me in this journey to learn memory-optimized tables and natively compiled stored procedures, at Pluralsight.

Creating Memory Optimized Tables

Version Check

Introduction

Hi. This is Pinal Dave, and I welcome all of you to the module, Creating Memory Optimized Tables. Let's see the agenda of this module. First, we will learn about the In-Memory OLTP technology of SQL Server. Right after that, we will discuss memory-optimized tables. Following that, we will also understand why memory-optimized tables give us faster performance, and also see some of the additional components of In-Memory OLTP. Along with that, we will see use cases and strategies where we will discuss about why we want to use In-Memory OLTP, and specifically memory-optimized tables. Following that, we will see a demonstration. Trust me, at the end of this module, you will learn how to create your very first memory-optimized table. Now, let's start understanding what is In-Memory OLTP.

What Is In-Memory OLTP?

What is In-Memory OLTP? As per a definition, In-Memory OLTP is the SQL Server technology for optimizing performance of transactional processing, data ingestion, data load, and transient data scenarios. Well, if this definition looks very complicated, let me say it in simple possible words that In-Memory OLTP improves performance of online transaction processing. If you have lots of data in a smaller transaction, which you are processing every single day, this particular technology can be very helpful to you. In-Memory OLTP is available in SQL Server, which is an on-premises installation, and Azure SQL Database, which is in the cloud. In-Memory OLTP is very similar to its own counterpart where we have data and tables on the disk. As it is very similar, you do not have to do many changes to use this technology. Additionally, you can also have your traditional disk-based tables in the same database where you have In-Memory OLTP technology enabled as well. You can seamlessly run your query with your in-memory table and disk-based table all together. Well, this is just a definition, let's understand the four major components of In-Memory OLTP next.

In-Memory OLTP Components

So far, we only know In-Memory OLTP's definition. Now, let's see four of the important components of In-Memory OLTP. So, the first component is memory-optimized tables. Well, memory-optimized tables, as the name suggests, stores the data into the memory, but just so you know, if anything happens with your memory, the data is not going to go away because data is definitely going to be persistent, or data is definitely going to be durable in the log file. As memory speed is 5 to 20 times faster by some of the benchmarks, you can be confident that memory-optimized tables always give you the fastest possible performance compared to disk-based tables. Remember, in the In-Memory OLTP system, you can always have memory-optimized tables along with a disk-based table, and you can use both of them together. The next component is nondurable tables. If you are a SQL Server DBA or a developer, you must be familiar with temporary tables. Temporary tables are created in the session and are destroyed at the end of the session. Similarly, you can use nondurable tables to replace a temporary table in SQL Server. When you create nondurable tables, just specify that you want to create only schema at SQL Server starts, so when you restart your system, all the data will be lost, and the schema will be kept. Now, the third important component, which is memory-optimized table variables. I'm sure many of us use table variables or table-valued parameters to store intermediate result sets in stored procedures. Well, the same can be used with memory-optimized table types. Just like nondurable tables, memory-optimized table types also does all the operations in memory and absolutely there is no activity on I/O. And the final module is natively compiled T-SQL modules. A stored procedure triggers a scalar a user-defined function at the create time can be declared as natively compiled T-SQL modules. All the natively compiled T-SQL modules reduces CPU cycles and they process a large amount of the data very, very quickly. Now, here we have all the four components of In-Memory OLTP. Memory-optimized tables use lots of memory, but that eventually leads to optimal performance for you. Nondurable tables reduce I/O and memory-optimized table types also reduce I/O consumption. And finally, natively compiled T-SQL modules reduce your CPU needs. When you look at this, it's very, very clear that In-Memory OLTP systems optimize the use of memory and reduces all the other resource requirements and eventually gives you the fastest possible performance. I am personally quite a big fan of In-Memory OLTP systems; however, before implementing memory-optimized tables and natively compiled T-SQL modules in your system, you should carefully evaluate your entire system, and that's what we are going to discuss in the rest of the course. Now, in the next clip, we are going to talk about some of the needs related to the In-Memory OLTP system and memory-optimized tables.

Incorrect Assumptions

Every time when I go for a SQL Server performance tuning consultancy and I talk about In-Memory OLTP, I immediately get some questions. It seems like there are a lot of misconceptions and incorrect assumptions about In-Memory OLTP, as well as memory-optimized tables. Let me list down a few of the incorrect assumptions and the realities along with it. The number one thing which I hear, it is that In-Memory OLTP is an immature product. Well, as per my opinion, it could be true if it was the year 2014 when this product was released. Definitely there were quite a lot of limitations along with In-Memory OLTP. However, right now, we are in the year 2017, and Microsoft has done a fairly good job in removing most of the limitations from In-Memory OLTP systems. As a matter of fact, even in the year 2014 when it was released, it was actually well researched for quite a long time by the Microsoft research team before it was released. However, in the year 2014 when this feature was initially introduced, it was introduced with limited features, hence many people think it was an immature product. Honestly, in In-Memory OLTP is extremely stable and robust, and many of my customers use that today. When I talk about memory-optimized tables, many of the senior DBAs often respond saying, it seems like reimplementation of DBCC PINTABLE. Actually, it's very incorrect. PINTABLE was only keeping table pages in a memory, which was 8 K in size and still has a latch and locks taken for any modification. In-Memory OLTP is a latch-free engine, so you will not see locks or latches on your table when you are using in-memory. That is one of the reasons why you should go for memory-optimized tables. When any of my customers come to me talking about latches and locks on their important tables, my first reaction to them is, have you considered memory-optimized tables as a replacement for your disk table which has lot of locks? Usually I hear no or incorrect assumptions, like two of these which I listed and a few I'm going to talk more about. The next incorrect assumption is, it's a separate installation. A lot of people think it's like a full tech search. You have your main engine and there is a separate engine. It is not true. In-Memory OLTP, or memory-optimized tables, or natively compiled stored procedures are not a separate product. This is part of the SQL Server product itself, but you will only find them in the Enterprise Edition 64 bit, or in the Developer Edition. Remember, the Developer Edition is good for your testing and development purposes, but you should not run that on your production. When you are installing your SQL Server in the Enterprise Edition, make sure that you select the Database Engine services component and In-Memory OLTP will be automatically installed. Once I gave a fact for three of the incorrect assumptions I discussed earlier, the next question my customers ask about is, if they have to do a modification to take advantage of the system. Well, don't trust anyone if they say zero modification is required. At least a tables schema needs modification to move a disk-based table to an in-memory table. We will learn more about this later in this module. And finally, the biggest one is that data is not durable. However, all of you already know that data is definitely durable in memory-optimized tables from the previous clip. When we say memory-optimized tables, it doesn't mean that data only resides in memory and it's not durable. In-Memory OLTP systems, and particularly memory-optimized tables provide full durability for its data. When a transaction has made changes to a memory-optimized table, SQL Server, just like any of the disk-based table, guarantees that changes are permanent and can survive the database restart provided that the underlying storage is available. I think that incorrect assumption of data is not durable might have come from nondurable tables which are replacements of the table. Now you know five of the important misconceptions and incorrect assumptions. Let's talk about one more incorrect assumption which I often to say from an IT administrator. It is about poor security of the data. Well, data is as secure in memory-optimized tables as secure as it is in your disk-based tables. As a matter of fact, memory-optimized tables now even support row-level security. So, rest assured that you do not have to worry anything about database security or data security when you use memory-optimized tables, and that's it. Now, a few of the use cases and strategies where you should use In-Memory OLTP and memory-optimized tables.

Use Cases

In the previous clip, we have seen some of the incorrect assumptions of In-Memory OLTP technology. Because of these incorrect assumptions, many of the people and industry experts do not implement this technology. As a profession, I am a SQL Server performance tuning consultant, and every single time when I go for a consultation at my customer place to help them with performance optimization, my first priority is to look for an opportunity to implement In-Memory OLTP technologies in their application. I know that if I successfully find a place and time to implement this technology, they will have a big win. Let's see some of the examples and use cases where I like to implement this technology. One of the first places is I look for caching or session state management implementation. The In-Memory OLTP technology makes SQL Server very, very attractive for maintaining session state. If you are using an ASP.NET application, you must be aware that caching is very critical for your ecosystem. It is important that you manage session state efficiently and effectively. Sometimes there are thousands or even millions of requests per second. With the help of In-Memory OLTP technology, you can successfully overcome any latching or locking situations with the tables which are using session state management. The next opportunity I always look for is data ingestion. For example, if you have any system where you are retrieving data from multiple different sources at the same time, you can definitely consider implementing In-Memory OLTP technology. I will definitely focus on nondurable memory-optimized tables, as well as memory-optimized tables to handle large volumes of data ingestion. Take an example of the Internet of Things. This is where you can efficiently use in-memory technology. I'm sure you've heard about multiple Vs of the big data. If you are ever dealing with high volume and high velocity and unit to return, your result almost in a real time, you want to consider memory-optimized tables. For example, one of my customers who is into a stock market business, they have thousands of data coming in every single second, and they have to analyze them and display almost immediately to all the clients. I was able to successfully implement a dashboard, which instantly shows what recommendation their customer has with the help of memory-optimized tables. At the risk or repeating myself, I would say it again, as there are no latches and locks, dealing with high-volume and high-velocity data is relatively easier when you are dealing with In-Memory OLTP technology. If you are using temporary tables, table variables, and table-valued parameters, you should definitely consider using nondurable memory-optimized tables. Memory-optimized table variables and nondurable tables reduce all the I/O oppressions which eventually leads to reduction in the CPU consumption. Recently at one of my customer's place, they had a huge amount of TempDB. This is because they were using lots of temporary tables and processing a huge amount of the data inside the temp table. They were also using lots of table-valued parameters to pass data from one stored procedure to another stored procedure. They were really struggling with their CPU. After replacing all of the TempDB objects with nondurable tables and variables, we were instantly able to improve performance multifold. I truly suggest that if you have a time, you should create a proof of concept where you can compare your regular temporary disk-based table with nondurable memory-optimized tables. You would be surprised how much performance improvement you can get by implementing this technology. And finally, I want to talk about ETL workload or ETL workflow. If you are using an extract, transform, and load mechanism, I strongly suggest that you consider using In-Memory OLTP systems where you load your data into a staging table, make necessary transformations, and load them into a final permanent table where you want it to be. If you ask me, in the industry, I see lot of my customers using In-Memory OLTP technology for ETL workflow. At this point in time, I'm pretty content when I see lot of ETL workflow using in-memory technology. However, I really wish that users start using in-memory technology beside ETL workflow and get maximum performance out of it for that application. Well, this is the use case scenario for In-Memory OLTP technology. All this theory has no meaning if you do not back it up with a good demonstration. Let's see a demonstration where we will create our very first memory-optimized tables next.

Demo: Creating Memory Optimized Table

In the previous clip, we have seen use cases for In-Memory OLTP technology. Now is the time where we will see our very first demonstration of creating memory-optimized tables. Before we go and create memory-optimized tables, we need to make sure that we have the proper environment, and right after that we will create our very first memory-optimized tables. So now is the time when we switch over to SQL Server Management Studio. Here we are in SQL Server Management Studio, and I will be running all of my demonstrations in SQL Server 2017. To check which version of SQL Server 2017 I'm running, I am going to select the statement and click on Execute. Over here, you can see the version of the SQL Server which I'm using. When I'm building this video course, this is the latest version of SQL Server 2017. Remember that In-Memory OLTP technology was introduced in SQL Server 2014 and was subsequently enhanced in 2016 and 2017. Now first, you will have to create a database. Here is the script which I will be using to create a database. If you've ever created a database in the past, you must be familiar with the script on the screen. We always need a minimum of two components. One is a data file and the second one is a log file. Now, if you want our database to support In-Memory OLTP technology, we need to also specify a few other things. Here is the highlighted portion for you. This portion indicates that this is the change from your regular database creation to In-Memory OLTP database creation. Yes, that's it. This is the syntax you need to remember, or you need to execute when you create your database to make sure your database is In-Memory OLTP enabled. First, we specify the file group, and here is the name of the file group. Right after that, we specify the directory where our In-Memory OLTP files will be stored. In-Memory OLTP technology is based on FILESTREAM. So when we specify this syntax, it will create FILESTREAM structure into the folder specified over here. Here is my D directory data folder. Currently, the folder is empty. Once we create our database, we will come back over here and check what different files are created. Now, we will select the statement and execute it. This will create new a database, SQLAuthority. The database is created successfully. Now, let's go back over to the D drive and the data folder and see what it has. There you go, it has three things. First is a data file, second is a log file, and the third one is the directory which we have just created. Double-click over here and click on Continue to give permission to access the folder. This is where all the content of in-memory technology will be stored. Now, let's go back over to SQL Server Management Studio.

Now we have successfully created a database, which is enabled for memory optimization data. First, we will select our database, which is SQLAuthority. Next, we will create a regular table, which is DiskBasedTable, as you can see over here. This is the same table which you have been creating all your life. Now, select the statement and click on Execute. Next, we will create our very first memory-optimized table. Remember, this table will be durable. Now, if you see line number 13 to line number 17 are exactly same as what we have seen before. If you want your table to support memory optimization, you just have to add these three lines. Yes, it is absolutely that easy. As a matter of fact, this line is also optional. If you do not specify durability, by default, SQL Server specifies durability as schema and data. The only reason I'm including this line over here is for additional clarity. Now, select the statement and click on Execute. As soon as we execute, it will create memory-optimized data. Now, this particular table is created in memory of SQL Server. If you want to create a nondurable optimized table, you can just specify durability as schema only. Let's create our very first nondurable memory-optimized table as well. Select the statement and click on Execute. A nondurable memory-optimized table will not contain any data when you restart your SQL Server. This will just act like your temporary table, whereas when we specify durability as schema and data, when we restart our SQL Server, our data will be also available in our table. Now, you may wonder why I have specified a PRIMARY KEY as NONCLUSTERED instead of CLUSTERED. Well the answer is very simple, but to answer your question, I am going to remove the word NON from this statement and run the same script with a different table name. Upon execution, the query will throw an error. It says error number 12317, and the message is, Clustered indexes, which are the default for primary keys, are not supported with memory-optimized tables. You must specify NONCLUSTERED index itself. Well, there you go. This our very first learning about limitations of memory-optimized tables. We will also see in a future module how we can further optimize in-memory tables, and also touch base on a workaround to create a clustered index. As you know, we have already created three tables in our database. The first one is the DiskBasedTable, second one was MemoryOptimizedTable with DURABILITY = SCHEMA_AND_DATA, and third one was our memory-optimized table, which was nondurable. You can run this DME and check various properties of your table. This is the name of the schema. Right after that we have the name of the table, and here, this column indicates if the table is memory optimized or not. When you see value equal to 0, that means this particular table is not memory optimized. The durability is also described in the very next column. First two tables are durable by schema and data, whereas the last table is only durable for schema, and the last two columns demonstrate create date and modification date. Well, that's what we wanted to learn in this module. There is a lot of different things we can discuss about optimizing in-memory tables, but that we will see in the very next module.

Summary

All right, so that was the demonstration about how to create our very first memory-optimized tables. Trust me, it is not as difficult as we think. Now, let's summarize this module. In-Memory OLTP technology is much faster and robust than it was many years ago. I strongly recommend to all my customers to look for an opportunity where they can use

In-Memory OLTP technology. Memory-optimized tables, durable or not durable, they both have their own place in your application. Our latest ecosystem needs to process high volume, high velocity data in almost real time. In-Memory OLTP technology absolutely addresses that with the help of not locking memory-optimized tables. You should definitely consider using memory-optimized tables for your system because they coexist with your traditional disk-based tables. As far as what I have seen in the industry, In-Memory OLTP systems are always able to support hybrid business requirements when it is about performance tuning. Now, in the next module, we will understand a little bit more about in-memory tables and how to optimize them.

Optimizing Performance of In-Memory Tables

Introduction

Hi. This is Pinal Dave, and welcome to the module on Optimizing Performance of In-Memory Tables. Let's first see what is the agenda of this module. In the previous module, we discussed how to create memory-optimized tables, and in this module, we will particularly focus on how to even further improve performance of the same memory-optimized tables, which we have created in previous module. First, we will talk about why memory-optimized tables are faster. Here, we will discuss about three of the primary reasons for the same. Right after that, we'll discuss about various tradeoffs related to memory-optimized tables. It's not that memory-optimized tables are always going to be the best possible solution for you. There are definitely some points which you need to keep in mind when you decide to start using this. Right after that, we'll discuss about compatibility levels. At this point in time, we will understand at what compatibility level memory-optimized tables are effective. The next topic is about memory-optimized filegroups. In the previous module, we did not discuss about this, but in this module it's important to understand how filegroups play a very important role for memory-optimized tables. Next, we'll discuss about snapshot isolation level. While in the real world we are going to use not only memory-optimized tables, but also disk-based tables, when you have to use both of these kinds of tables in a single query, you definitely need to focus on snapshot isolation level, which is suitable for memory-optimized tables. Otherwise, you will not get necessary performance, but rather you will end up with an error. We will discuss this thing in detail in this module. Right after that, we'll discuss about memory-optimized nonclustered indexes. There are two different kinds of indexes available for memory-optimized tables, and we will create both of them and compare their performance, and also discuss in detail which index is suitable in which scenario. We will definitely back this up with interesting demonstrations. Well, this is our agenda of this module. Now, let's jump to very next module where we will discuss about why memory-optimized tables are faster.

How Memory Optimized Tables Perform Faster?

In this clip, we will discuss why memory-optimized tables are faster. First of all, memory-optimized tables have a dual nature. Dual nature means a memory-optimized table is in memory, as well as on the hard disk. However, all the transactions are done against the memory, hence you get amazing speed at one bit of the memory, and you do not get slow down of the traditional reasons of disk. Additionally, memory does not have any pages, so all the transactions get the benefit of a pageless structure, which removes contention due to latches and spinlock. Dual nature is the primary reason why memory-optimized tables are faster. No locks. The memory-optimized table relies on an optimistic approach, and they do not put any lock on any version of the updated row of data. The goal is to give maximum performance while running against concurrency and high throughput. As there is absolutely no lock on this pageless architecture, memory-optimized tables greatly reduce contention when you have a high amount of transaction going on in your system. You may think, if there is no lock, how do memory-optimized tables manage the concurrency? The answer is row versions. Instead of locks, the memory-optimized tables depend on row versioning in the table itself instead of traditional TempDB. The original row is always kept until after the transaction is committed. That's why if there is any other transaction who wants to read the data, it can always read the original data where the newer or updated data is returned in an absolutely separate row. This is an amazing architecture change compared to the disk-based operation where you do not have to put any locks on any of your data if it is being read while it is being updated. The same architecture is definitely there in a disk-based table, but at that point of time, the row versioning is created in a TempDB. Now, you are actually using multiple I/O resources and you are further slowed down, whereas when you are using memory-optimized tables, row versioning is much easier and frictionless compared to disk-based architecture. Additionally, we also get benefit of less logging. The before and after version of updated rows are held in the memory-optimized tables itself. As we have now both the version before and after, the system has to write less amount of data into a log file. Due to less logging nature of memory-optimized tables, the performance is even further improved. If you are ever facing performance bottleneck related to write log, definitely memory-optimized tables can help you. And finally, memory-optimized tables perform best when they are accessed with natively compiled stored procedures. Natively compiled stored procedures greatly reduce the number of instructions to execute, hence they further improve performance of memory-optimized tables. Well, these are the five reasons. We will discuss natively compiled stored procedures in detail in the next module. However, for now, in the next clip we'll discuss about tradeoffs related to memory-optimized tables.

Trade-Offs

In this module, we are going to discuss about three important tradeoffs of memory-optimized tables. The very first one is estimating memory. Absolutely, this is one of the biggest tradeoffs because it's impossible for a DBA or developers to estimate how much memory you would need for your memory-optimized tables. Definitely there are quite a lot of calculations and resources available where you can input your data and can get a guess of how much memory you need, but it's never going to be accurate. As a DBA or developer, you will just have to trust yourself and put relatively quick mathematics to estimate the memory. If you do not estimate a good amount of the memory, it's quite possible, eventually your memory-optimized tables may struggle. Another important tradeoff is partitioning a large table. It's quite possible that you want to optimize a very large table, but that table may not fit into available memory for your server. In this scenario, you might have to partition a large table and create two different zones, or two different partitions. The first part is, which is called as hot recent data, where you store all the data which is needed for your active queries and active needs. And the second part is called cold legacy rows where you store any data which you do not need immediately, or archived data. Well, this is something you can definitely do with a memory-optimized table where you create two different zones, but this is not something memory-optimized tables is going to do it for you. This is something you would have to design and implement yourself and only store active data into your memory-optimized partitioning. This is not as easy as you'd think, but if you are an advanced user of memory-optimized tables, this is the only option for you where your table is extremely huge. And finally, natively compiled modules. Natively compiled modules are usually very, very fast because they execute less amount of the instruction when they execute, but when you restart your SQL Server services in any of the natively compiled modules, it's called. At that point of time, your natively compiled module has to be recompiled whenever it runs the first time. This compilation may take little bit of time. I do not think it's a big tradeoff because, as a matter of fact, if you have a disk-based stored procedure, they often have to recompile when you bring back your SQL Server as well. Well, these are the three known tradeoffs of memory-optimized tables. Now, let's go to our very first concept about the compatibility level and we will see a demonstration right after we discuss about the compatibility level.

Compatibility Level

The compatibility level is one of the most important settings or configurations when we discuss about memory-optimized tables. It sets certain database behaviors to be compatible with the specified version of SQL Server. If you want to use memory-optimized tables, you need to make sure the value of this configuration setting is set to a minimum of 120. The maximum value of this configuration setting can be whatever is the latest for your SQL Server. In this course, we'll be using SQL Server 2017, hence the value of this setting would be 140. If you are using SQL Server 2016, the value of this configuration would be 130. Remember, memory-optimized tables are not supported for any version prior to SQL Server 2014. That means if you have a compatibility level which is lower than 120, you cannot use memory-optimized tables. My solution is to keep this configuration setting to whatever maximum value your SQL Server supports. In the next clip, we will see how we can change compatibility level with the help of SQL Server Management Studio and T-SQL.

Demo: Compatibility Level

Here I am in SQL Server Management Studio, and first I'll be running a setup script which will create a database called Pluralsight. Right after that, I am changing compatibility of this to an earlier version of SQL Server. This is just so I can teach you how to change the compatibility level with the help of T-SQL and also with SQL Server Management Studio. So now our real demonstration will start. First, I will go to Object Explorer and select Databases, and hit on Refresh. Over here, you can see now my database, Pluralsight, is visible. To check what is the compatibility level for all the databases in your system, you can just run line number 2 and 3. When you select the statement and click on Execute, it will show you all the databases on your server and its own compatibility level. Here is our database, Pluralsight, and its compatibility level is 110. Well, 110 is not the right value if you want to use memory-optimized tables, hence we need to change it to either 120 or higher. Now if you want to check the compatibility level of your database only, you can run this entire script with the WHERE condition and it will demonstrate your database and its settings and its configuration value. Now, as I'm using SQL Server 2017, I will set my database compatibility level to 140 so I can take advantage of all the new enhancements available in SQL Server 2017. Here is the command to change compatibility level to the latest version of SQL Server. You can select the statement and click on Execute. Syntax is very simple. ALTER DATABASE, name of the database, and setting COMPATIBILITY_LEVEL to the latest value. You can do the same thing with the help of SQL Server Management Studio as well. Right-click on your database and go to Properties. Now go to page Options, and over here right on the top, the third drop-down is related to compatibility level. SQL Server 2017 supports all the way back until SQL Server 2008 with the value 100 over here. If due to any reason you set the compatibility level to 2008 or 2012, your database will not support memory-optimized tables. You need to make sure your value is either 2014, 16 or 17. If you are using SQL Server 2019, then you need to make sure this particular value is set to 150, but that's the conversation you need to do when SQL Server 2019 will be released. For the moment, we will set the compatibility level to 2017, and click on OK. That's it, now we have set the compatibility level to the latest versions of SQL Server, so we will be able to use every single enhancement available for In-Memory OLTP.

Memory Optimized FileGroup

Memory-optimized filegroup. In the previous module, we have learned that when you create memory-optimized database, at the same time, you also create memory-optimized filegroup. That's totally fine if you are creating a new database. Think like this way, you created a database many years ago, and now you upgraded your SQL Server. At this point in time, if you want to take advantage of memory-optimized tables, you have to follow certain steps. First of all, you'll change the compatibility, as I explained in the previous clip, to the latest compatibility level, and once you create that, you will have to add a memory-optimized filegroup. The reason you have to create a memory-optimized filegroup is very simple. Memory-optimized tables cannot be created on a disk-based filegroup. You have to create memory-optimized filegroups to create memory-optimized tables on it. The good thing is that we can add this filegroup after we have created a database. This will help many business applications to take advantage of In-Memory OLTP technology. Now we will see a demonstration where we will use SQL Server Management Studio to add memory-optimized filegroups.

Demo: Memory Optimized FileGroup

So far, we have created the database Pluralsight and changed the compatibility level to the latest compatibility level. Now we will try to attempt to create two tables in this database. First, change the context of the database. Next, we will create a disk-based table. This is the regular table which you have been creating all along. Please note that PRIMARY KEY over here is NONCLUSTERED. Now, let's execute this script, and you can see we have now created a disk-based table. If you are wondering why the PRIMARY KEY is NONCLUSTERED, it's because memory-optimized tables do not support a cluster index, and I want to create disk-based tables just to mimic memory-optimized tables, hence, I have left the PRIMARY KEY as NONCLUSTERED over here as well. Now, we will try to create a memory-optimized table. Please note, again, PRIMARY KEY is NONCLUSTERED as we cannot create a cluster index over here, and here are extra keywords to create a memory-optimized durable table. DURABILITY, over here is SCHEMA_AND_DATA. Let's create this table. You will see when we create this it will give us an error, and the error is it cannot create memory-optimized tables because it does not have MEMORY_OPTIMIZED_FILEGROUP. The error is very clear that to create memory-optimized tables we need at least one memory-optimized filegroup. Now let's go back and create a memory-optimized filegroup. Here is the script for the same. ALTER DATABASE, name of the database, ADD FILEGROUP, and you have to specify an additional two keywords, CONTAINS MEMORY_OPTIMIZED_DATA. That's it, this is the one line you need to remember if you want to add memory-optimized filegroup. Now, let's execute this script, and our database has memory-optimized filegroup along with regular filegroups. Let's validate that by adding memory-optimized file into this filegroup. To do that, we will execute this script. This script will now add a new file in this directory and put data on this filegroup. Remember, filegroup itself does not contain any data. It's the files which contains data. As the name suggests, filegroup is nothing but a collection of the files, so after creating a filegroup, we have to add a file to that filegroup. Let's execute this script, and the script is successful. Now, once again we will run the exact same script which we had run a few seconds ago and we had received an error. Let's select the statement, and when we execute the script this time, it will complete successfully and our memory-optimized table is created on our memory-optimized filegroup. Let's double-check that by going to the properties of the Pluralsight database. Right-click over here and go to Properties. Next, click on Filegroups. We have a regular filegroup, and right below over here, we have memory-optimized data, and this is the filegroup where we are going to store memory-optimized files. Also on the right side, we can see how many files we have created, which belongs to this filegroup. Now, let's go to the page which says Files, and over here we can also see additional data related to memory-optimized filegroup. Here is the logical name of the file, and the type of it is FILESTREAM. You can clearly see the filegroup. It says Pluralsight_FG, which we have added as a memory-optimized filegroup. As it is FILESTREAM data, there is no need to specify any autogrowth value. Click Cancel, and we are back to SQL Server Management Studio. In this demonstration, you have seen how we can take a regular database, which is not optimized for memory, and you can convert that to a memory-optimized database by just adding a filegroup and creating memory-optimized tables. Next, we will learn about another real-world scenario where we have to retrieve data from a regular table, which are disk-based and memory-optimized tables.

Snapshot Isolation Level

So far, we have seen that we need the latest compatibility level with memory-optimized filegroup to create memory-optimized tables. However, there is another real-world scenario where we need to discuss the snapshot isolation level. If you do not know what is snapshot isolation level, well, in simple possible words, it sets certain database behaviors to be compatible with the specified version of SQL Server. That's just in definition, a but a real-world scenario is such that you need to write a query which needs to access memory-optimized tables and also disk-based tables. When you need this kind of situation, you really need a different kind of isolation level for your query to work if you are going to use explicit transactions. For cross-container transactions, you need to use the MEMORY_OPTIMIZED_ELEVATE_TO_SNAPSHOT isolation level. When a transaction involves both a disk-based table and a memory-optimized table, we call that a cross-container transaction. Let's understand this concept a little bit more in detail with the help of a demonstration. We'll be once again going to our SQL Server Management Studio in the next clip and understand what explicit transaction is and how it accesses disk-based and memory-optimized tables together, and why we need this specific isolation level.

Demo: Snapshot Isolation Level

Here we are in SQL Server Management Studio, and now we will understand what is the deal about the snapshot isolation. First, we will change the context of our database to Pluralsight. Now we will run our queries with autocommit transaction. When we run each of the SELECT statements, they provide us results just fine. As there is no data in our tables, you are not seeing anything over here. However, the result of this query is just fine. The first SELECT statement is on memory-optimized table, and the second SELECT statement is on disk-based table. Both of these SELECT statements are running with autocommit transaction. What it means is that when we run this SELECT statement, as soon as the statement is completed successfully, the transaction is autocommitted. This is just working fine, even if we do join between memory-optimized tables and disk-based table. Let's see that by executing this query. Now we will see an example of explicit commit. Here we are beginning the transaction by specifying BEGIN TRANSACTION. Right in the middle, we have the same JOIN statement which we have just run successfully, and right after that, we have COMMIT TRANSACTION, which will complete the transaction we will start over here. Let's see what happens when we have transaction with explicit commit. First, let's execute this statement to start our transaction. Next, we will run the SELECT statement, and as soon as we run the SELECT statement, there is an error. Let's reformat the error message so we can read it properly. Here is the error, which is 41368. It clearly tells us accessing memory-optimized tables using the READ COMMITTED isolation level is supported only for autocommit transactions. This we have experienced just few seconds ago, and it further explains that it is not supported for explicit or implicit transactions. If you want to use explicit or implicit transaction, we need to provide a supported isolation level for memory-optimized tables using a table hint, such as WITH SNAPSHOT. We can definitely go for this method, but that means for every single query which you are writing, in transaction you will have to specify that particular query hint. Trust me, many of my customers try to do that, but eventually they get very much frustrated.

The solution of this is very simple. What we can do is to change the entire database's isolation level. For example, you can just set the database isolation level to MEMORY_OPTIMIZED_ELEVATE_TO_SNAPSHOT. That means when we execute the statement, the isolation level of this database is changed to this new isolation level. Here I have written a simple SELECT statement based on sys.databases. Over here, we are selecting three columns. The first is name of the database following with two columns. The first one is is_read_committed_snapshot_on, and the second one is is_memory_optimized_elevate_to_snapshot_on. Now, from the result set, it is very clear that our database, Pluralsight, definitely has is_memory_optimized_elevate_to_snapshot setting on. Here it is. Now, let's go and run our transaction, which previously failed. Let's start transaction, and right after that we will run this SELECT statement. Remember, this SELECT statement didn't work the previous time when we did not enable memory_optimized_elevate_to_snapshot. Now when we execute the statement, it just works fine. You can also execute COMMIT TRANSACTION right following it. This demonstration clarifies that you need to enable is_memory_optimized_elevate_to_snapshot if you are going to use explicit or implicit transactions in SQL Server where you are going to use memory-optimized tables along with disk-based tables. So far, what we have seen is related to configuration changes to make our memory-optimized tables useful. Next, we are going to learn about indexes where we will compare performance of the same queries with different indexes.

Indexes on Memory Optimized Tables

Now is the time when we discuss about indexes, indexes on memory-optimized tables. There are two different kinds of indexes available on memory-optimized tables. The very first kind of the index is memory-optimized nonclustered index. We have been seeing this from the beginning of our course. Every time when we create an index on memory-optimized tables, that index was always a memory-optimized nonclustered index. Remember, memory-optimized tables have one critical requirement, that it must have an index and it should not be a clustered index. That forces us to create this kind of index when we create memory-optimized tables. However, if you do not want to create memory-optimized nonclustered indexes, we do have another option. We can always create a hash index as well. This index is commonly known as hash index, but actually it is also memory-optimized hash index. Now, let us see three important differences between memory-optimized indexes and the disk-based indexes. First of all, memory-optimized indexes are entirely in memory. There is no concept of pages, FillFactor, or anything related to that. You do not have to worry about fragmentation or maintaining indexes as well. All the modifications to the index always stays in a memory, and they never go to the disk. That's why performance of memory-optimized indexes is phenomenally faster than disk-based indexes. And finally, when your database comes online, every single memory-optimized index is rebuilt from the beginning. This is why they are always available in memory and eager to serve any need of application. These are the three primary reasons why memory-optimized indexes are superfast, and they even take performance of memory-optimized tables to the next level. We discussed about two type of indexes. Let's understand little bit more about each of them in the next clip.

Nonclustered Index and Hash Index

In this clip, we are going to talk about two different kinds of indexes and where they are used. First, we will talk about memory-optimized nonclustered indexes. The durability of this particular index is schema and data. Most of the time, you have seen me doing this with primary key nonclustered. As I mentioned earlier, primary key in memory-optimized tables can be only nonclustered. This kind of index is very helpful when your queries with ORDER BY clause or queries with inequality, as well as with value ranges. We will see a demonstration about this one a little bit later on in this module. Now is the time to discuss about the second kind of index, which is memory-optimized hash index. Memory-optimized hash index is just an array of pointers. Every element of the array is called a hash bucket, and each bucket is the size of 8 bytes. In SQL Server, the maximum bucket per index is this large number. I'm not exactly going to read this number, but the number is in billions. The ideal number of the bucket count is twice the number of the distinct row in your column in your table. Memory-optimized hash indexes are very helpful when queries have exact match. Now we have seen these two kinds of indexes, let's see in action in the next clip and compare its performance with each other.

Demo: Nonclustered Index and Hash Index

Now here we are in SQL Server Management Studio, and we will use our previously created database, Pluralsight. First, let's change the context of this execution to the database. Next, we will be creating two different tables in this database. Remember, this database is already supporting memory-optimized tables. Here is our first table where I'm going to demonstrate that we are going to create a table with PRIMARY KEY NONCLUSTERED. Here are the keywords which you need to know if you are going to create memory-optimized tables. The DURABILITY of this table is SCHEMA_AND_DATA. Now, the table is created successfully. Next, I'll be creating a hash index. This is a new thing for you. So far, whenever I create an index, I have created nonclustered index, but this time I would be creating nonclustered hash index. There is additional an keyword, HASH, over here, and there is a BUCKET_COUNT. BUCKET_COUNT should be twice the number of distinct rows from the column on which you are going to create an index. I am well aware of it that in this table I'm going to insert around 500, 000 rows, hence, I selected BUCKET_COUNT as 1 million. A too large of a bucket count is also not good as it will waste a lot of memory and your table will remain empty. Now, if you see over here, the syntax is pretty similar to what you have seen before, so there is nothing new for you to remember when you create a hash index besides this highlighted portion of the syntax. Now, let's select this query and create a table with hash index. Once the table is created successfully, we will now populate both of these tables with around 500, 000 rows. The content in both of these tables is absolutely identical to each other. First, I will populate a table where we have nonclustered memory-optimized table. Next, I will populate our hash index table by executing this query. Now, we will do a quick test where we'll be doing a range scan on LName. Remember, I have created an index on column IDs. I have not created any index on column LName, and now I'll be running this range scan on LName. Before that, I will enable SET STATISTICS IO and TIME. All the memory-optimized tables are in memory, and they are actually not doing any I/O operation. Let's see if that is really true or not. Now, I executed the statement and it ran successfully. That means after every single query, we will be able to see various statistics related to I/O and TIME if they are available. Next, I will run both of these queries together, but when I run it, I'm going to enable actual execution plan by clicking over here. You can also press the shortcut, Ctrl+M, to enable actual execution plan. Now when I execute this, the answer is identical to each other. When I click on the execution plan, I see identical execution plan as well with missing index addition. This is a good sign because the table on the top has nonclustered memory-optimized tables, and table on the bottom has nonclustered memory-optimized hash indexes, but they both are on column ID which is their primary key. Whenever I'm looking for any other column which is not indexed yet, SQL Server just goes for table scan and gives us identical performance over here. This is definitely a good sign. When I click on Messages, I'm not able to see any I/O-related statistics because, as I mentioned, there is no I/O operation when we are dealing with In-Memory OLTP architecture. There is some CPU time, but we will talk about it a little bit later on. Now, I'm going to create two indexes. On our very first table, I will create index on LName. This is nonclustered, memory-optimized index, and on a second table, I will be creating hash index on the LName. Here, I'm keeping bucket size of 128. The reason is very simple, because I know in the LName, I'm going to insert around five to six different values only, so that means I can also keep the hash count around 12 or 20, but I'm going to go little bit higher and keep the bucket count around 128. Now, let's create this index by executing the script. Once the script is executed, I will run same script which I had run it few minutes earlier. Remember, here we are looking for LName, which is between A and E. That means any last name which starts with A, B, C, D or E will be answer of this particular query. Similarly, the query on the bottom is also an identical query. The only difference between both of these queries is FROM clause where I have used a different table name. Now, if you remember, memory-optimized tables are really good when we are doing range scan. Is it really true or not? Well, we can only know if you run this query. So now, I executed both of these queries, and once the query is executed, I will go to Execution plan and see the result. Query on the top is running with 26% query cost, where the query on the bottom is running at query cost of 74%. Remember, addition of both of these query costs will be around 100. A query which takes less resources is definitely an optimal query, and over here the optimal query is on the top. A perfectionist would say execution plan is not a great idea to look at query performance, and I totally agree with you. However, in this particular case, execution plan is demonstrating the right value. Here, the query on the top has Index Seek, and the query on the bottom has Table Scan, and also an index hint. The query on the bottom is not optimal. That means the index which we created, hash index, on LName is not the best choice for this situation whereas nonclustered memory-optimized index was definitely a good choice when we are doing a range scan. If you look at the execution plan, the query on the top has not taken any time in execution, however, the query on the bottom where we have created a hash index is definitely taking quite a lot of elapsed time. Yes, it is indeed true during the range scan memory-optimized tables are benefited by memory-optimized indexes. Now, let's look at a different demonstration when we will check the equality. Here, I'm just going to check LName with one particular value, and this is an equality test. Let's select both of these queries and execute them together. Now we have identical results in front of us, though the order of the rows is different, the result is actually the same, and that's what really matters to us. Now, when we go to Execution plan, it is very clear to us that in this scenario, the index on the top is really not a good choice, however, the index on the bottom with only 14% query cost is an optimal solution. So, it is very clear that when you create memory-optimized nonclustered indexes, it's not as good for equality operation as much as memory-optimized hash index. With these two demonstrations, we have learned that when we can use the hash index and where we can use the regular memory-optimized nonclustered index. Well, from this demonstration, you have learned when to use which index when you are using memory-optimized tables. Now, we will quickly look at the summary of this module.

Summary

Here we are in the final clip where we are going to discuss about summary of what we have learned in this module. First of all, we have learned that memory-optimized tables are very fast compared to disk-based tables. The reason for that is very simple because memory-optimized tables have absolutely zero I/O-related operations. All the operations of memory-optimized tables are done in memory, hence there is absolutely no lock, latches or any activity related to data pages. This brings memory-optimized tables to the next level of performance because memory is much faster than traditional disks. The tradeoff of memory-optimized tables only comes into play when you have less amount of the memory and you have a large amount of data to be stored on the same. If you go on a partitioning route, that may be a little bit more complicated for you compared to a disk-based solution. Memory-optimized tables are already fast from the beginning, and you can make it even more fast by creating indexes on the top of it. There are two types of indexes, memory-optimized nonclustered indexes and memory-optimized hash indexes. Both of these indexes are useful in their own use cases. For example, if you are scanning ranges or using the ORDER BY clause, go for nonclustered index, and if you have exit match and equality in your WHERE condition, hash index is most optimal for you. Well, so far in this course and in this module, we keep on talking about memory-optimized tables and memory-optimized indexes. Now is the time where we switch our learning, talk about natively compiled modules and natively compiled stored procedures. In the next module, we will talk about this topic in detail.

Creating Natively Compiled Stored Procedures

Introduction

Hi. This is Pinal Dave, and I welcome all of you to the new module, Creating Natively Compiled Stored Procedures. You have been hearing me talking about natively compiled stored procedures for a while, and in this module, we will discuss this in detail. Let's see the agenda of this module. First, we will talk about what is a natively compiled stored procedure and what are the different aspects of it. Right after that, we will see the use cases where we can use this efficiently and effectively. Next, we will talk about some of the limitations of natively compiled stored procedures and understand where it is not a good fit. Following that, we will compare it with interpreted T-SQL stored procedures or our regular stored procedure. This will be a very interesting conversation. And finally, we will see various interesting demonstrations. The way this model is structured is that first we will talk about a few theoretical concepts, and right after that we will see multiple demos together. Let's start discussing what are natively compiled stored procedures in the next clip.

Natively Compiled Stored Procedures

Now, let's discuss natively compiled stored procedures. First, we will see the definition. Natively compiled stored procedures are Transact-SQL, or T-SQL, stored procedures compiled to machine code rather than interpreted by the query execution engine. One of the most popular questions I often receive about natively compiled stored procedures is that do we have to learn compiler language or C# or any other machine language? Well, the answer is absolutely no. You can still write natively compiled stored procedures with the help of T-SQL, but you have to specify additional commands so it is compiled to machine code and runs much faster. Natively compiled stored procedures use way less CPU cycle than traditional interpreted T-SQL stored procedures. We will see various demonstrations of its performance later on, but first we will discuss about when to use natively compiled stored procedures in the real world.

Use Cases

When to use natively compiled stored procedures? Let's discuss about this thing. If you have any application where performance is absolutely critical, you may want to consider using natively compiled stored procedures along with memory-optimized tables. As discussed, natively compiled stored procedures directly compiles to machine code, hence it is way faster and will give you amazing performance. There are scenarios with my customers where we have to frequently execute some of the stored procedures. We have used natively compiled stored procedures and replaced traditional interpreted stored procedures. When I go for SQL Server performance unit consultancy, I look for following business critical logic in the application. If I see they're using a lot of aggregation, nested loops, multi-statement procedures, complex expressions or any of the procedural logics, I always suggest that we need to consider using natively compiled stored procedures. Most of the time, clients are a little bit apprehensive using natively compiled procedures because they think they will have to change a lot of things in their traditional interpreted T-SQL stored procedure, but that's not the case. When I explained them that they can keep using that traditional stored procedure, but we have to tweak a few things around it and they can take advantage of amazing performance, they usually agree with it. There are cases where we have got tenfold or even more performance from our stored procedure compared to our traditional stored procedures. However, there are always some cases which are showstopper for us to use natively compiled stored procedures. Let's discuss those technical limitations where natively compiled stored procedures may not be a good fit. Otherwise, if you see your database is using memory optimized tables, you must consider natively compiled stored procedures. Now, let's see some of the limitations.

Limitations

Now let's discuss some of the limitations of natively compiled stored procedures. As I mentioned in the previous clip, I always look for an opportunity to implement natively compiled stored procedures when we are using memory-optimized tables. Now, let's see where a natively compiled stored procedure cannot be used because of technical limitations. One of them is cross database queries and transactions. If you're going to use multiple databases in your query where your transaction is spanning across multiple databases for any operation, memory optimized natively compiled stored procedures may not work there. This is where you will have to use traditional interpreted stored procedures. If you want to truncate your table, you also want to consider using traditional stored procedures because memory-optimized tables would not support this command. Similarly, if you are going to use merge operations, which are very popular in integration services, you may have to look for a workaround. And the final thing which can be heartbreaking for many people is TempDB access. Memory-optimized tables are created in the same database, which is marked for memory optimization. TempDB is definitely a different database, and as I mentioned in point number one, cross-database queries and transactions are not supported by natively compiled stored procedures. Hence, if your query is going to use TempDB, that is technically a cross-database transaction and that is not supported into natively compiled stored procedures. This is one of the limitations which I really wish Microsoft worked in the future version and come up with some workaround. But for the moment, that's definitely going to stop you using these amazing, high-performant stored procedures. Now let's see some of the popular operators which are not supported in natively compiled stored procedures. You cannot use distinct operator or case statement in your natively compiled stored procedure. You have to find a workaround to use either of them. Maybe you can use groupby and can come up with some kind of if-else logic instead of case statement. If you have to access views, you can also not use them into your natively compiled stored procedures. Remember, natively compiled stored procedures are amazingly fast and to make them run very fast, SQL Server has to put limitations on some of the features. Just like view, there are also limitations on using CTE in natively compiled stored procedures. Well, these are some of the keywords which you cannot use in the T-SQL, but you can always come up with some kind of workaround to overcome limitations of operators listed on the screen. There are a few other things also which are not supported by natively compiled stored procedures, but trust me, in your business if you find an opportunity to use this natively compiled stored procedure, definitely use them. Now in the next clip, we will discuss about compilation of natively compiled stored procedures with interpreted Transact-SQL stored procedures.

Comparison of Stored Procedure

Now, let's do a comparison between natively compiled stored procedures and interpreted stored procedures. As we know, natively compiled stored procedures are compiled to machine code whereas interpreted T-SQL stored procedures are interpreted by execution engine. They are understood at very different levels and at different components of SQL Server. Remember, natively compiled stored procedures cannot access this base table, this may be a deal breaker for you if you have to use memory-optimized tables with a disk-based table altogether. In this place, you can definitely use interpreted T-SQL because interpreted T-SQL can use disk-based tables, as well as memory-optimized tables. This is not a workaround, but interpreted T-SQL stored procedures are meant to work with both kinds of tables. They may not give you amazing performance, but they will get your work done and you can still do your traditional optimizations with interpreted T-SQL stored procedures. Natively compiled stored procedures are compiled at the time of the creation, whereas interpreted T-SQL stored procedures are compiled at the first execution. There is a big difference between both of them, and that also leads to a very important concept which I'm going to talk next, and that is parameter sniffing. Natively stored procedures are compiled at the time of the creation; it has no idea what kind of parameter you would pass in to your stored procedure, hence, it is compiled always with the optimize for UNKNOWN keyword. Whereas in case of interpreted T-SQL, parameter sniffing works very differently. It combines at first execution, it also is aware of what are the parameters, and those are the parameters it uses to build an execution plan. Now is the fun stuff, no hash match or hash join operators are available in natively compiled stored procedures. My clients always ask me if a natively compiled stored procedure does not have this operator, how are they going to optimize various T-SQL queries? Let me answer you this. Natively compiled stored procedures may not have these operators, but it has many different workarounds to build an efficient execution plan. You may not trust me by my word, so we will be seeing a demonstration on this topic a little bit later on in this module. Interpreted T-SQL stored procedures can use hash match, hash join along with all of the different kinds of operators. Finally, this is the one thing you may want to remember. Natively compiled stored procedures are always recompiled on your server restart or database start. This also happens if your database comes online the first time or you reboot your entire workstation. That's why whenever you bring back your new database, sometimes natively compiled stored procedures do not immediately work efficiently in subsequent run just like interpreted T-SQL stored procedures, they are always efficient. Remember, interpreted T-SQL stored procedures are also recompiled at the server start or when the plan is evicted from cache. Well, these are the primary differences between both of them. If you're not sure which one to use at some point, here is my single line of advice. Interpreted T-SQL stored procedures will always work for you, it would be your task and job to find where you can use natively compiled stored procedures along with memory-optimized tables and get the best possible performance for your application and business. Next, we will go to a demonstration and see not one, but multiple demonstrations sequentially to understand everything we have discussed so far.

Demo: Natively Compiled Stored Procedures

So far, we have been learning theory about natively compiled stored procedures. Now, we will see our very first demonstration where we will create our first natively compiled stored procedures. Let's switch to SQL Server Management Studio. Here we have SQL Server Management Studio. If you remember in the previous module, we had created a database called Pluralsight. We'll be using the same database once again in this demonstration as well. We have two tables also created in the previous demonstration, which is DiskBasedTable and MemoryOptimizedTable. If you remember, both of them were empty. We will populate certain data in it. The data generation script is over here. First, I will insert around 500, 000 rows into my DiskBasedTable. I will also select the first two lines because I also need to change context of my database from Master to Pluralsight. Now, I will go to the second table, MemoryOptimizedTable, and populate the same table with absolutely similar data. We have inserted 500, 000 rows into this table as well. This was our setup. Next, we will go and create our very first stored procedure. Here we are, and this is our syntax. I'm sure all of you are familiar with interpreted transaction stored procedures. The word interpreted transaction stored procedure stands for the stored procedure which we have been creating for all of our life so far before In-Memory OLTP technology was introduced. Now let's understand which syntax is different from traditional interpreted stored procedures. This line is entirely as it is in both the kinds of stored procedures. The difference from traditional stored procedures and natively compiled stored procedures are these few lines. We have to mention with native compilation, schema binding and execute as owner. If you do not specify any of this word, your natively compiled stored procedure will not work. As a matter of fact, without any of these words, you will be not able to create your stored procedure at all. Right after that, we also have to specify transaction isolation level, which is snapshot. This is something we had discussed in the previous module. The language is your preferred language, and over here it's going to be US English. Let's select this statement and create our very first natively compiled stored procedure. Natively compiled procedures are compiled as soon as you create them, hence, you will notice they will take a little bit more time compared to traditional interpreted T-SQL stored procedures. In our stored procedure, we are retrieving data from our memory-optimized table between the range which we are passing as a parameter to our stored procedure. First, I will enable SET STATISTICS IO and TIME. When I execute this statement, SQL Server will show all the I/O consumption in the message window, and also the time taken to execute the stored procedure. There is a reason why I am enabling this at the beginning of the stored procedure. Now, I will run our very first stored procedure where I'm returning data between 555 and 555. Both of these are absolutely the same number, and looking at my WHERE condition, it should return me a single row. Let's select the statement and click on Execute. Upon execution, there is one result set and going to messages reveals us that this particular stored procedure took 16 ms of CPU time, but it returned as the enter result in merely 7 ms. This is very little time to return us the entire result, and that shows the power of how quickly stored procedure runs. Over here, let's notice that we have specified also STATISTICS IO, but in our messages, we do not have any detail related to I/O because we are retrieving all our data from memory-optimized tables. Memory-optimized table is created in memory and it does not do any I/O related operation. Now as we are retrieving data from this table and our stored procedure is also natively compiled stored procedure, there is absolutely zero I/O related operation when I executed this stored procedure. Now let's execute the second stored procedure and notice how it behaves in terms of I/O and time. There is no I/O and CPU time is just 32 ms. That's very promising as this entire query has taken very little time. Also, elapse time is a little bit higher because we are pretty much returning 99, 000 rows in this result set. The time which SQL Server has taken is just to draw all the results set one row at a time in SQL Server Management Studio. Now going back to editor, we will run our third stored procedure and see how it behaves. This is going to retrieve us much less a row than previously, and among the rows are just 3, 000. Going to Messages, CPU time is next to nothing as elapsed time is 127 ms. In all the three demonstrations, we have noticed that CPU time is just next to nothing because natively compiled stored procedures are amazing to retrieve data from memory-optimized tables. Next, we will create interpreted T-SQL stored procedures and measure I/O and time.

Demo: Interpreted Stored Procedures - Executions

Now, we will continue our previous demonstration. We have seen that when we have executed natively compiled stored procedures, the maximum CPU time was around 32 ms in all the three cases when we executed. There was absolutely no I/O operation at all. We will create interpreted T-SQL stored procedure and retrieve data from our database table. The stored procedure syntax is over here, the SELECT statement is exactly the same as the previous stored procedure. Let's create the stored procedure. The stored procedure was created instantly because natively compiled stored procedures are compiled at the time of execution and interpreted T-SQL stored procedures are compiled when they execute the very first time. In this case, I am going to run this query. So, our interpreted stored procedure will be compiled with the value 555. SQL Server now will build an execution plan based on whatever the first parameter you pass in. I can further optimize this thing by specifying additional keywords like optimize for unknown, but that's out of the scope of this course. Let's run this stored procedure and see how much I/O and time it takes. Upon execution, there is a single row, and going to Messages, there is a logical grid of four pages, which is very little, and CPU time is again next to nothing. That means both the stored procedures gives us equal performance and we are to retrieve very little data. Remember, this will change when I run our second test where I have a large amount of the data. Upon executing the stored procedure, the logical reads are much higher than before, previously with natively compiled stored procedure which was just 0, and now we have 1629 pages. Each page is of size 8 K; you can calculate the size of it by your own. Also, SQL Server has to pass and compile the stored procedure where it took around 155 ms and for execution it has taken 141 ms. You must pay attention to elapsed time as well, which was around 1339 ms. Both of these numbers are much higher than what we have seen in natively compiled stored procedure. Natively compiled stored procedures don't have to deal with a lot of other issues which interpreted T-SQL stored procedure has to deal with. There're like locking, latches, and any of the isolation related issues. So now when you have to retrieve a large amount of the data, natively compiled stored procedure is definitely a good choice when we are using memory-optimized tables. To complete our demonstration, I will run the stored procedure, which retrieves only a few rows. The query runs instantly, and when we go to Messages, logical reads are still very, very high. The logical reads are 1629 because the entire table is of this size and it seems like the case that SQL Server might be scanning the table. Now, we have created our very first natively compiled stored procedure. Let's see some of the errors which are related to natively compiled stored procedures.

Demo: Errors and Resolutions

In the previous clip, we have learned how to create our very first natively compiled store procedure. And in this clip, we are going to learn about errors related to natively compiled stored procedures. I've seen in the industry that many of my clients want to get started with natively compiled stored procedures, but as soon as they start creating them, they get very quickly frustrated with various errors related to creating their very first natively compiled stored procedure. In this clip, I'm specifically going to focus on things you must avoid or things you must include when you create your natively compiled stored procedure. Let's go to SQL Server Management Studio and try a few things out. Here we are in the SQL Server Management Studio, and on the screen we have the same stored procedure which we have created in the previous module. Here we have specific commands which are related to natively compiled stored procedures. If any of this is missing, you are going to get various errors when you create it. First, we will comment out the word NATIVE_COMPILATION. When you comment in this one out, IntelliSense doesn't show us any error, but when we attempt to create the stored procedure, at the time we see an error that SCHEMABINDING option is supported only for natively compiled modules and it's absolutely required. This is a very descriptive error message and you should go back and now include NATIVE_COMPILATION keyword. Similarly, if you comment out on SCHEMABINDING, IntelliSense has no error, but when you try to create the stored procedure, similar error and it's going to tell you that you need to have SCHEMABINDING option. Next, you can also attempt to comment out EXECUTE AS OWNER and see what happens. It's interesting when you run this stored procedure, it will compile just fine. So, EXECUTE AS OWNER is not one of the keywords, which you must have it, but trust me on it, I work with many customers and many clients. Without this particular keyword, you may end up in the permission-related issue of executing natively compiled stored procedures. Now is the part where we are going to talk about the keyword ATOMIC. ATOMIC is also important because natively compiled stored procedures have to follow the asset principles. When you comment that out, you can clearly see there is an error with the help of IntelliSense and it also demonstrates that there is something wrong with this stored procedure. Upon executing it that it's not permitted and you need to have ATOMIC block in your natively compiled module. So let's go back and now include ATOMIC back once again. Next is transaction isolation level. This is also very, very critical, and without it, you will also face an error. Let's execute the statement and see how it works. There you go. It says if you're going to use BEGIN ATOMIC, you need to also have transaction isolation level as a part of it, otherwise you will face an error. I really love how Microsoft has done good work in describing each of the errors and guiding us what we should do to fix that. Now the final one is language. Let's comment this one out and see what happens. You will find that when you do not specify any language, natively compiled stored procedures, once again, cries for that particular keyword. With close, with BEGIN ATOMIC must have option of the language, otherwise it will not compile. You need pretty much each of these blocks to get going with your natively compiled stored procedure. I want to make sure that none of you ever get frustrated when you start with natively compiled stored procedures. Hence, this particular clip, where we discuss errors in detail. Next, I will be talking about natively compiled stored procedures and joins.

Demo: Joins and Natively Compiled Stored Procedures

In this clip, we are going to see a demonstration related to natively compiled stored procedures and the joins. Whenever I go for SQL server performance tuning consultation with my clients and I propose natively compiled stored procedures, the very first argument I get back is that natively compiled stored procedures do not support certain kinds of keywords and certain kinds of joins. The biggest concern usually DBAs show is about hash join. Natively compiled stored procedures do not support hash join. Well, that's absolutely true, but in this demonstration we are going to see that even though natively compiled stored procedures do not support hash join, it has its own workaround behind the scene and natively compiled stored procedures with memory-optimized tables are much faster than interpreted T-SQL stored procedures on disk space tables. Let's switch to SQL Server Management Studio and see that demonstration. Here we are in SQL Server Management Studio and we are going to create two identical stored procedures with different tables in it. The very first stored procedure we are going to create is natively compiled stored procedures, and this stored procedure uses memory-optimized tables joined with another memory-optimized table over here. We are retrieving first name and last name from each of these tables. Now, let's execute the statement. Next, we will create interpreted T-SQL stored procedure on the DiskBasedTable. The business logic is absolutely identical. Now, we will go and enable statistics time and I/O. Once we enable this, we will run both of the stored procedures one after another. First, we will run natively compiled stored procedure and see how long it takes in terms of I/O and CPU. There is a lot of data, hence the query took some amount of the time to execute. I will go to

Messages, and here is the time which I will remember. I remember that CPU time was 375 ms and elapsed time was over 5 seconds. Please note that elapse time is really important, but not as much as CPU time. Because SQL Server Management Studio took around 5 seconds to draw each line of these 500, 000 rows. We will go to our second case where we are going to run interpreted stored procedures. When you execute this stored procedure, it will take some time. On the surface, the query took the same 5 seconds of time and it has retrieved around 500, 000 rows. Next, we'll go to Messages to check CPU time and elapsed time. Elapsed time is pretty much the same; CPU time is quite high. Here, query took almost 2 seconds to run in terms of CPU time. CPU time is nearly eight times more than the previous case. The reason is very simple. There is a lot of I/O activity going on in this stored procedure. It has to retrieve a lot of data from DiskBasedTable and the logical read is over here; 3, 252 is number of the page, and if you want to understand how much data it is, you just have to multiply with 8 K and that's your answer. It is very clear that when you are using interpreted T-SQL stored procedures, they usually take more time in terms of CPU. Now, let's dig a little bit deeper with the help of execution plan and understand what is going on behind the scenes. If you want to see an execution plan of natively compiled procedures, you cannot see that by just executing query with actual execution plan. Let me demonstrate that to you very quickly. And now run natively compiled stored procedures. There is no tab for actual execution plan even though the query was completed. If you want to see the execution plan of natively compiled stored procedures, you just have to either see estimated execution plan or you can enable option of SET SHOWPLAN_XML ON right before you execute this stored procedure. Let's execute the stored procedure with SHOWPLAN_XML ON. In the execution plan, this particular query is using nested loops. Nested loops as a very bad name for a very slow operation, and I totally agree with you if you have to retrieve a large amount of the data and you are using DiskBasedTable with interpreted stored procedure. But, if you are using natively compiled stored procedure, it takes a fraction of the CPU to run this nested low. Actual execution plan of your interpreted stored procedure is using very different operator in execution plan, which is Hash Match, and it also uses the power of Parallelism. With the help of Parallelism and Hash Match, the query still runs pretty heavy and reads a lot of data and CPU time is still tired. I hope this demonstration conveys an important message that you need to look for every single opportunity in your database where you can implement natively compiled stored procedures and memory-optimized tables. Let's see a quick summary about what we have learned so far.

Summary

Let's summarize this entire module. And the summary of this module is a single line statement. Natively compiled stored procedures get best performance from memory-optimized tables. Yes, that's it. As a DBA, you need to look for every single opportunity in your database where you can implement memory-optimized tables. Once you implement memory-optimized tables, the next task is very simple. You need to rewrite or write your stored procedure in such a way that you only access your memory-optimized table in that piece of logic. Once you create such logic, you can implement that with the help of natively compiled stored procedures and get the maximum out of your application. In the last clip we have noticed that execution plan of natively compiled stored procedure was not available to us easily and we have to use alternate method to retrieve that. That's because this stored procedure works very, very differently from interpreted T-SQL stored procedures. In the next module, we'll focus on understanding how to collect various execution statistics of natively compiled stored procedures and how we can measure them with the help of DMVs.

Collecting Execution Statistics for Natively Compiled Stored Procedure

Introduction

Hi. This is Pinal Dave, and I welcome you to the module, Collecting Execution Statistics for Natively Compiled Stored Procedures. This is a little bit shorter module than the other modules, but one of the very crucial modules because whatever you do with natively compiled stored procedures, this is the place where you can measure it, as well as get some of the very crucial details about them. Let's see what we are going to cover in this module. First, we will talk about what are the various challenges related to natively compiled stored procedures and their execution statistics. After that, resolutions. Following that, two demonstrations back to back where first we will see the challenges, and right after that, the resolution. This is our agenda. Now, let's jump to various challenges related to natively compiled stored procedures.

Challenges

Let's see some of the challenges related to collecting execution statistics for natively compiled stored procedures. Remember, performance is one of the most important goals for natively compiled stored procedures. To get maximum performance from natively compiled stored procedures, Microsoft has done some of the adjustments. One of them you are already familiar with it is there is no actual execution plan when you run natively compiled stored procedures. You can only get estimated execution plan, and that also you have to specify explicitly to see it. This is good in one way because building actual execution plans also takes little bit of time, whereas for natively compiled stored procedures, every single millisecond is very crucial. They want to give it to SQL Server Engine to give us the best possible performance. For the same reason, traditional DMVs don't collect execution statistics by default for natively compiled stored procedures. As a matter of fact, you will not see anything about natively compiled stored procedures in traditional DMVs. Often people get surprised when they run dm_exec_query_stats or dm_exec_procedure_stats and see no details about natively compiled stored procedures. One of my clients was really frustrated with this situation because they wanted to check how fast and how frequent their natively compiled stored procedures were running, but they were not able to see anything in these DMVs which are very critical for performance tuning. Well, these are just challenges, but let me tell you, the solution or resolution is even simpler. Let's see them in the next clip.

Resolutions

In the previous clip, we have discussed some of the challenges related to statistics collection for natively compiled stored procedures. In this clip, we will see resolution of the same. As discussed, traditional DMVs don't collect statistics by default for natively compiled stored procedures, but we can always explicitly enable statistics collection by enabling them with the help of stored procedures. Here is one of the stored procedures, which enables collection at procedural level for natively compiled stored procedures, and here is another stored procedure. It helps us to enable even further granule detail related to query from the stored procedures as well. With the help of these two stored procedures, we can easily collect all the necessary statistics for natively compiled stored procedures. However, it's important for you to understand that any statistics collection for natively compiled stored procedures comes with a very little performance hit. That's the reason they were disabled from the beginning. By enabling them, you may slow down a little bit your natively compiled stored procedure, but you can collect the necessary data which you need to tune your queries. Here is one more thing you need to learn. These two stored procedures also give us finer control on what exactly we want to capture. We do not have to enable statistics collection for the entire server. You can specify that you want to enable this at the database level, or you can even further go down and you can say you want to enable statistics collection for only certain queries as well. This kind of finer control enables you to get the maximum out of your natively compiled stored procedures. Well, that's where our theory ends, and in the next clip we will see collection of statistics in action with the help of SQL Server Management Studio.

Demo: Enabling Execution Statistics

Let's see our very first demonstration where that collection of execution statistics is disabled by default for natively compiled stored procedures, and we will learn how we can quickly enable them with the help of a couple of stored procedures. Let's go to SQL Server Management Studio next. Here we are in SQL Server Management Studio. There are two levels of statistic collections possible in SQL Server with related to natively compiled stored procedures. First is at procedure level, and the second one is at query level. Let's first understand how we can enable procedure level statistics collection. This is the stored procedure, which helps us to enable statistics collection, as well as it is the same one which helps us to understand if statistics collection is already enabled or not. First, we will change our database context to Pluralsight, and next we will execute these three statements. Over here, we are declaring a variable which we are passing as the output variable for this stored procedure, which will get us the status of statistics collection at procedural level. When you execute it, it tells us that current status of statistics collection at the procedure level is disabled as the value is 0. Now, we can enable it by just executing the same procedure with the different parameter new_collection_value is equal to 1. Let's execute that. Let's check the value executing the same script we ran earlier. When we execute it, the current status of statistic collection is set to 1. If you are using Azure, you can enable statistic collection at the procedure level by executing the statement which is displayed on the screen. The syntax for enabling statistics collection is a little bit different because on Azure it uses scoped configuration and you can only access that from script as displayed on the screen. Now, query level statistics collection, the script is pretty identical. The only change between what we have just seen and this script is name of the stored procedure. Name of the stored procedure is different and the rest of the content is the same. When you execute this statement, we will get current status of statistics collection at the query level. It is set to 0. Now, let's execute this stored procedure and make it 1. Once we run it successfully, when we run the same query again, the current status of statistics collection is set to 1. Our natively compiled stored procedure is tracked at the query level as well. And here is one more thing. If you are using Azure, you can do the same task on Azure by enabling scoped configuration at the query execution level. Syntax is a little bit different for Azure, but it's not so difficult to remember. Now, we have enabled collection of execution statistics for natively compiled stored procedures at the stored procedure level, as well as query level. In the next clip, we will execute some of the stored procedures which we have created in the previous module and collect their statistics.

Demo: Collecting Execution Statistics

In the previous clip we have seen how we can enable statistics collection for natively compiled stored procedures, and in this module we will go and actually collect the execution statistics for them and understand a little bit more about how they work. Let's switch to SQL Server Management Studio next. Here we are in SQL Server Management Studio, and as I mentioned we'll be using the stored procedure which we had created earlier. If you remember, in the previous clip I was retrieving data from memory optimized tables for different range. I will create identical three stored procedure and will name them NativeSP1, NativeSP2, and NativeSP3. The reason is that I want to collect data for all three stored procedures. The next thing is to run all these three stored procedures, but I will run them a different number of times. The first stored procedure I will run six times, the second stored procedure I will run four times, and the third stored procedure I will run it two times. Let's execute them all together. That's it, we are done running all these stored procedures a different number of times. Now, we will run DMV-based queries. First, we will run on dm_execute_procedure_stats. This will give us details for our stored procedures, which we have just created. Let's select the statement and execute and see what kind of details it provides us. Upon execution, you can see NativeSP1 was executed six times, NativeSP2 was executed four times, and NativeSP3 was executed two times. Here are the object IDs and this is the time when it was actually cached. The last execution time is also displayed over here, and here are a few of the additional important details. Total worker time stands for the CPU time for executing this query four times. Here is the last worker time. That means individual query execution time when it was executed at this moment. Here is minimum worker time for executing this stored procedure out of its four execution, and here is maximum worker time with a similar definition. The total elapsed time is also listed over here for all the four queries, and here are last elapsed time, minimum elapsed time, and max elapsed time. The name of these columns are self-explanatory. Well, this is about stored procedures and stored procedure level details. Now, we will get details at the query execution level from another dynamic management view. Over here, we have cross showing dm_exec_query_stats with dm_exec_sql_text and got additional detail about queries which are running inside these stored procedures. When you execute this query, the data is very similar, but now, we also have visibility inside the query text. The query text is listed over here along with object name and their object ID. As you know, we had created three identical stored procedures, hence, the query text is absolutely the same. The detail which you see over here is now specific to the queries which ran inside the stored procedure, and they may be a little bit different from procedure level details. For example, let's turn off query execution status and only leave statistic collection at the stored procedure level. Let's run this query, and now current status of statistics collection at query level is disabled. In this scenario, when I run these queries again for the same number of time, you will quickly notice when I bring the result set there is a difference between what SQL Server has collected. Let's run both of these collection procedures together. Now, here is the result set. From the result set, the execution count for NativeSP2 is now twice, or double, from what we have seen before, whereas for query level collection it is still saying 4, 2, and 6. Here, they are 8, 4, and 12 because we have collected procedure level statistics, but disabled query level statistics. Hence, these numbers are no longer increasing. I hope now it is clear why we have two different stored procedures provided by Microsoft to collect statistics for natively compiled stored procedures. Here is last tip. If you mouse over the stored procedure, which enables collection for query, there is an option to specify database ID, as well as object ID. By specifying them, you can only enable statistics collection for one particular stored procedure as well. That's what I wanted to discuss in this demonstration. Let's quickly summarize this entire module in the next clip.

Summary

Here is our final clip of this module, and let's summarize our module very quickly. First of all, we must accept that natively compiled stored procedures are performance-focused objects. They were built to get maximum performance from our SQL Server. Hence, by default, execution statistics collection was disabled. We can easily configure them to collect various granular details of natively compiled stored procedures. If you do not need execution statistics, I strongly suggest that you leave them disabled and get the maximum out of your memory-optimized tables and natively compiled stored procedures. Well here we complete this module, and in the next module we will see a final summary of what we have learned in this course, as well as where you can go next from here. Also, I have kept a very small demonstration in the summary module as well, which will help you to sell this concept to your manager, as well as your colleagues. See you in the final module of summary.

Summary

Agenda

Hi. This is Pinal Dave, and in this module, we will summarize entire course which you have learned today. However, I don't want our summary to be just a plain old summary where I discuss what I have covered before. So, I'm going to show you in this module, along with what we will cover, an interesting demo which definitely will convince your boss that In-Memory OLTP is the way you'll want to move forward, and also will give you enough ammo to fight with anybody who says In-Memory OLTP is not a successful technology. Before we go to a demonstration, let's start with what we have covered in this module in the next clip.

In-Memory OLTP

Let's summarize this entire course. We have seen so far that In-Memory OLTP technology is very fast. All the operations are done in-memory and there is very little I/O, hence, In-Memory OLTP always performs best. One of the biggest misconceptions with In-Memory OLTP is that it's not durable because it's in-memory. Well, that's not true. Everything about In-Memory OLTP is 100% durable and technology does not support data laws. All the data is eventually moved to your disk for persistency and durability. Lots of people think that you cannot take advantage of this technology if you have not enabled In-Memory OLTP for your database from the beginning. Well, that's not true. You can take any database and make it capable to use In-Memory OLTP. It is also very easy to convert any disk space table to in-memory optimized table with the help of few keywords. As we discussed and seen in various demonstrations, there is almost little or no disk I/O, hence, In-Memory OLTP gives us amazing performance. If this was not enough, In-Memory OLTP further supports nonclustered and hash indexes. With the help of indexes, you can even get more performance from your already memory-optimized tables. I love this feature so much that I look every single opportunity to implement them if I spot memory-optimized tables at customer's place. Finally, we have natively compiled modules which takes very less of CPU cycle then traditional interpreted T-SQL modules. They are even faster to modify data into your memory-optimized tables. When you look at what all In-Memory OLTP technology has to offer, it's impressive. If you are a performance tuning expert, I strongly encourage that you look for opportunities where you can implement this technology and give best to your client or your organization. So far, we have talked about theory. Now, I'm going to show you a demonstration which I present to my customer who is still not convinced to use in-memory technology. Let's go to a demonstration in the next clip.

Demo: Convince Your Boss

Now let's see a demonstration. I go to many customer places and I try to explain to them that they should start using In-Memory OLTP, memory-optimized tables, and natively compiled stored procedures. I usually get a pushback saying they get the idea that they retrieve the data faster, but they have quite a lot of workload which is later to insert, and they're very much worried with the data laws and slow performance of insert. Now, let's jump to SQL Server Management Studio to prove that not only select, but even inserts are much faster with In-Memory OLTP technology. Here we are in SQL Server Management Studio, and I'm going to create a brand-new database. I have enabled SET STATISTICS IO on over here, and next I will create this database where I held FILEGROUP, which is memory optimized. Now, let's execute this statement and create our database. Once the database is created successfully, our next task is to create two tables. First table I like to call is a dummy table and it's just a regular disk-based table. Next, I'll be creating DummyTable with _MEM, which stands for memory, but I will make it memory optimized by specifying these additional keywords. I'm also creating PRIMARY KEY NONCLUSTERED with a hash and BUCKET_COUNT as this large value. Let's execute this statement, and once the table is created, we will create two more stored procedures. First stored procedure is a regular stored procedure which will be inserting some amount of data into my DummyTable. So, this is interpreted T-SQL stored procedure inserting data into disk-based table. Let's create this procedure by executing the statement. Next, I will create natively compiled, memory optimized stored procedure. This stored procedure also will do exactly the same task, but this time it will insert data into memory optimized table. Let's create this stored procedure as well. Now, I have created two tables and two stored procedures. One thing you should notice in the stored procedure is that at the beginning of the stored procedure, I am capturing the start time. Right after all the operations are done, I take difference of the start time with current time and print how much time this stored procedure has taken. This I have done in this memory optimized natively compiled stored procedure, as well as an interpreted T-SQL stored procedure before I started this. Let's check how much data I held in both of these tables. When you run count start, tables are currently empty. We will run both of these stored procedures. First, I will run simple stored procedure, which is interpreted T-SQL stored procedure. When I run this stored procedure, it will take a certain amount of the time and we will see how long it takes at the end of the execution. The time taken by this stored procedure is around 28 seconds. That's a lot of time, but maybe that's needed because SQL Server has to do quite a lot of operation about inserting the data. Now, coming back over here, we will insert the same amount of the data into our in-memory, optimized, natively compiled stored procedure. Let's see how long this stored procedure takes. As soon as I started the stored procedure, it immediately completed. The time taken by my stored procedure is almost 0 seconds. Well, nothing runs in 0 seconds, but you got the point that query takes relatively very little time. When I go to Messages, I don't see anything over here, which I have seen in earlier case. This is because there is absolutely no I/O related operation, hence, there is nothing to display over here. All the data, which we inserted, were inserted into memory. Now here, many of my clients often ask that if the second stored procedure really inserted data, or it just run without doing anything. To check that, we will count the data from DummyTable, as well as DummyTable_Mem, with the help of COUNT *. Let's see what these queries return. There are 100, 000 records in both of these tables, so definitely both of the stored procedures had done the task, which we assigned to them. However, the first stored procedure, which is interpreted T-SQL stored procedure, took quite a lot of time compared to natively compiled stored procedures when I have to do insert. This can change how you load your data in SSIS, as well as many other places where you are to do large amount of the data operation. When I show that SELECTs are faster in In-Memory OLTP technology, my customers are impressed, but when I really show them this particular demonstration, I've seen them getting really excited to try this technology and see how they can improve their applications or performance. Here is the final clean up code. You can execute this and drop the database, as well as table, which you had created earlier. This entire demonstration works independently and you can use this to convince your boss, coworker, or your clients. Let's go to our final slide in the next clip and see what you can do next.

Final Note

Here is the final clip of this entire course. I'm going to give you two keywords, which you must remember. First is In-Memory OLTP and Microsoft. If you go to Google, Bing, or any of your favorite search engines, if you search with this keyword, you'll end up on page which is a very important documentation resource from Microsoft. I frequently refer to this page because Microsoft keeps on modifying it as and when they release new enhancements in this technology. Lots of people also ask me if really In-Memory OLTP data is durable. At that point of time, I always search for in-memory optimization and Microsoft and also show them the very first link, which talks about internals of this technology and how data is going to be durable. The reason I'm not providing you any link right now, but giving you a keyword is quite frequently I've seen that links ends up on a 404 if underlying data or navigation structure changes. Now, here is my final piece of advice if you are using In-Memory OLTP. Make sure that you take your backups and test your restores. I strongly encourage that you frequently take your backups and test them to make sure your backup works. Well, with this we are almost at the end of this course. If you have any question about any of the topic, which we discussed, or want to discuss further SQL Server performance tuning, you can always post questions in the Pluralsight forums or you can reach out to me at pinal@ sqlauthority.com and I will be happy to help you. Now is the time when I say thank you very much and have a good day. I hope this course helps you to start your journey with In-Memory OLTP technology.