Course Overview

Course Overview

Hi everyone. My name is Gerald Britton, and welcome to my course, Managing SQL Server Database Concurrency. I'm a senior solutions designer at TD Bank in Canada. I've been designing and troubleshooting SQL Server databases for more than 10 years, and love to help others with the next steps on their own journey. I have also been a Pluralsight author since 2016. Have you ever wondered how a busy database system manages to handle thousands of queries at once without ever messing up the business critical data it holds? Or have you ever seen or even written a seemingly simple query that somehow manages to bring the system to its knees? Maybe you just want to know how things really work, and the things you can do to make them work better. If so, this course is for you. In this course, we are going to explore the methods SQL Server uses to manage concurrent activity while preserving database integrity, and see the many things a query writer can do to maximize concurrency while balancing critical business needs. Some of the major topics we will cover include transactional isolation using the four ANSI isolation levels, row versioning using SQL Server's snapshot isolation, locking in the database engine, troubleshooting deadlocks, and writing efficient queries while maximizing concurrency. By the end of this course, you'll know how SQL Server uses locking to preserve database integrity, how to troubleshoot locking and deadlocks, and how to leverage row versioning, and how to write queries with concurrency in mind. Before beginning the course, you should be familiar with writing queries in Transact SQL and have an IDE, such as Azure Data Studio or SSMS installed and ready to go. From here, you should feel comfortable diving into advanced SQL Server topics with courses on creating stored procedures, functions, and triggers, Extended Events, and using XML in SQL Server. I hope you'll join me on this journey to learn the ins and outs of SQL Server operations with the Managing SQL Server Database Concurrency course at Pluralsight.

Introducing SQL Server Concurrency Control

Version Check

Introducing Concurrency

Hello. Welcome to the course, Managing SQL Server Database Concurrency. My name is Gerald Britton. I'm a senior IT solutions designer, SQL Server specialist, and Pluralsight author. This course dives deeply into the defaults, options, and tradeoffs concerning concurrent access to SQL Server. Along the way, I'll work through several demos to show different approaches to concurrency and their effects on results and performance. Before I get ahead of myself, though, I think I should answer the unspoken question. What do I mean by concurrency? Where I live, city traffic sometimes looks like this, 2 levels, 10 or more lanes, multiple entrance and exit ramps, full of cars, and trucks, and buses going nowhere fast. This highway is, however, a great example of concurrent usage. If we just count the major sections, there are four lane groups, each with two or three lanes that are able to function independently, more or less. Of course, at some point, drivers in the inner lanes will want to exit, and drivers in the outer lanes will want to switch to the inner lanes. Not an easy thing to do in this picture, yet the hundreds of vehicles in the picture, and the thousands more you can't see, all use the same highway, concurrently. For traffic, concurrency is controlled by speed limits, restricted access, signage, and of course, the laws of the road. If you've ever been stuck in a situation like this one, you know that it doesn't always work perfectly. Nevertheless, there's an overriding principle. People need to be able to share the road and eventually get from point A to point B safely. This is a great analogy to concurrency in computing systems in general, and SQL Server in particular.

Concurrency in Computing Systems

A modern computing system is a busy place. You will find one or more processors, network cards, hard drives of some type, user interfaces, and various internal clocks all clamoring for attention. In fact, each of these individual devices have processors of their own dedicated to their particular jobs. On the other hand, the main CPU must be able to coordinate all these auxiliary devices, and yet still find time to do it's own work. Now while it would be easy to allow only one thing to run at a time, that would not let the CPU handle interrupts in a timely fashion or make good use of the powerful resources available. So, what's the solution? Most operating systems use a combination of preemptive and cooperative multitasking. Preemptive means that the system is interruptible. When some device or event asks for attention, the OS handles it as soon as possible. Cooperative multitasking, on the other hand, means

that processes running on the system voluntarily yield control of the CPU, usually in a time-based fashion. SQL Server is an example of just such a system. Under the covers, the SQL operating system, or SQL OS for short, starts up sessions, which run cooperatively, yielding control at regular intervals. On the preemptive side, it handles interrupts as they come in, notifying waiting sessions that some unit of work is ready. SQL OS works hard to squeeze the maximum performance out of the system components it controls. A deeper discussion is outside the scope of this course, and really is a subject for true nerds.

Understanding ACID

Concurrent operation has big implications for data integrity. In order to guarantee integrity, a relational database system must comply with the ACID principle. This is normally implemented through transactions. In ACID, A stands for Atomicity, and that means when data is modified by some transaction, either all of its data modifications are performed or none of them are performed. The C in ACID stands for Consistency. When a data modification transaction completes, all data must be in a consistent state. All constraints must be satisfied, and all internal structures must be correct. I is all about Isolation. Modifications made by one transaction must be isolated from those made by other concurrent transactions. This also implies that if you redo the same operations with the same starting data, the results will always be the same. The D in ACID stands for Durability. When a transaction is complete the results are stored permanently in the system and persist even if a system failure occurs. Now in order to get you set up, so that you can follow along with all the demos, let me introduce you to the tools I'll be using.

Getting the Tools Installed

First, let me point out this URL. It will open a gist on GitHub that contains all the links for this course. It's probably a good idea to bookmark it now and keep it handy as you move through this course. I'll keep adding to it as more topics and resources unfold, so that any screenshots I use may look a little different from module to module, and from what you see when you open it. For the demos in this course, I'll be using Azure Data Studio, a free cross-platform IDE for developing SQL Server applications. At the time of writing, it could be found at the bit.ly link shown. If, by chance, the page has moved, your favorite search engine should be able to find it quickly. I suggest you take a moment and install it on your own machine before continuing. I also suggest that you install either SQL Express or SQL Developer, both accessible from the bit.ly link shown, and easy to find otherwise. I'll be using LocalDB, which comes with SQL Express and is quite lightweight. Alternatively, you can download Docker containers for SQL Developer and SQL Express at the link shown, or just search for them on Docker Hub. I'll launch Azure Data Studio, so you can see its major components. The default screen is quite spartan. There are a few icons down the left hand side and a little help section in the middle showing a few hotkeys. The Ctrl+G hotkey corresponds with the topmost icon, Show Servers. Opening that shows no server names listed, so let's add one. I'll click on the Add button for a

new connection, and a dialog opens up. I mentioned I'll be using LocalDB, so I'll just paste that connection information in here, and hit the Connect button. The connection is successful, and I can see the database. Also, Data Studio has opened the server dashboard, which shows some basic information and reveals that I am running the 64-bit Express edition of SQL Server on Windows 10. You will, no doubt, see slightly different details. You should take your time and explore the tabs and actions here. For now, I'll just click on New Query, which brings up a second tab with an Edit window. I'll just write a simple query here to show that it works, and run it. There. Down below, the Results panel shows, well, the result set, and there's also a Messages panel at the bottom where some statistics and any error messages would appear. The third icon down opens the Explorer view. Right now it says, no folder opened, so I'll open one. I'll just use this one right here. It gives me the option now to save my query, so I'll do that. So I've saved my little query, and you can see that it now appears in the Explorer view. There are many more options and capabilities, and a whole range of extensions available. I'll let you discover those on your own. We have enough now to get going.

Upcoming Modules

The 35, 000 foot level view of this course looks like this. I'll start with a discussion of transactions, and touch on topics including the explicit and implicit transactions, safe points, and the function of autocommit. Next, I'll talk about the basic isolation levels, that is, those in the ANSI standard, how to implement them, and what effect they have on concurrency and query results. Then I'll turn my attention to snapshot isolation, an alternative approach to transaction isolation using row versioning. With that material covered, I'll dig into locking in the database engine and how it is used to provide transaction isolation. That will lead naturally into a discussion about optimizing concurrency and locking. There, you'll learn about deadlocks, how to capture, analyze, and remediate them. Also, you'll see some tips for writing high performance but concurrency friendly queries. In each module there will be demos showing how to implement each isolation level and see the effect on query results and other active queries. Let's get started.

Understanding Transactions

Introducing Transactions

Hello. Welcome back to the course, Managing SQL Server Database Concurrency. My name is Gerald Britton. In the previous module I discussed the ACID principle, the idea that the work done by database transactions must be atomic, consistent, isolated, and durable. In this module I'll begin to look at this in greater detail. In particular, you'll want to see how to start and stop transactions and who is responsible for what. Hint: you have a big part to play. The first thing to be covered is understanding responsibilities. Who plays what part? What do you need to do? What does the SQL Server database engine need to do? Then you'll learn all about controlling transactions, that is, how transactions start and stop, and what to do about errors. There are different types of transactions available to the SQL Server developer. I'll cover these types and show the most important ones in the demos. A user can set a safe point or a marker within a transaction. The safe point defines a location to which a transaction can return if part of the transaction is cancelled for some reason. I'll do a special demo on that facility, and since transactions are implemented using locks, you need to understand locking basics before we go further. Now let's look at responsibilities.

Understanding Roles and Responsibilities

The two main actors in transaction processing are SQL programmers, that's you, and SQL Server itself, specifically the database engine. SQL programmers are responsible for starting and ending transactions at points that enforce the logical consistency of the data. Note that that is more than just obeying some form of normalization or making sure that all constraints are satisfied. It's really about the business rules that are in place for the organization using the database, and SQL Server doesn't know those. Although, perhaps in the future artificial intelligence enhanced versions it will. So the programmer has to build transactions that complete a logical unit of work with respect to those rules. The SQL Server database engine is responsible for ensuring the physical integrity of each transaction, and that includes locking facilities that preserve transaction isolation, logging facilities that ensure transaction durability, even if the server hardware, operating system, or the instance of the database engine itself fails, it must be able to rollback any uncompleted work when the system restarts. It uses the transaction log for that. Transaction management has features that enforce transaction atomicity and consistency. After a transaction has started, it must be successfully completed or else all changes must be undone to return the database to the same state it was in before the transaction began. Now let's see how to code up a simple transaction in the first demo.

Demo: Autocommit

In Data Studio I've connected to my LocalDB instance and opened a little query. You might be wondering what version of SQL Server I'm running. You can always run this command to find out. When I run it the results show that I am running SQL Server 2016 Service Pack 2; the Express edition. The internal version is 13, which is equivalent to SQL Server 2016, and I am running it in 64-bit mode, which is actually the only way you can run this version. SQL Server stopped supporting 32-bit operations starting with 2016. The numbers after the version indicate the current cumulative update level of this instance. You can usually run this command on any SQL Server instance you can connect to, and it's a good starting point to see what you've got. By the way, @@VERSION is one of many system configuration functions that you can use to see how the system is set up. There's a link to these functions in the gist I mentioned in the first module. Now here's a script that I'll use to run the first transaction. The query runs fine, but where's the transaction? Well, you see, out of the box SQL Server runs in autocommit transaction mode. In this mode, each individual statement is a transaction. That means that the simple SELECT statement you see here automatically begins a transaction, runs the query, then commits the transaction. But what if something goes wrong? In autocommit mode, if an error occurs, the transaction is rolled back. Of course, this statement only reads data, so there would be no changes to roll back. You may be wondering what the point is to wrapping a simple query in a transaction, and that's a great question. We'll be digging into that question, and to many related ones, as this course progresses. For now, imagine that there is a second session open to SQL Server, and that it is updating the table I'm reading. Think about what could happen if that session inserts new rows or deletes some rows that this simple query is trying to read, and what would you want to happen?

Demo: Explicit Transactions

Now let me show you a different script. This one uses an Explicit transaction. These always begin with a BEGIN TRANSACTION statement and then with either a COMMIT or ROLLBACK statement. Oh, and note that there is no END TRANSACTION statement. I just get a syntax error on that line of code. COMMIT says to write the database changes back to persistent storage. ROLLBACK says to undo any changes made, so that the affected tables are in the same state as at the start of the transaction. Since this is a simple, non-updating query, it makes no difference if I commit or roll back, however, ROLLBACK is usually used as part of error handling in complex transactions. So using COMMIT here is the right choice, and follows normal conventions. Running this transaction produces the same results as before. However, because this is an explicit transaction, I can add something to it that will be useful later on. I'll add the command, SELECT @@TRANCOUNT inside the transaction and after the COMMIT. This system function returns the number of BEGIN TRANSACTION statements that have occurred and are still active on the current connection. Each BEGIN TRANSACTION increments this number by one, and the COMMIT statement decrements the transaction count by one. In case you're wondering, transactions can be nested. This script shows

how the transaction count changes as new transactions are begun and committed. The results show this behavior. Now what if I changed that inner COMMIT to a ROLLBACK? Let's try that. Oops, that's an error. You see, ROLLBACK is different than COMMIT. When nesting transactions, this single statement rolls back all inner transactions, all the way to the outermost BEGIN TRANSACTION, and then sets the transaction counter to 0. This is the standard behavior of ROLLBACK, though there are some nuances we need to look at.

Demo: Implicit Transactions

So you've seen autocommit and explicit transactions. Well, what about implicit transactions? In implicit transaction mode SQL Server automatically starts a new transaction after the current transaction is committed or rolled back. You don't do anything at the start of a transaction. You only commit or roll back each transaction. In this way, you get a continuous chain of transactions. Now since autocommit mode is the default, you have to explicitly enable implicit transaction mode. In T-SQL this is done with the SET IMPLICIT_TRANSACTIONS ON statement. I'll use this little script to demonstrate implicit transactions. First off, here is a simple way to see if they are currently enabled. The @@OPTIONS system function returns the current options from the SET command as a 32-bit integer, where each bit represents the status of each option. There are currently 15 SET options, so we're not likely to run out of bits any time soon. As it turns out, the second bit tells us if ImplicitTransactions are on. This statement then does a binary AND to see if this bit is set to one. If so, ImplicitTransactions are in effect, otherwise, not. Then I turn IMPLICIT_TRANSACTIONS ON. After that, I do a SELECT and a COMMIT. At each step I query the transaction count and finally, I SET IMPLICIT_TRANSACTIONS OFF. Running this, you can see the effects. At the start of processing, ImplicitTransactions are OFF and the current transaction count is 0. After I set ImplicitTransactions ON and run my query, the transaction count has incremented, indicating that I am now in a transaction. After the commit the transaction count is 0, which shows that the transaction has been completed. And finally, I turn ImplicitTransactions OFF again. Now you may be thinking that this is handy, and be tempted to use ImplicitTransactions everywhere. That may not be a great idea. Since the transaction stays active until you commit, things can be left hanging. They can lock out other operations. Put simply, this operation mode is for very special circumstances. Normally, you would not use it. You may, however, find that you are running with ImplicitTransactions if the DBA has set ANSI default on at the instance-level, which would also be quite unusual. Because that setting affects other things as well, be sure to check what settings are in effect whenever you start to work on a server that is new to you.

Revisiting Autocommit, Explicit, and Implicit Transactions

Let's recap what we've seen so far. The default transaction mode of SQL Server is autocommit. Each statement is wrapped in a transaction that is either committed or rolled back if an error occurs. In explicit transaction mode you

start with a BEGIN TRANSACTION statement, which can be shortened to just BEGIN TRAN. Beginning a transaction also tells SQL Server that the data is consistent at that point. Explicit transactions are ended by either a COMMIT or a ROLLBACK statement. If rolled back, the data is returned to the state it was in at the beginning of the transaction. In the implicit transaction mode transactions are begun whenever a statement is executed, and finished when you issue a COMMIT or ROLLBACK command. In an interesting irony, you must explicitly enable IMPLICIT_TRANSACTIONS. There are two other transaction modes that we haven't discussed yet. Batched-scoped transactions are only applicable to MARS: Multiple Active Result Sets. This mode is only used by a session set up for MARS processing. I won't be discussing this advanced transaction mode further in this course. Distributed transactions are used when transactions cover operations in more than one SQL Server instance. These are managed by the Microsoft Distributed Transaction Coordinator, MS DTC. I won't be discussing these either, though you need to know that they are available. The BEGIN TRANSACTION statement also accepts a name, which can be any valid identifier, no more than 32 characters long. It can also be a variable name of type char, varchar, nchar, or nvarchar. Again, only 32 characters can be used. Any extra characters will be truncated. The optional WITH MARK section specifies that the transaction is marked in the log. Description is a string that describes the mark, and it can be up to 128 characters long. This allows for restoring a transaction log to a named mark. Like BEGIN TRANSACTION, you can also use a name in the COMMIT state, but that is ignored by SQL Server. The database engine does not match the name of the COMMIT TRANSACTION with the name used in the BEGIN TRANSACTION statement. Think of it as a way of commenting or highlighting where a transaction begins and ends. This can be helpful in long sections of code, especially if nested transactions are used. In that case, though, be sure to indent any code inside an explicit transaction to make it easy to see how it works. The COMMIT TRANSACTION statement also has a possible option with DELAYED_DURABILITY, which can be set to ON or OFF. If used, DELAYED_DURABILITY uses asynchronous write to disk for the log file. Now this can reduce contention, since sessions do not wait for Log.io to finish, and that can increase concurrency. On the other hand, data loss is possible with this option, for exactly the same reason. This effectively breaks the D in ACID. However, if you can tolerate some data loss, this option can increase throughput. I just wouldn't use it when working with banking transactions or anything that can not tolerate data loss. The ROLLBACK TRANSACTION statement can also take a name. It is not ignored. It must match the name from the BEGIN TRANSACTION statement in an ordinary explicit transaction, and must be the same name from the outermost BEGIN TRANSACTION statement when nested transactions are used. Oh, and one other thing. You can also use the COMMIT WORK and ROLLBACK WORK statements instead of the previous commands. These are part of the ANSI SQL-92 standard. The only real difference is that these aliases do not accept transaction names. You should probably stick with COMMIT and ROLLBACK transaction to avoid mixing styles and standards.

Introducing Savepoints

The SAVE statement sets a savepoint within a transaction. As you can with BEGIN TRANSACTION statement, you give it a name using the same rules, a valid identifier with no more than 32 characters. This defines a location to which a transaction can return if part of a transaction is canceled for some reason. The ROLLBACK statement is used for this operation and specifies a previously set savepoint to return to. This does not cancel the entire transaction though. You'll still need to finally COMMIT or ROLLBACK the whole transaction. Now duplicate savepoints are allowed in a transaction, but you can only rollback to the last savepoint with that name. Let's see how that works in the next demo.

Demo: Savepoints

Here's a script to demonstrate these concepts. First, I create a simple test table. Then I start up an explicit transaction giving it the name, Hitchhikers. After inserting one row I set a savepoint. Then I insert a second row and show the contents of the table. Next, I rollback to the savepoint and again show the contents of the table. I'll insert a third row, then save the transaction with a name defined by a variable. I'll insert one more row, then roll it back to the second savepoint. Finally, I commit the transaction. Let's run this. The results show how the savepoints have been used. The first result set, before the first rollback, shows the two rows inserted so far. However, after the rollback to the first savepoint the second row added is gone. Now the script adds to more rows and sets a savepoint in between them. Before the second rollback there are three rows, however, at the end of the transaction the final result shows that only two rows have been committed to persistent storage. Now, at any point, I could reuse a savepoint name. Perhaps all savepoints are somewhat generic, and returning to the last one is just fine. Reusing savepoint names would make sense in this case. Now this would be a great time to roll up your sleeves and try this out. Taking this script as a template, build a new one, reusing savepoint names along the way. While you're at it, add in a few more mathematical constants, like the gravitational constant, the find structure constant, the plane constant, and others that you know or can look up.

Module Summary

In this module we looked at the framework of T-SQL used to set up transactions. I talked about autocommit, explicit, and implicit transactions, and touched on MARS and distributed transactions. You learned about the syntax for setting up explicit transactions and enabling implicit ones. Then I discussed and showed you savepoints where you saw good reasons to name transactions and save points. Finally, I left you with a little challenge using savepoints. Try to complete that before continuing on to the next module where you will learn the basics of transactional isolation and the levels of isolation available in SQL Server.

Managing Basic Isolation Levels

Introducing Isolation Levels

Hello. Welcome back to the course, Managing SQL Server Database Concurrency. My name is Gerald Britton. In the previous module, I discussed the framework for starting, committing, saving, and rolling back transactions. Transactions protect units of work from other processes. This module will go into the various levels of protection that are available. These are called isolation levels, and they range from the one at a time, don't mess with me level down to an almost free for all mode. Isolation levels directly affect concurrency. Here's an overview. First, I'll talk about the general concept of isolation levels to get you thinking about what they are. I'll also cover the anomalies that can appear if you use an inappropriate level, and you'll learn about the effect each isolation level has on concurrency. Then we'll look at the free for all level, READ UNCOMMITTED, also known as NOLOCK. Now misunderstanding this one has led to some wacky results in production systems. Still, it has good uses, so I'll spend some time on it. It can maximize concurrency, but has the most possible anomalies. The next one, READ COMMITTED, is actually the default isolation level for most queries, and for good reason, as you'll see. It has fewer anomalies and concurrency is almost as good as READ UNCOMMITTED. REPEATABLE READ provides another level of safety and removes another possible anomaly. This limits some types of concurrency, however. SERIALIZABLE is the most restrictive isolation level, and free of the common anomalies, but can put the brakes on concurrent operations. SNAPSHOT isolation is designed to avoid these anomalies and increase concurrency. It's a larger topic than the others, so I'll leave that one to the next module.

Understanding Isolation Levels and Anomalies

Thinking about the traffic analogy can help give you a picture of the four main isolation levels. READ UNCOMMITTED is like traffic chaos; anything goes. At any moment, you might get into trouble. Concurrency is high, in that there are many simultaneous road users, but things are far from smooth. Also, at this level, you can get all three of the major anomalies, dirty reads, nonrepeatable reads, and phantom reads. I'll dive into those in a moment. The default isolation for most transactions is READ COMMITTED. It's like organized traffic; heavy at times, but with reasonable concurrency and no chance for dirty reads. The REPEATABLE READ isolation level removes the NONREPEATABLE READ anomaly, as the name implies. Concurrency can be somewhat less, like the lightly used road in the picture here. When you choose the SERIALIZABLE isolation level you're saying that you want the road all to yourself, concurrency, shmoncurrency. Who cares what others need. I'm going to get my job done, and the rest of

you can wait. At this level the anomalies disappear. Okay, so what are these anomalies? Dirty reads happen when a session reads data that has not yet been committed that may, in fact, be rolled back. You get something, but can't count on it being a source of truth. A NONREPEATABLE READ happens when a session queries a table, does some work with results, then queries the table again,, but the second time the results are difference, since another session may have updated that table in the meantime. Phantom reads happen when one session is reading from the database and another one is inserting rows. It can happen that the first session gets some of the rows being inserted by the second session, even if the latter is not finished yet.

Demo: Read Uncommitted

In this demo we'll look at the lowest, most unrestricted isolation level, READ UNCOMMITTED. This is also known as NOLOCK, though I prefer to use the full name for reasons you'll see in a moment. On the other hand, the name NOLOCK does give a hint about how things work. All isolation levels work via a hierarchy of locks. It can get complicated, and I'll dig into the whole subject in a later module in this course. For now, just keep in mind the idea that a lock on a resource can be shared, which means many simultaneous readers are exclusive, which means what it says, one user at a time. Here in Data Studio I have two queries up, side by side. The one on the left uses the isolation level, READ UNCOMMITTED, using this SET statement. It starts the transaction, and then does two SELECTs. The query on the right updates a row in my Orders table, so first let me start the left hand query just running the first bit of it. The results are exactly as expected, since there are no other sessions running at the moment. Now let me see if I can force a dirty read. In my dirty read script I start a transaction that will do an update. I'll stop after the first query. At the moment then, I've updated a row, but not committed it. In fact, I realize I've made a terrible mistake. No customer, or their descendants, is going to wait almost 1000 years to get their order, so I'll rollback my erroneous update. Before I do, though, let me run the second half of the left hand query. See that? I am reading that new requested date, even though it will never be committed, and that is a dirty read. Okay, I'll rollback that update. Now let's start again. This time I'll use my NONREPEATABLE READ script. I'll start the left hand transaction again, but not the SELECT statement. Now I'll run the first update on the right, and now the first SELECT on the left. I see they updated and committed OrderRequestedDate. Now with that transaction still active, I'll run the second update query, fixing the erroneous update. And back in the left hand query I'll run the rest of the transaction. See that date? It's back to what it was, and that is a NONREPEATABLE READ. For the third try I'll use the PhantomRead script. It inserts a row into the Orders table, then rolls it back. I'll run the first part of that insert, and on the left I'll run the first part of the transaction. You can see the new row, even though I haven't committed it yet, and if I roll back the insert, then run the rest of the left hand query, that new order is gone. That's a phantom read; here today, gone tomorrow. Now if I can easily produce these anomalies, does that mean there is no locking? No it does not. SQL Server takes a special kind of lock called a schema stability lock when running under READ UNCOMMITTED. This lock blocks any changes to the DDL operations and some DML that might change the table

definition while it is being read, and that's why I don't like to use the name NOLOCK, since there is always at least a schema stability lock. So with all these anomalies, why use READ UNCOMMITTED at all? Well, suppose you were asked to help debug a long running query that is updating some table. You find out that the table has an exclusive lock on, but you want to look at it anyway to get an idea of the progress of the query. Well, running a query under the READ UNCOMMITTED isolation level will let you do that. On the other hand, you should almost never use this isolation level in any code that matters or produces business results that need to be accurate, oh, and before we move on, you need to know that you can run under READ UNCOMMITTED without a separate SET statement. You can use a table hint on the query using the WITH clause, like this. Before you do that though, note that table hints are generally discouraged. Microsoft recommends that hints be used only as a last resort by experienced developers and database administrators. An exception would be a debugging session, like the one I mentioned just now. In the next demo we'll move up a level to recommit it to see if we can shake off any of those anomalies.

Demo: Read Committed

READ COMMITTED is the default, out of the box transaction isolation level for SQL Server. Of course, like so many defaults, this one can be changed. That means you should never assume what defaults are in effect. Ask if you don't know, and remember that explicit is better than implicit. Back in Data Studio, let me first show you how you can determine the current isolation level. SQL Server comes with a set of dynamic management views that can tell you what is going on. You can use sys.dm_exec_sessions to determine the transaction isolation level that is in effect. It's quite simple. If I run this now, you can see that the current level is ReadCommitted, the default. Now let's revisit the queries we used in the last demo, but this time under the READ COMMITTED transaction isolation level. As before, I explicitly set the desired isolation level. Then I'll run the first query to set the baseline. Note that OrderRequestedDate, June the 1st, 2019. Now let me run the first part of the dirty read script. Remember that I haven't committed that update yet. If I run the second query on the left, it doesn't return immediately. That's because it's trying to get a shared lock to the data, but it can't, since the other session has an exclusive lock, which it needs to update the table. That's how READ COMMITTED works. After about 5 seconds it times out, and that's the default lock request time for LocalDB. On the other hand, READ UNCOMMITTED only locked the schema and didn't wait or time out. Now, I'll modify the dirty read script to make it wait for 4 seconds, and then roll back the update. Now I'll quickly rerun the second query. It waits, then returns the unchanged results. That dirty read is gone. To show that the queries still suffer from nonrepeatable reads I'll repeat the exercise from the first demo. First, start the left hand transaction. Next, perform the first update. Then, run the SELECT on the left. I can see the updated results. Now undo that update on the right, and again, SELECT on the left. Did you notice how the first query returned the updated date, and the second the original date? That NONREPEATABLE READ is still there. This is probably a good time to roll up your sleeves and create a situation where you can produce a phantom read. Inserting a row or two in one script while reading from another one ought to do it. Pause this video and give it a try. If you did that, you now know

that phantom reads are still there, even under READ COMMITTED, but leaving that problem for a moment, let's see if I can get rid of that NONREPEATABLE READ.

Demo: Repeatable Read and Serializable

Here's the REPEATABLE READ isolation level. This is the same script as the last demo, except for the SET statement. I'll run just the first part of this script. Now I'll run the first update in the NONREPEATABLE READ script. See that? It's waiting. In order to guarantee results, the query on the left under the REPEATABLE READ isolation level has taken locks to prevent updating while that transaction is active. Now if I allow the REPEATABLE READ query to complete, the update query will as well since the locks have been released. Oh, and by the way, did you try that little exercise at the end of the previous demo? Were you able to see phantom reads? If so, try your scripts again under the REPEATABLE READ isolation level. Note that you may need to add many more rows due to the small size of the rows in this table, in fact, you may need to add several hundred this time. The reason is that SQL Server has put a lot on the page containing the rows in order to make the reads repeatable, and a row in this table only uses 23 bytes of the 8090 that are available, excluding overhead. That's about 350 rows per page. So the first three isolation levels could not get rid of all of the anomalies. Let's take it up a notch. This is the SERIALIZABLE transaction isolation level, and this is the same script as the last demo, except for that SET statement. It will produce the same results as before against the NONREPEATABLE READ script, although you should try phantom reads again. You won't be able to reproduce the phantom reads out of the SERIALIZABLE isolation level, and that's the intention. SQL Server takes locks to prevent that from happening.

Summary

In this module you learned about the concepts behind isolation levels and the kinds of anomalies that can occur at each level. You saw that as you increased the isolation level from READ UNCOMMITTED to SERIALIZABLE the anomalies disappeared one by one. However, each level also results in lower concurrency. That is caused by the locking used to ensure the requested isolation. This approach to concurrency is called pessimistic. Going back to the traffic analogy, SQL Server assumes that a collision may occur and uses traffic signals, that is locking, to prevent it. There is also a different approach called optimistic concurrency, which basically assumes that everything will likely go fine and raises exceptions when things don't. I'll talk about that approach in the next module.

Implementing Snapshot Isolation Levels

Introducing Snapshot Isolation and Row Versioning

Hello again. Welcome back to the course, Managing SQL Server Database Concurrency. My name is Gerald Britton. In the previous module I introduced four isolation levels with decreasing anomalous behavior, but also decreasing concurrency. The implementation is done using locks and is called pessimistic concurrency, since it assumes a collision may occur, and seeks to prevent that. In this module I'll discuss a different approach called snapshot isolation, which uses optimistic concurrency. First, I'll describe snapshot isolation and how SQL Server implements it. Next, I'll show you the database-level settings that can be used to enable and control snapshot isolation. You'll see how READ COMMITTED behaves differently with snapshot isolation. Then I'll talk about data modification operations under snapshot isolation and what happens with conflicting updates. I'll talk a bit about space usage in tempdb and give some tips on how to manage that. Of course, there'll be demos along the way to show you the transact SQL to be used. Now let's take a look at how snapshot isolation works. When a snapshot isolation level is used the SQL Server Database Engine maintains versions of each row that is modified. When data is read it uses a version of the data that was committed at the time the reading transaction began. In this way, the chance that a read operation will block other transactions is greatly reduced. When data is modified SQL Server uses a copy-on-write mechanism. Any transactions that begin before the update see a consistent set of rows. To make this possible, every time a row is modified by a specific transaction the instance of the SQL Server Database Engine stores a version of the previously committed image of the row in tempdb. However, we need to keep an eye on tempdb space usage. This will increase with snapshot isolation and can become a problem, even with the most careful design. Now let's take a look at the database setup you need to get started with snapshot isolation.

Demo: Running with Read Committed Snapshot Isolation

In this demo I'll show you how to set up a database to use snapshot isolation. I'll also demo how to determine the current settings and how the operation of an ordinary READ COMMITTED transaction changes when you enable snapshot isolation. First, I need to enable READ COMMITTED snapshot isolation. This is done at the database-level using an ALTER command, like this. Note that you will frequently see this called by its acronym, RCSI. The command succeeds. Here is how you can check to see the current snapshot isolation settings for a database. If I run this now it shows that READ COMMITTED snapshot isolation is on, but not the second option, which I'll explain in the next demo. But what does this mean? It means that whenever there is another transaction reading a table that is being

modified in another session SQL Server will use role versioning to isolate those changes. I won't see any changed rows until the modifying transaction is done. Now I'm reusing an update query from the previous module with one modification. I've added an explicit wait to keep the transaction open for 10 seconds. Now I'll reuse that READ COMMITTED script. Note that I haven't changed the statement setting the transaction isolation level. It still says, READ COMMITTED. However, now that I've enabled RCSI at the database-level, this will now use snapshot isolation. To begin, I'll just start that transaction and run the first select. Now I'll switch to the update query and run it. Then, quickly back to the ReadCommitted query, and run the second SELECT. Because snapshot isolation is in effect, that means this finishes immediately, even though the update is not yet finished. After a few more seconds, we see the update finish. I rolled back that update, though, to fix that bogus order requested date. The key thing here is that the results from the SELECT are returned immediately, even though the update is still running, whereas before, under the READ COMMITTED isolation level, the SELECT query had to wait for the update to finish. Now it no longer does. This is the magic or row versioning at work. No data locks on rows are taken, though, of course, schema locks are still required. It's like running a query under READ UNCOMMITTED, aka NOLOCK, without the risks of anomalies.

Demo: Using Snapshot Isolation When Modifying Data

In this demo you'll see how to use snapshot as an isolation level and how it affects the operation of the database. You'll also see what happens when data modifications between transactions are in conflict with each other. This is also done at the database-level. This command sets ALLOW_SNAPSHOT_ISOLATION ON. Note that this is different from RCSI above, though under the covers the same mechanisms are used. Setting RCSI on affects all read operations. Allowing snapshot isolation, as in this command, just grants permission. To actually use it we need to set the isolation level explicitly for each transaction that needs it. Running this command enables snapshot isolation. Now let's use it. In this update script I first set the ISOLATION LEVEL to SNAPSHOT. I'm also setting the LOCK_TIMEOUT to 15 seconds. Wait a minute, I hear you say. I thought there would be no locking. Well, under snapshot isolation SQL Server still needs an update lock briefly to figure out which rows will be updated. For any row not matching the selection criteria, the lock is immediately released. Then I run the transaction to do the update. That works fine. However, to make it interesting, let's start up a second script, like this one. Since both scripts are updating the same row, I don't want a timeout at this phase. Now what is this error message I see before me? Sorry to go all Shakespearean on you. Snapshot isolation aborted due to update conflict. But what happened? Both transactions ran under snapshot isolation. That means that both were using role versioning to keep their updates separate. However, when both transactions finished SQL Server had to store something in the database. It chose the session that finished first, and aborted the second one. So it used the update from the first session, but not the second one. Now if I were to change the order ID of the second session, both updates would succeed. In fact, why not pause this video now and try that combination to verify.

Demo: Dynamic Management Views (DMVs)

In this demo I'll show you how to check up on how the database is doing with snapshot isolation and the effects on tempdb. I'll use a few of the many Dynamic Management Views available that show information about active snapshot transactions. You'll find more information in the Gist for this course, here. First, let me start up the update script once more and make it wait 30 minutes before committing. Now that it is running, let's see what we can discover. This script uses some of the interesting DMVs available. First, let's see just what's going on. This DMV returns a virtual table for all active transactions that generate or potentially access row versions. I'll join on the sessions DMV to get the database name. Running that query, you can see the updating transaction that I just started in the other session. The is_snapshot column shows that there is an active snapshot transaction, the one I started up a moment ago. The elapsed_time column shows the time in seconds for this transaction. The second query returns a table that displays total space in tempdb used by the version store records for each database. Here you can see that BobsShoes, where I've got the update transaction running, is currently using 8 pages, which is 64 K. Recall that a page is 8 KB in SQL Server. This query also pulls in statistics from tempdb as a whole, and shows how much of tempdb space is used by the version information for our database, as well as the overall tempdb space usage, as a percentage of the allocated space. The third query returns a virtual table that displays all version records in the version store. Now be careful with this on a busy system. Since it returns actual data rows in the version store, this can be expensive to run. Here, though, you can see the rowset_id that was generated by the database engine for the updating query. The last query I'll show you here returns a virtual table for the objects that are producing the most versions in the version store. Note that this one can also be expensive to run, so be careful when you use it.

Locking vs. Row Versioning

Let's take a moment to look at some of the pros and cons of the two approaches to concurrency management; locking and row versioning. Locking uses four levels of isolation, from READ UNCOMMITTED through SERIALIZABLE. Row versioning has two, READ COMMITTED snapshot isolation, which changes all READ COMMITTED operations in a database to use row versioning, and snapshot isolation, which is set on a per transaction basis. The first difference is that the four locking isolation levels are compliant with the ANSI SQL-92 standard. SQL Server's row versioning using snapshot isolation is proprietary, and that means that if you want to stick to the standard pessimistic concurrency using locking is your only choice. Locking is usually the preferred choice in systems with long- running updates, and by long-running I mean inserts, deletes, and updates that involve more than a few thousand rows. With larger loads, transactions running under snapshot isolation have a greater chance of experiencing update conflicts. However, with loads that are read-heavy, such as reporting operations, row versioning eliminates the three hombres of anomalies. That means no dirty, nonrepeatable, or phantom reads, and you'll also get consistent results every time and maximize concurrency. Locking places no extra load on tempdb,

though, of course, tempdb continues to be used for other work. Row versioning uses tempdb for the version store, and you need to watch latency and storage consumption. Locking, by definition, means that some sessions may block others if they hold more restrictive locks or just get there first, and that can mean less concurrency. On the other hand, row versioning does not use blocking and can attain higher concurrency levels.

Summarizing Snapshot Isolation

In this module we looked at row versioning using snapshot isolation, an optimistic approach to maximizing concurrency. After a brief introduction to how SQL Server implements row versioning, the first demo covered READ COMMITTED snapshot isolation or RCSI for short. This is a database setting that forces the use of row versioning instead of locking for all READ COMMITTED operations, which can be a good option for read-heavy loads. The next demo showed how transactional snapshot isolation works once enabled for the database. It provides benefits for data modification operations, but carries the risk of high tempdb usage and possible conflicting updates. Then I covered some of the Dynamic Management Views, or DMVs, that you can use to keep tabs on what is happening with row versioning. Finally, I lined up some key differences between pessimistic concurrency using locking and optimistic concurrency using row versioning and gave some guidance on when you might choose one over the other. The TLDR of this discussion is that there is no one-size-fits-all answer. You need to benchmark, test, and retest. Fortunately, it's usually easy to switch between the two concurrency modes, especially if you use the implicit SET READ_COMMITTED_SNAPSHOT ON option, which requires no changes to existing queries. Now since locking is such a big part of this discussion, this is a good time to look more closely at how locking works in SQL Server. See you in the next module.

Locking in the SQL Server Database Engine

Introducing Locking

Hi again. Welcome back to the course, Managing SQL Server Database Concurrency. I'm Gerald Britton. So far in this course, we've covered the essential material about the I in ACID, that's isolation, and how to achieve that with varying degrees of certainty using the various levels of isolation available to us. In this module I'll take you in deeper to the mechanism that makes all of this possible, locking. You'll learn about the many locking levels and how lock escalation works. Let's get started. First, I'll cover the basic locking mechanism and how it works to protect data. Going one level deeper, I'll expose the lock granularity and hierarchies used by the database engine, and how they work to minimize the cost of locking while simultaneously maximizing concurrency. That's the first dimension of locking. The second dimension of locking contains lock modes. These work together with lock granularity and hierarchies, and determine how the resources can be accessed by concurrent transactions. Lock compatibility controls whether multiple transactions can acquire locks on the same resource at the same time. I'll tell you right now that this is complicated stuff, but I'll concentrate on giving you the big picture here, even though the devil may be in the details. The official documentation will, of course, be your go to resource. Locks can escalate, that is, move up the lock hierarchy. This is usually handled dynamically by the database engine, and you need to know the what and why of lock escalation. The last topic I'll cover is deadlocks. A deadlock occurs when two or more tasks permanently block each other when each has a lock on a resource that the other ones need. I'll introduce the topic here and delve much deeper into it in the next module. Locking is actually a simple idea. Here we have two users, Trillian and Arthur. They both want to access the SQL Server database. Now if they both want to read, well, that's fine, away they go. If Trillian wants to insert, delete, or update some information, though, there must be some control. This is done by putting a lock on the object being updated from an individual row all the way up to the database, depending on what Trillian is doing. Now if Trillian starts her update and the objects get locked, what happens to Arthur? His access is blocked, waiting for Trillian to finish. Similarly, if Arthur is already reading the data that Trillian wants to update, she is blocked until he finishes. Let me show you what this looks like in the next demo.

Demo: Blocking in Action

It's easy to see blocking in action. SQL Server provides lots of information about what's going on inside. First, let me spin up two sessions. Both Trillian and Arthur are reading from the database. I'm going to set the CONTEXT_INFO so that I can easily tell the sessions apart in a separate query. I'm also explicitly using REPEATABLE READ as the

isolation level, just to see some interesting locks. Finally, I have disabled the LOCK_TIMEOUT by setting it to -1. I have a third session using the dynamic management view, sys.dm_tran_locks. Let's see what we can find. This query also shows the context info for the two sessions I started. With the two queries running, each query holds four locks. They're all shared locks, which is what the S means. Now the IS locks are signals that the transaction intends to inquire a shared lock, and that will prevent another transaction from getting an exclusive lock at the same time. More about those later on when I talk about lock compatibility. The locks are about the database, page, key, and object level, and here the object is the table being read. All of these locks have been granted, so that they are enforced at the moment. Now there is much more information available using this DMV. Consult the official documentation for all the details. The link to that is in the gist for this course. Now I'll roll back both queries, then change Trillian's to update the table instead of just reading from it. Now let's see what happens. Without getting ahead of ourselves, just notice that Trillian now has the exclusive locks, that's the X, and that Arthur is waiting for a shared lock, the S. Here, I'm going to introduce you to a secret weapon used by DVAs all over the world to troubleshoot locking and blocking, among other things. Sp_WhoIsActive is a free, stored procedure from Adam Machanic, long-time SQL Server guru and Microsoft MVP. I suggest you download it and install it on just about every SQL Server instance you work with. It will save you time and give you many helpful insights into what's going on, especially when there seems to be a problem. Sp_WhoIsActive shows our two sessions and also shows blocking sessions. Here you can see that session 60, that's Arthur, is blocked by session52, that's Trillian. You can also see what Arthur is waiting for. LCK_M_S, which means he is waiting for a shared lock, which makes perfect sense. Now this is just the tip of the iceberg with respect to locking. I'm going to step away from the demo for a bit to introduce you to locking granularity.

Demo: Understanding Lock Granularity and Hierarchy

SQL Server uses multigranular locking. This means that different types of resources can be locked by a transaction. To keep locking costs low, the Database Engine locks resources automatically to fit the task at hand. You don't request specific locks explicitly. Locking at a smaller granularity, such as rows, means greater concurrency, but more overhead because it may need many locks to cover a large set of rows. On the other hand, locking in a larger granularity, such as the table level, means lower concurrency, since it can block any other access to the same table, but has less overhead because it needs fewer locks. SQL Server often has to get locks at multiple levels of granularity to fully protect a resource. We call such a group of locks a lock hierarchy. Let's go back to the demo to see this in action. We'll use the Trillian and Arthur queries again to see how these two sessions use locking granularity and hierarchy to get the job done. Once again, the sys.dm_tran_locks Dynamic Management View will show us the state of affairs. First, let's look at Trillian's locks. She has a shared lock on the database and an exclusive key lock, since the table is a clustered index, and intense exclusive locks on an object, which is the table itself and the page. I'll come back to intent locks in a moment. Now what about Arthur? He also has a shared lock on

the database and is waiting on a shared key lock. His intent locks have already been granted. This is a perfect, if simple, picture of a locking hierarchy. We can see the different granularities from database down to page and the different requested modes and status of each lock. If you have an active SQL Server instance that you work with, take a moment and see what you can find out using this DMV. You'll probably discover, though, that many sessions do not set context info. Okay, now let me show you the complete set of granularities that are in use in SQL Server.

Reviewing SQL Server Locking Granularities

There are currently 11 locking granularities in SQL Server. Let's look at each one. At the top, which is the lowest granularity level, is the RID or row ID used to lock a single row within a heap. Recall that a table is either a heap or a clustered index. A key resource is used to lock a row within an index, which, of course, includes clustered indexes. It is used to protect a range of rows in a SERIALIZABLE transaction. It does this by preventing the insertion of new rows with keys that fall within the range of keys being read by the transaction. In this way, phantom reads are prevented. A page lock is the next higher level. This represents an 8 KB page in the database, and could be a data or index page. An extent is a contiguous group of eight pages, which includes data and index pages. This kind of lock is taken when a table or index needs to grow, and a new extent is required. A heap or a B-tree lock is a lock protecting a B-tree, that's an index, or the heap data page is in a table that does not have a clustered index. A table lock, as the name implies, locks an entire table and all data and index pages. A file lock locks a database file. In a simple database there may be only two such files, one for data and index pages, and another for the log file. Larger databases with more complex structures can have up to 32, 767 files. Any operations against those files in the file system will be protected by file locks. An application lock is one specified by an application, such as a .NET application. A metadata lock is a lock on a piece of catalog information. Now that might be the metadata for a database or a table, for example. An allocation unit is used to store all the data and indexes belonging to a table partition. Recall that all tables have at least one partition, usually called the primary partition, and may have many more, depending on the database design. You might catch one of these locks when you drop or rebuild large indexes or tables, since freeing those unused pages is not done until the transaction completes. Finally, a lock on the database resource, as the name implies, locks the whole database. So that's the hierarchy. From the lowest, most granular to the highest, least granular. This is only one dimension, however. The other dimension is that of the lock modes, which we'll look at next.

Introducing Locking Modes

Now that you know more about locking granularities, it's time to learn about locking modes, which comprise the second dimension of locking. There are seven lock modes used by SQL Server, some of which have more than one sublevel. Let's look at them. Shared logs allow concurrent transactions to read using pessimistic concurrency. They

prevent modifications by other transactions as long as the shared lock is held. Normally, they are released as soon as the read completes, unless the transaction isolation level is set to REPEATABLE READ or higher, or special hints are used. This lock mode is not used with snapshot isolation, since it would be redundant. Update locks are used to prevent common deadlocks. Only one transaction can hold an update lock on a given resource, which can prevent deadlock situations. I'll be looking at deadlock debugging in the next chapter. Exclusive locks prevent access to the locked resource by other transactions. Also, read operations will only work on those resources when using the READ UNCOMMITTED or NOLOCK isolation level. Since inserts, updates, and deletes first read data before changing it, these operations usually request both shared and exclusive locks. Intent locks are special and have six sublevels. They are called intent locks because they are acquired before a full lock and show the intention to place locks at that level. This, then, prevents other transactions from invalidly modifying a higher level resource. Intent locks also help the Database Engine to detect lock conflicts at higher levels in the hierarchy, which improves efficiency. The acronyms in parentheses indicate the sublevels. IS protects shared locks on some resources lower in the hierarchy. IX protects exclusive locks. SIX combines S and IX in one shot and allows other IS logs, but blocks any other IX logs on the resource. IU protects update locks on all resources lower in the hierarchy. They are only used on page resources, so they cover key and RID locks. SIU combines S and IU locks or share with intention to update. If a transaction has both read and update operations this lock mode covers both in one request. UIX combines the U and IX locks. When both have been acquired separately, and therefore, both locks are helped. Schema stability locks are acquired whenever an operation is started that cannot tolerate changes in table structure. DDL operations typically do that. If you could read from a table while a column was being modified your results could be unpredictable, so DDL operations acquire schema M locks, M for modify. DML operations typically acquire schema S locks. Bulk Update, when combined with the TABLOCK option, allows multiple threads to load data efficiently while preventing other processes from accessing the target table. Key-Range locks protect a range of rows being read by a statement using the SERIALIZABLE isolation level. This prevents phantom reads, as well as phantom insertions and deletions. These locks have shared, update, exclusive, and null levels, as well as the corresponding intent levels. Oh, and a null resource lock here is used to test that the key to be inserted does not already exist. Now let's go back to that last demo one more time to see how the lock modes are combined with the hierarchy.

Demo: Locking Modes in Action

Here we can see both dimensions at work. Trillian and Arthur both have shared locks at the database-level, and since Trillian is doing an update, her session also holds an exclusive key lock, and there are intent exclusive locks at the page and table level. The object here is a table, since when her update completes, the page and table will be modified. Holding the intent exclusive locks allows for easy conversion to full exclusive locks if required. Arthur has only shared an intent shared locks. Note that the intent shared locks have been granted, but the shared key lock is waiting, waiting for Trillian, in this case, to finish her update. This is a pretty normal picture. Note that you will rarely

see it so neatly laid out in a real system. I have, in effect, frozen these in time by only running the first part of the transactions without commit or rollback. Normally, these locks are granted and released quickly. When they are not, deadlocks may become an issue. I'll be delving into deadlocks in the next module. For now, it's time to talk about lock escalation.

Introducing Lock Escalation

One thing should be clear, at this point concurrency and lock granularity have an inverse relationship. At the lowest level, row or key locks can deliver maximum concurrency; however, for large tables with thousands, millions, or even billions of rows using row locks for everything adds overhead, as you probably guessed. SQL Server uses a process called dynamic locking to keep a balance between concurrency and granularity. This means that, for example, a set of row locks may be exchanged for a table lock, at some point, depending on the number of locks held in the activity in the system. This process is called lock escalation. It has several advantages. For one thing, it simplifies database administration. DVAs do not need to adjust lock escalation thresholds. For example, you cannot set the number of row locks that pauses an escalation to a table lock. This is handled dynamically. This also increases performance, since the Database Engine minimizes system overhead by using locks appropriate to the task. It is also automatic, freeing developers to develop applications instead of trying to manage locking during processing. While you cannot adjust lock escalation thresholds directly, you can indicate what type of escalation behavior you want to use at the table level. You can choose between AUTO, which is the process I just described, TABLE, which forces escalation to the table level, even when a lower granularity is available, or DISABLE, which prevents lock escalation, in most cases. But note that this control is given to handle special cases. Normally, you should let SQL Server manage locking dynamically, and only use this option when that appears to be suboptimal for some case. Also, before you use controls like this one, benchmark your current performance, then check for improvements or regressions against the benchmark after making changes, and as time goes by. Now let's see if we can catch lock escalation at work. See you in the next demo.

Demo: Lock Escalation in Action

In this demo I want to see if I can catch lock escalation in action. To do that I'm going to use this INSERT query. Let me walk you through it. The idea is to be able to insert a given number of rows into the OrderItems table in my database so that I can see what locks are in effect as I increase the number of rows being inserted, and I'm doing it all in a transaction so that I can see the locks before they are released. The query itself uses an old trick. Perhaps you've seen it before, but in case you haven't, it works like this. The INSERT statement uses the results of the SELECT query, which uses a fixed set of values. However, the two CROSS JOINs effectively multiply the number of rows available for insert. Recall that a CROSS JOIN is a Cartesian product. At the time I recorded this, the

sys.columns table had about 1000 rows in it, therefore, the two CROSS JOINs give me 1 million rows to work with. However, I'll limit the final row count with the TOP clause in the SELECT statement. I also have open a query to look at the locks in place for the database. So let me start up the INSERT query, but not finish the transaction. Great, 100 rows have been inserted. Now let's look at the locks. There are 104 locks in place. There are 100 key locks, 1 for each row inserted, 2 database locks, a page lock, and if I scroll down a bit, an object lock with a lock mode of intent exclusive. Well, what is that object? If I take that associated entity ID and call the OBJECT_NAME function on it you can see that it is the OrderItems table. So, after inserting 100 rows, the OrderItems table has an intent exclusive lock. What if I increase the number of rows to be inserted? Let's try 1000. Note that the transaction is still open. Now there is a new lock, an allocation unit lock, which is higher up the lock hierarchy. There are also more page locks and the lock on the table is still intent exclusive, but what's this? Only 199 locks after inserting 1000 rows? That means that SQL Server invoked lock escalation to reduce the number of locks held. That's what the page and allocation locks are doing. Okay now, let's go big or go home; 10, 000 rows, and the locks, only 10 locks in place, and note the lock on the table. It has been promoted to exclusive. Lock escalation strikes again. You may be wondering if there is some magic number that will cause the locks to escalate to the table level. The answer is, it depends. In earlier versions of SQL Server the legend was that 5000 locks would trigger an escalation. With modern versions, however, that number is no longer set in concrete. It depends on system activity, and may go up or down as conditions warrant. Now it's time to recap what's been covered in this module.

Module Summary and Additional Resources

This module gave you an introduction to locking. Given the details presented here, that may surprise you. However, there is much more to learn about locking, and whole books have been written on that topic alone. As I was writing this course I could easily find 4 full length books, 1 close to 1000 pages on this very topic. I've put links to some of them in the gist for this course. I talked about locking granularity and hierarchy, and in the demos showed how even simple transactions involve locks at different granularities, such as row, table, and database. Then we looked at the other dimension of locking, the modes, or strength if you like, of locks from the very low shared intent to the ultimate exclusive lock. I would up with a look at lock escalation, and dynamic locking, and the limited control we have over that as developers. now that you know something of how SQL Server manages concurrency using locking, it's time to look at what happens when things go wrong, and what you can do about it. See you in the next module.

Optimizing Concurrency and Locking Behavior

Introducing Concurrency Optimization

Hello. Welcome back to the course, Managing SQL Server Database Concurrency. My name is Gerald Britton. In the previous module you learned about the basics of locking in SQL Server. Locking is the strategy used to maintain integrity in an active database with many sessions querying and updating data. At the very least, you now understand that locking is complex with multiple granularities and modes. Now since you have to plot a course between a free-for-all, anything-goes database and a my-way-or-the-highway database it is essential to understand how to optimize concurrency to achieve the business objectives. We'll look at some techniques to do that and also how to diagnose problems that can occur, especially the dreaded deadlock. Here's what I'll talk about. Though I briefly mentioned it in the last module, I'll begin with a discussion of lock compatibility. You no doubt see that two sessions cannot have exclusive locks on the same object at the same time, but compatibility on the whole is fairly complicated. When things get busy a phenomenon called deadlocking can occur. You'll see how that could easily happen and what SQL Server does to detect and remediate deadlocks. Now when deadlocks do occur it is imperative to know how to read a deadlock graph. That will give you key insights to understand the specific issues causing deadlocks, which will help you find ways to reduce them. I'll spend a little time to show how to leverage T-SQL to help with responding to deadlines with more than a headshake and try, try again. Lastly, I'll show you some common tweaks that can help reduce blocks and deadlocks or at least keep the important queries up and running. The high level concepts of lock compatibility are these. First, lock compatibility controls whether multiple transactions can acquire locks on the same resource at the same time. Clearly, two transactions can read from the same table at the same time, since both have shared locks, but what about other modes? If a resource is already locked by another transaction, a new lock can be granted only if the mode of the requested lock is compatible with the mode of the existing lock. For example, share locks are compatible with each other, but update locks are not. If the mode of the requested lock is not compatible with the existing lock, the transaction requesting the new lock waits for it or waits for the lock timeout to expire. We say that the transaction holding the lock blocks the other one. Note that this is not a deadlock situation, but rather normal operation in a busy system, and always keep in mind that no two transactions can have an exclusive lock on the same resource. Exclusive locks are never compatible with each other.

Demo: Understanding Locking and Blocking

Let's go back to Arthur and Trillian and see what we can learn about locking and blocking. I have divided the screen into three sections so that you can see everything at once. First, I start up Trillian, who starts to update the Orders table, and then waits for 1 hour. There. We've hit the wait. Next, I'll start up Arthur. If Trillian's locks are compatible with Arthur's request this should end quickly, but it doesn't. To see why, let's look at the locks that we have in effect. Both have shared locks on the database. Shared locks are compatible. Trillian has an exclusive lock on a key. Arthur is waiting for that lock since a request for a shared lock is not compatible with Trillian's exclusive lock. No lock mode is compatible with exclusive. You can also see that Trillian has an intent exclusive lock on the table; that is the object and the page, and Arthur has been granted intent shared locks on the same resources. That means that IX and IS locks are compatible. Now with that as a background, let's look at a little table that shows some of the more common compatibilities and incompatibilities.

Common Lock Mode Compatibility

This table shows some of the most common lock modes and their compatibilities. The first column shows lock modes that have been granted to some transaction. The requested modes show which modes the granted mode is compatible with, if any. This chart drives home what we've covered with shared and exclusive locks. Shared and intent shared locks are compatible with every other lock mode except the strongest, exclusive. The SIX is a combination lock mode, and it means shared combined with intent exclusive. Exclusive modes are incompatible with all other modes on the chart. Note that the update lock is compatible with other shared locks, but nothing else, and that the exclusive lock is still stronger. Since update locks are required before exclusive locks, however, they help prevent some common deadlock conditions Also, an intent exclusive lock is compatible with another intent exclusive lock because IX means the intention is to update only some of the rows rather than all of them, and those sets of rows may not overlap. The full compatibility chart, found in Microsoft documentation, looks like this. Now I won't attempt to explain all these various combinations here, just be aware that this chart is easy to find in the documentation, should you need it, in the transaction locking and row versioning guide. You can find a link to it in the gist for this course.

Demo: Lock Incompatibility and the Consequences

In this demo I want to create a deadlock so that we can explore deadlocks and how to troubleshoot them. Now you know that some locks are compatible with each other, but what happens when they are not? To see what happens, I have created two special update scripts. Each script updates the Orders in OrderItems tables, but does it in the opposite order. Trillian first updates the Orders table, then the OrderItems table, and the author tries to do it the other way around. Now I'll start both Trillian and Arthur's update scripts through the first update. Great. Now the initial locks are in place. Now I'll run the second parts of Trillian and Arthur's update scripts, but then the sys.dm_tran_locks

query. I've enhanced this script to show the name of the object in the results. When this demo is done you might want to take a moment to look up the object name function and the sys.partitions DMV to get a better idea of how this query works. The result set is bigger this time, so I've opened it in Excel to make it easier to view. The transaction locks DMV shows that both Trillian and Arthur have exclusive key locks on the tables they updated. You can see that Arthur's key lock is on the OrderItems table, and Trillian's is on the Orders table. However, now each of them wants to update the resource locked by the other. You can see that in the requested update locks, and both sessions are waiting on those. Since update locks and exclusive locks are never compatible with each other, something's got to give. What happens? You can see it in Trillian's session. The message tells it like it is. Transaction was deadlocked on lock resources with another process and has been chosen as the deadlock victim. Rerun the transaction. Whoops. Since Trillian's updates were in a transaction, they are not committed. The transaction has to be rerun. Before we get to that, however, there's a few questions that need answers. How did SQL Server know that a deadlock was happening? How did Trillian get chosen as the deadlock victim, and how can I find out more about a deadlock? Well the answer to the first question is simple. SQL Server has a deadlock detector that runs by default every 5 seconds, looking for situations like this one. When a deadlock is detected it has to do something about it. It looks at the transactions that are deadlocked and chooses one of them to live and another to die. How does it choose? Well, all things being equal, it could be either transaction. In this case, it was the first one I started, Trillian, but don't count on that. Later I'll talk about some hints you can give to the deadlock detector to help it choose its victim. For the moment, though, let's focus on that last question. How can I find out more about a deadlock?

Introducing Deadlock Analysis Using SSMS

Since SQL Server 2008, Extended Events have been available as the go-to method to track many things, including deadlocks. Viewing the results in an easy to read manner requires a different tool, however, SQL Server Management Studio. At the time I recorded this course, Azure Data Studio did not have the capabilities to nicely view details about deadlocks, at least not graphically. If you've been following this course so far from a computer running Linux or Macintosh, perhaps, I have a bit of bad news. SSMS is not available on those platforms, as yet. You'll need windows for this, either on another box or in a virtual machine. You can download SSMS from this link in the gist for this course or just search for download SSMS to find it. If you need to, pause this module here and download and install SSMS. We'll be using it in the next demo.

Demo: Analyzing Deadlocks Using Extended Events

SQL Server uses Extended Events to keep track of many things, including deadlocks. By default, there is a very useful session, the system_health session. Running this query, you can see that it has already been started. Once again, I've run the two queries for our little updates until the deadlock is detected. That means that the deadlock

should now be viewable in the system_health events. Now I'll switch over to SSMS and look at the Extended Events node under Management. At the moment, there are four sessions defined, and only the one active, the system_health session. Let's see what it has discovered. I'll select Watch Live Data and wait a bit. I am also filtering this list to only look for deadlocks. There is the report of the deadlock that just happened. If I select it and click the Deadlock tab at the bottom I get a nice graph showing the situation. Starting from the left, we see the deadlock victim. That's why it has the large X through it. If I hover my mouse over it, I see the query and can also see that this is indeed Trillian. Similarly, on the right we can see that this is Arthur's query. The two rectangles in the middle tell us who has what locked and what they want. The HoBt, that is Heap or B-Tree, and object IDs are the internal IDs for reference. Also, the object names are printed in full, and you can see that the orders in our item's tables are involved, along with their primary keys. Owner Mode tells us who owns what lock. So, at the time of the deadlock, Trillian has an exclusive lock on the Orders table, and Arthur has one on the OrderItems table. Request Mode tells us who wants what lock. Trillian wants an update lock in the OrderItems table, and Arthur wants one in the Orders table. Since update locks are not compatible with exclusive locks, there is a problem without a channel solution. We have a deadlock. SQL Server chose to kill Trillian's transaction, which would then have to be retried to complete. As nice and visual as this is, there is more information available. If I click on the Details tab there is an XML report. In fact, SSMS used the XML in this report to plot the graph. Interestingly, though Azure Data Studio cannot yet show the graph, it certainly can extract and show the XML report, so let's switch back there. This rather complicated query will extract the XML report. Let me give you a walkthrough. First, notice that I'm joining two DMVs, xe_session_targets and xe_sessions. I need both, which are joined by address, to zero in on the system_health session, which is found in the ring buffer and the in memory session. The CROSS APPLY's extract the XML report from the extended session entry. I use CROSS APPLY like this just to compute an in-row expression, and give it a name to use later. So first, I get the payload from the extended session entry, and from that I extract the deadlock report information. Finally, in the SELECT, I extract the deadlock node, which contains the graph, also in XML. Now let me pause for a second and emphasize that I'm using two advanced features in SQL Server; Extended Events and XML-based querying using XQuery. Both are beyond the scope of this course, but there are other great Pluralsight courses on these topics, and you can easily find many tutorials using your favorite search engine. Of course, the official Microsoft documentation is your reference, and you'll also find detailed how-to's and great examples there. Okay, if I run this I get this result. Double-clicking the last row gives us an XML deadlock report, like the one we just saw in SSMS. It has three main sections. The victim-list gives the id of the session that was sacrificed. The process-list's show all the processes involved in the deadlock, and note that there can be more than just two, as in this example. I could easily match the victim's process id with the first process in the list, and the query in the input buffer shows us that this is Trillian's session. There's other great stuff here too, like the isolationlevel, in effect, and the start and completion times, as well as environmental info, like the hostname, loginname, and database name. The lockMode attribute shows us what lockMode has been requested, and update lock, in this case. The entry for Arthur is much the same. The last section is the resource list, which shows what objects are involved, what locks are held, and which are requested. I suggest

that you get comfortable reading this XML. Not only does it contain more information than the graph, you can also get it from Azure Data Studio and other tools that lack the special graphing capabilities found in SSMS. The system health extended event session is kept in memory, and on an active system can wrap quickly. That's why the target is called a ring buffer. Now suppose you want to isolate deadlocks and save them to a file for later processing. That's easy to do with a custom extended event. Here I have a query that will create a custom event session to collect deadlock information. First, I create a new Extended Event session called Deadlocks, and in the same command I do two more things. First, I add the event I'm interested in, then I specify where to put the reports. Here I was specifying a file in the deadlocks directory of my temp folder. Then I have to start the session. Now I could rerun the previous demo, but why don't you take a crack at it. See if you can create a deadlock session using this script, then cause a deadlock, then view the events from the event file. Hint: you'll need to use the system table value function, sys.fn_xe_file_target_read_file to read the reports from disk. Also, your X query will not need to specify the ring buffer root node, since you're not reading from that location. Pause this video and give it a shot. When you're done, I'll go on to talk about what to do when deadlocks occur, and steps you can take to minimize them.

Handling and Avoiding Deadlocks

On a busy system deadlocks can occur for a variety of reasons, not just the classic example I've been using. That means that applications need to expect deadlocks and handle them appropriately. In practice, that means agreeing with your customer what to do when deadlocks or other errors occur. Generally, there are two options; allow the deadlock victim to fail or retry the operation to try to get it to work. Now you could set up some fixed number of retries or retry for some period of time or just keep trying until it works. You need to know that these are not programming questions, but rather business questions. See if there are standard sets for your organization and make a few suggestions if not, but get agreement. You should also work to avoid deadlocks as much as possible. That usually means keeping transactions as sort as possible, especially with the more restrictive isolation levels. For example, if you have a query that reads some data, processes it, then updates some of it don't put the whole process in a single transaction. You can also consider setting the DEADLOCK_PRIORITY for a transaction. This way you can indicate an important transaction that should not be a deadlock victim, and conversely, set a low priority for transactions that can be considered to be victims in a deadlock situation. I'll show you that in the next demo, along with an application technique called batching that can help keep transactions short, even when handling a lot of data.

Controlling Deadlocks with DEADLOCK_PRIORITY

First, let's return to our competing updates, this time with a small difference. I have set Arthur to have a low DEADLOCK_PRIORITY and Trillian a high one, and I've added a wait between the updates this time, so I can run both scripts all at once. Now I'll fire them both up and wait. See that? Our author is the victim this time. In fact, with

just these two transactions running he will always be the victim. The SET DEADLOCK_PRIORITY statement takes care of that. It can be set to low, normal, or high, and also can take a number from -10 to +10 for finer grain control. You can also use a variable here for flexibility. Normal is the default, as you might expect. For most purposes that is fine. Only start changing priorities when problems arise, and use them sparingly. I've actually gone overboard by setting the DEADLOCK_PRIORITY in both sessions. Now let me show you a sample of a technique that is useful for handling large volume queries and maximizing concurrency at the same time.

Demo: Frameworks for Avoiding and Handling Deadlocks

This script batches up deletes in bunches of 1000 rows, and it takes advantage of the table structure. The Order table has a primary key on the OrderID column, which is defined as an identity value starting at one. This column is also the clustered index. The loop works like this. First, it finds the largest OrderID out of the first 1000 matching the desired customer ID, and greater than the maximum OrderID found so far. Since we initially set the maximum to -1, we start at the start, and since the OrderID is the primary key for the table, the table is processed using a clustered index scan, which helps minimize the IO required. Once we've found the largest OrderID in the batch, the loop deletes matching rows, but restricts it to the OrderIDs between the last batch and the next one. After deleting matching rows, the loop waits for a bit before preceding to give other transactions a chance. If no rows are deleted the loop stops. The advantage of this approach is two-fold. Since the batch size is small enough, the database engine will typically not escalate the locks beyond row and page locks. Of course, you can tune that number of 1000 rows to whatever works best in your environment. And the second advantage is that every batch will be processed very quickly, minimizing the possibility of blocking or deadlocking. The framework shown here can also be applied to insert and update operations, and of course, simple selects under higher isolation levels. Basically, you're trading off the elapsed time of the query for overall increased concurrency in the server, and that's usually a good thing. Now this script shows the framework for a deadlock Retry Loop. Actually, you can use it for retries in general by changing the error messages you're looking for. The loop ends when the retry count is exhausted or the update succeeds. Inside the loop, the transaction is nested inside the T-SQL try catch construction. The code to be retried is inside the transaction. If it succeeds the transaction is committed and the loop is exited. If the transaction fails the code in the CATCH section first checks the error type. If it is not the deadlock victim error it is re-raised for the calling program to handle. This also happens if the maximum number of tries has been hit. Then any active transaction is rolled back. Then the try count is increased, and the code waits for the specified delay. Note that if you change the error you're looking for, you can use the same framework to retry for other conditions or even multiple conditions. Well, that wraps the demos for this module. Let's take a look at what was covered.

Recapping Concurrency Optimization and Deadlock Handling

In this module you learned about lock compatibility and how lock incompatibility can sometimes result in deadlocks. Given the many lock modes in SQL Server, compatibility is a little complicated. I showed you how to find details on deadlocks that occur, both graphically in SSMS, as well as obtaining and reading XML deadlock graphs in Azure Data Studio. Deadlock detection and deadlock reports use Extended Events and XML that can be parsed using XQuery. I noted that both of these technologies deserve courses on their own, and if you look for them on Pluralsight.com you'll find what you need. I discussed coding applications properly to anticipate and handle deadlocks. In fact, error handling in T-SQL is essential in itself, and there are many Pluralsight courses that cover that subject well. You also saw two ways to avoid deadlocks for high priority transactions. The first uses the SET DEADLOCK_PRIORITY command to designate a possible victim or unstoppable winner. The second uses a technique called batched updates to minimize lock escalation and maximize concurrency. In the next module I'll summarize all the topics we've covered in this course and, where useful, point out some areas for additional study. See you there.

Summary

Course Summary

Hello. Welcome back to the course, Managing SQL Server Database Concurrency. My name is Gerald Britton. We're almost done here, and you've made it through some tough material. Now let's review the key concepts we covered in this course. When I introduced transactions I talked about the roles and responsibilities of SQL Server and the database developer. In short, the developer is responsible for starting and completing transactions while dividing business requirements into logical units of work. The Database Engine is responsible for maintaining the integrity of the database, following ACID principles. You learned about explicit and implicit transactions and the role of autocommit. I also discussed savepoints as a way to checkpoint your work within a transaction. With the foundations of transactional processing established, I introduced the standard ANSI isolation levels, sometimes called pessimistic concurrency. Starting at the anything goes, READ UNCOMMITTED level, we climb the isolation level ladder up to the SERIALIZABLE level. Along the way, you saw different kinds of anomalous behavior in the demos, including dirty reads, non-repeatable reads, and phantom reads. I should re-emphasize that READ UNCOMMITTED or NOLOCK, as it is often called, should be used with extreme caution, since all types of anomalies can occur at any time with that isolation level. On the other hand, it has its legitimate uses, especially when troubleshooting seemingly endless queries in progress. Now there's always a tradeoff between isolation and concurrency. If all transactions are SERIALIZABLE, isolation is perfect, and no anomalies will occur. However, that can absolutely kill concurrency, which is a highly desirable characteristic of high performance database systems. Leaving pessimistic concurrency behind, we looked at optimistic concurrency. SQL Server uses a technique called snapshot isolation, which works with row versioning instead of locking. This approach can be better for read-heavy loads that you might see in a DataMart or data warehouse because of dubious value on mixed loads, such as those often found in OLTP systems. One important caveat is the use of tempdb, since it is used to hold the version store for rows being inserted, updated, or deleted. Without careful planning and testing, tempdb can become a bottleneck and needs to be watched for signs of wear and tear. Since everything depends on locking, I spent a whole chapter on that topic, and even so, just scratched the surface. You saw that transactions have to block each other when they need access to the same resources, depending on what they need to do. I covered lock granularity from the lowly row all the way up to the entire database, and how those locks form a hierarchy. I also talked about lock modes from the barely there intent shared up to the almighty, exclusive. Together, the lock hierarchy and lock modes form a matrix that directly impacts the overall concurrency in the system. I also took a look at lock escalation and how locks sometimes have to be raised to higher levels in order to properly isolate a transaction. The other important aspect of locking is compatibility, a topic which I largely left for the next module. Under the gentle title of Optimizing Concurrency we finally discussed the elephant in the room, deadlocks. To get there, I dug into the topic of lock compatibility, or rather incompatibility.

When things are compatible everyone is happy and concurrency is maximized. Compatibility is a complex topic, however, and the full compatibility matrix is large and full of dark corners. When locks are incompatible deadlocks can occur, and when that happens SQL Server has to kill one of the deadlock transactions, called the victim, to allow the system to continue working. To understand that, you learned how to trap deadlocks using Extended Events and view them in SSMS and in the XML deadlock report. Finally, I talked generally about what applications need to do to handle deadlocks when they occur during job runs, and specifically about techniques to use to avoid them in the first place. These included setting deadlock priority, when applicable, and programming techniques to minimize lock escalation during large updates, which should not only help keep deadlocks at bay, but also helps sustain a high level of concurrency. As I wrote this course I assembled a collection of links to relevant material and software that I hope you will find useful. You can find it in a gist on GitHub.com. The link to it is in the first line in the document. Please feel free to use it and suggest corrections for anything that is not right. Also, if you find any other resources, please send them my way, either through GitHub or using the discussion forum for this course on Pluralsight.com. Speaking of which, please do use the discussion forum. I love feedback and questions. Although I may not be able to solve every specific issue you face, I will try to point you in the right direction to get the answers you need. Well, congratulations. You've made it to the end of this course, Managing SQL Server Database Concurrency. My hope is that it will help you advance your skills as a SQL Server developer or DBA, and I'm looking forward to seeing you again, on Pluralsight.com.