How to Track Down Deadlocks Using SQL Server 2005 Profiler

20 June 2008
by Brad McGehee

It is irritating, sometimes alarming, for the user to be confronted by the 'deadlock message' when a deadlock happens. It can be a tiresome business to prevent them from occurring in applications. Fortunately, the profiler can be used to help DBAs identify how deadlocking problems happen, and indicate the best way of minimising the likelihood of their reappearance.

A Quick Introduction to Deadlocks

Deadlocking occurs when two or more SQL Server processes have locks on separate database objects and each process is trying to acquire a lock on an object that the other processes have previously locked. For example, process one has an exclusive lock on object one, process two has an exclusive lock on object two, and process one also wants an exclusive lock on object two, and object two wants an exclusive lock on object one. Because two processes can't have an exclusive lock on the same object at the same time, the two processes become entangled in a deadlock, with neither process willing to yield of its own accord.

Since a deadlock is not a good thing for an application, SQL Server is smart enough to identify the problem and ends the deadlock by choosing one process over another. It does this by killing one of the processes (usually the process that has used the least amount of server resources up to this point) and lets the other one to continue to run. The aborted transaction is rolled back and an error message is sent to the application. If the application is deadlock aware, it will resubmit the killed transaction automatically and the user may never know the deadlock happened. If the application is not deadlock aware, then most likely an error message appears on the application's screen and you get a call from a disgruntled user. Besides irritating users, deadlocks can use up SQL Server's resources unnecessarily as transactions are killed, rolled back, and resubmitted again.

Deadlocks have been the bane of many a DBA. While rare for a well-designed and written application, deadlocks can be a major problem for—how can I say this delicately?—"less efficient" application code. What is even more frustrating is there is not much the DBA can to do prevent deadlocks, as the burden of preventing them in the first place is on the developers of the application. Once an application is designed and written, it is hard for the DBA to do anything other than to identify the offending code and reporting it back to the developers so it can be fixed.

In SQL Server 2000 and earlier, the most common way to track down deadlock issues was to use a trace flag. In SQL Server 2005, trace flags can still be used (1204 or 1222), but they aren't always easy to use. When SQL Server 2005 was introduced, new events were added to the SQL Server 2005 Profiler (they are also in SQL Server 2008) that makes identifying deadlocks very easy. In this article, we learn how to use SQL Server 2005 Profiler to capture and analyze deadlocks.

A Brief Profiler Primer

If you are not already well versed in using SQL Server Profiler, I want to do a quick overview of how Profiler works. If you are an experienced Profiler user, you can skip this section and go right on to the next section.

SQL Server Profiler is a GUI front end for a feature in SQL Server called SQL Trace. Essentially, SQL Trace has the ability to internal SQL Server activity, allowing you to see what is happening inside your SQL Server, including deadlocks. In this article, we will be using the SQL Server Profiler GUI, although you can use Transact-SQL code to accomplish the same thing using SQL Trace.

To capture a SQL Server trace using SQL Server Profiler, you need to create a trace, which includes several basic steps:

1) You first need to select the events you want to collect. Events are an occurrence of some activity inside

SQL Server that Profiler can track, such as a deadlock or the execution of a Transact-SQL statement.

2) Once you have selected the events you want to capture, the next step is to select which data columns you want to return. Each event has multiple data columns that can return data about the event. To minimize the impact of running Profiler against a production server, it is always a good idea to minimize the number of data columns returned.
3) Because most SQL Servers have many different users running many different applications hitting many different databases on the same SQL Server instance, filters can be added to a trace to reduce the amount of trace data returned. For example, if you are only interested in finding deadlocks in one particular database, you can set a filter so that only deadlock events from that database are returned.
4) If you like, you can choose to order the data columns you are returning, and you can even group or aggregate events to make it easier to analyze your trace results. While I do this for many of my traces, I usually don't bother with this step when tracking down deadlock events.
5) Once you have created the trace using the above steps, you are ready to run it. If you are using the SQL Server Profiler GUI, trace results are displayed in the GUI as they are captured. In addition, you can save the events you collect for later analysis.

Now that you know the basics, let's begin creating a trace that will enable us to collect and analyze deadlocks.

Selecting Events

While there is only one event required to diagnose most deadlock problems, I like to include additional context events in my trace so that I have a better understanding of what is happening with the code. Context events are events that help put other events into perspective. The events I suggest you collect include:

- Deadlock graph
- Lock: Deadlock
- Lock: Deadlock Chain
- RPC:Completed
- SP:StmtCompleted
- SQL:BatchCompleted
- SQL:BatchStarting

Events	TextData	ApplicationName	DatabaseName	ServerName	Duration	SPID	LoginName	CPU	Reads
Locks									
Deadlock graph	☑			☑		☑	☑		
Lock:Deadlock	☑	☑	☑	☑	☑	☑	☑		
Lock:Deadlock Chain	☑		☑	☑		☑			
Stored Procedures									
RPC:Completed	☑	☑	☑	☑	☑	☑	☑	☑	☑
SP:StmtCompleted	☑	☑	☑	☑	☑	☑	☑	☑	☑
TSQL									
SQL:BatchCompleted	☑	☑	☑	☑	☑	☑	☑	☑	☑
SQL:BatchStarting	☑	☑	☑	☑		☑	☑		

***Figure 1**: I like to include extra context events to help me better understand what is happening with the code.*

Here's a brief explanation of each of these events.

Deadlock Graph

Of seven events I have listed above, the only event you must have is the Deadlock Graph event. It captures, in both XML format and graphically, a drawing that shows you exactly the cause of the deadlock. We will examine how to interpret this drawing later in this article.

Lock:Deadlock

This event is fired whenever a deadlock occurs, and because of this, it is also fired every time the Deadlock Graph event is fired, producing redundant data. I have included it here because it makes it a little easier to see

what is happening, but if you like, you can drop this event from your trace.

Lock:Deadlock Chain
This event is fired once for every process involved in a deadlock. In most cases, a deadlock only affects two processes at a time, and because of this, you will see this event fired twice just before the Deadlock Graph and the Lock:Deadlock events fire. In rare cases, more than two processes are involved in a deadlock, and if this is the case, an event will be fired for every process involved in the deadlock.

RPC: Completed
The RPC: Completed event fires after a stored procedure is executed as a remote procedure call. It includes useful information about the execution of the stored procedure, including the CPU time used to execute the stored procedure, the total length of time the stored procedure ran, logical reads and writes that occurred during its execution, along with the name of the stored procedure itself.

SP: StmtCompleted
Stored procedures are made up of one or more statements. In SQL Server 2005, each statement within a stored procedure is traced. The SP: StmtCompleted event indicates when a statement within a stored procedure has ended. The StmtCompleted event's data columns provide lots of useful information about the statement, including the actual code in the statement, the duration the statement ran, the amount of CPU time used, the number of logical reads and writes, the number of rows returned by the statement, among others.

SQL: BatchStarting
The SQL: BatchStarting event is fired whenever a new batch begins. Once a batch begins, then one or more individual Transact-SQL statements occur. The SQL: BatchStarting event is a good event to easily see where a batch begins, but other than this, it is not particularly useful.

SQL: BatchCompleted
The SQL: BatchCompleted event occurs when a batch completes. This means that one or more Transact-SQL statements have completed for the batch. The SQL: BatchCompleted event is more useful than the SQL: BatchStarting event because it includes useful information like the duration of the entire batch, the logical number of reads and writes caused by all the statements inside the batch, the total number of rows returned by the batch, and other useful information.

Selecting Data Columns

You don't need to select many data columns to capture the data you need to analyze deadlocks, but you can pick any that you find useful. At the very minimum, I select these data columns, and order them as they are ordered below.

- Events
- TextData
- ApplicationName
- DatabaseName
- ServerName
- SPID
- LoginName
- BinaryData

Selecting Column Organization

I don't perform any grouping or aggregation when tracing Profiler events, but I generally order the data columns in a way that works best for me.

Running the Trace

One of the problems with troubleshooting deadlocks is that they are often hard to predict. Because of this, you

may have to run your deadlock trace for a substantial amount of time (like 24 hours or more) in order to capture deadlocks when they occur. Ideally, you will only perform the trace during time periods where you know deadlocks are likely to occur, in order to minimize the impact of the trace on your server.

If you run a trace for 24 hours, many events may be captured, especially on a very busy production server. If this is the case, you may only want to capture the Deadlock Graph event, and no others, in order to reduce the load on the production server. As I mentioned earlier, the other events I list are context events and are not required to troubleshoot most deadlock problems.

Analyzing the Trace

Now that we know how to set up a trace to analyze deadlocking behavior, let's look at an example to see what information is collected, and how we can best use it to identify the cause of a deadlock.

EventClass	TextData	ApplicationName	DatabaseName
Trace Start			
SQL:BatchStarting	USE [AdventureWorks]	Microsoft...	AdventureWorks
SQL:BatchCompleted	USE [AdventureWorks]	Microsoft...	AdventureWorks
SQL:BatchStarting	BEGIN TRAN --Update One: Run 1st...	Microsoft...	AdventureWorks
SQL:BatchCompleted	BEGIN TRAN --Update One: Run 1st...	Microsoft...	AdventureWorks
SQL:BatchStarting	USE [AdventureWorks]	Microsoft...	AdventureWorks
SQL:BatchCompleted	USE [AdventureWorks]	Microsoft...	AdventureWorks
SQL:BatchStarting	BEGIN TRAN --Update: Run 2nd (...	Microsoft...	AdventureWorks
SQL:BatchStarting	--Update Two: Run 3rd UPDATE sale...	Microsoft...	AdventureWorks
Lock:Deadlock Chain	Deadlock Chain SPID = 55 (010086470...		AdventureWorks
Lock:Deadlock Chain	Deadlock Chain SPID = 54 (010086470...		AdventureWorks
Deadlock graph	<deadlock-list> <deadlock victim=...		
Lock:Deadlock	(010086470766)	Microsoft...	AdventureWorks
SQL:BatchCompleted	BEGIN TRAN --Update: Run 2nd (...	Microsoft...	AdventureWorks
SQL:BatchCompleted	--Update Two: Run 3rd UPDATE sale...	Microsoft...	AdventureWorks
Trace Stop			

***Figure 2**: These are the results of capturing a deadlock using the events I have recommended.*

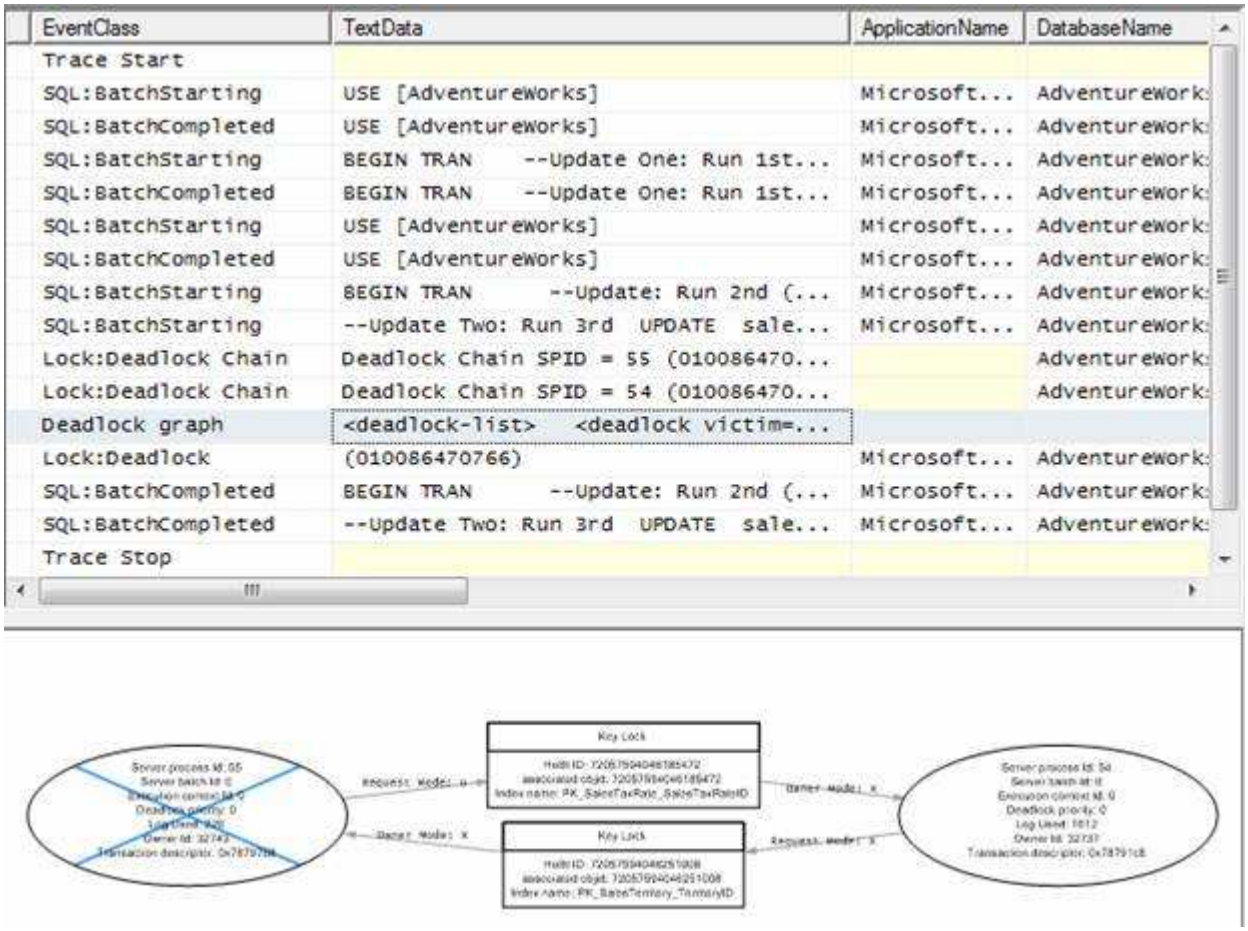
To create a deadlock for demonstration purposes, I ran two separate transactions in two different processes that I know would create a deadlock. These are represented by the eight SQL:BatchStarting and SQL:BatchCompleted events at the beginning of the above trace.

When SQL Server determines that a deadlock has occurred, the first event that denotes this is the Lock:Deadlock Chain event. There are two of these in the above trace, for SPID 55 and SPID 54. Next, the Deadlock Graph event is fired, and last, the Lock:Deadlock event is fired.

Once SQL Server detects a deadlock, it picks a loser and a winner. The SQL:BatchCompleted event that immediately follows the Lock:Deadlock event is the transaction that is killed and rolled back, and the following SQL:BatchCompleted event is the event that was picked as the winner and successfully ran.

If you have trouble following the above example, don't worry, as it will all make more sense when we take a close look at the Deadlock Graph event.

When I click on the Deadlock Graph event in Profiler, a deadlock graph appears at the bottom of the Profiler screen, as shown below.



EventClass	TextData	ApplicationName	DatabaseName
Trace Start			
SQL:BatchStarting	USE [AdventureWorks]	Microsoft...	AdventureWork:
SQL:BatchCompleted	USE [AdventureWorks]	Microsoft...	AdventureWork:
SQL:BatchStarting	BEGIN TRAN --Update One: Run 1st...	Microsoft...	AdventureWork:
SQL:BatchCompleted	BEGIN TRAN --Update One: Run 1st...	Microsoft...	AdventureWork:
SQL:BatchStarting	USE [AdventureWorks]	Microsoft...	AdventureWork:
SQL:BatchCompleted	USE [AdventureWorks]	Microsoft...	AdventureWork:
SQL:BatchStarting	BEGIN TRAN --Update: Run 2nd (...	Microsoft...	AdventureWork:
SQL:BatchStarting	--Update Two: Run 3rd UPDATE sale...	Microsoft...	AdventureWork:
Lock:Deadlock Chain	Deadlock Chain SPID = 55 (010086470...		AdventureWork:
Lock:Deadlock Chain	Deadlock Chain SPID = 54 (010086470...		AdventureWork:
Deadlock graph	<deadlock-list> <deadlock victim=...		
Lock:Deadlock	(010086470766)	Microsoft...	AdventureWork:
SQL:BatchCompleted	BEGIN TRAN --Update: Run 2nd (...	Microsoft...	AdventureWork:
SQL:BatchCompleted	--Update Two: Run 3rd UPDATE sale...	Microsoft...	AdventureWork:
Trace Stop			

***Figure 3**: The Deadlock Graph summarizes all the activity that caused the deadlock to occur.*

Yes, I know you can't read the graph just yet, but I wanted you to see the relationship between the top and bottom portions of the Profiler screen. There is more to this graph than appears obvious from a first look.

The left oval on the graph, with the blue cross, represents the transaction that was chosen as the deadlock victim by SQL Server. If you move the mouse pointer over the oval, a tooltip appears. This oval is also known as a Process Node as it represents a process that performs a specific task, such as an INSERT, UPDATE, or DELETE.

The right oval on the graph represents the transaction that was successful. If you move the mouse pointer over the oval also, a tooltip appears. This oval is also known as a Process Node.
The two rectangular boxes in the middle are called Resource Nodes, and they represent a database object, such as a table, row, or an index. These represent the two resources that the two processes were fighting over. In this case, both of these Resource Nodes represent indexes that each process was trying to get an exclusive lock on.

The arrows you see pointing from and to the ovals and rectangles are called Edges. An Edge represents a relationship between processes and resources. In this case, they represent types of locks each process has on each Resource Node.

Now that you have a basic understanding of the "big" picture, let's drill down into the details. Let's start by looking at each of the Resource Nodes, starting with the successful one, on the right side of our Deadlock Graph.

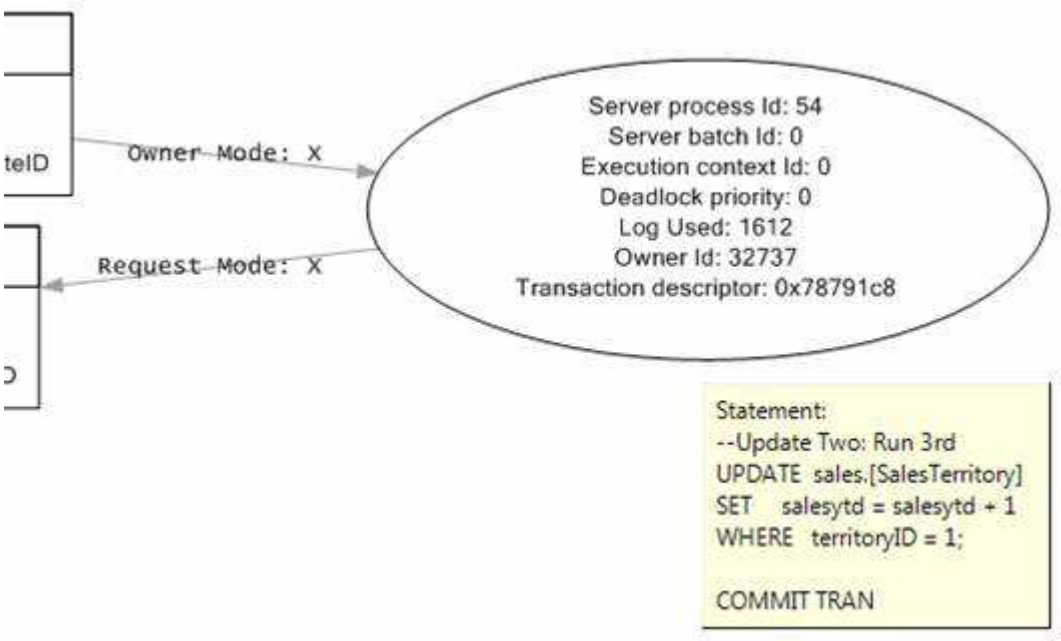


***Figure 4**: This transaction was selected as the winner*

Before we discuss what this Resource Node is telling us, we first need to learn a few new terms. As you can see, there are a number of new terms listed inside the Resource Node.

- **Server Process ID**: This is the SPID of the process.
- **Server Batch ID**: This is the internal reference number for the batch this code is running in.
- **Execution Context ID**: This is the internal reference number of the thread for the above SPID. A value of 0 represents the main, or parent thread.
- **Deadlock Priority**: By default, no one transaction has a greater or smaller chance of becoming a deadlock victim than the other. However, if you use the SET DEADLOCK PRIORITY command for a particular session, then this session can be assigned a value of Low, Normal, or High; setting the priority of this session's transaction over another session's transaction. This allows the DBA or developer to control which session is more important that another when it comes to deadlocks. A value of 0 indicates no priority has been assigned to this process.
- **Log Used**: This is the amount of log space used by the transaction up to the point the deadlock occurs. SQL Server uses this information to help it determine which transaction has used up the most resources so far, so that the transaction that has used the least resources is killed and rolled back, helping to minimize the amount of resources used to deal with the deadlock.
- **Owner ID**: This is the internal reference number for the transaction that is occurring.
- **Transaction Descriptor**: This is an internal reference number that indicates the state of the transaction.

As you can see, there is a lot of data provided, but it is not all that useful unless you have an intimate knowledge of the internal workings of SQL Server. What is more useful is the tooltip. It lists the exact Transact-SQL code that was executed to cause the deadlock to occur.

Now that we have the Process Node definitions down, let's take a more detailed look at what figure 4 is telling us. First, we know it was the winning transaction because it does not have a blue cross through it. Second, it provides the exact Transact-SQL code that was running that caused the deadlock. This is extremely useful information because it allows us to trace the event to specific problematic code. Third, it tells us that this Process Node has an exclusive lock on the top Resource Node (the X represents an exclusive lock). And fourth, it tells us that it has requested another exclusive lock on the bottom Resource Node. When you look at this Process Node in isolation, this is not a big deal. The problem occurs when this transactions bumps heads with another transaction, as we find out next.

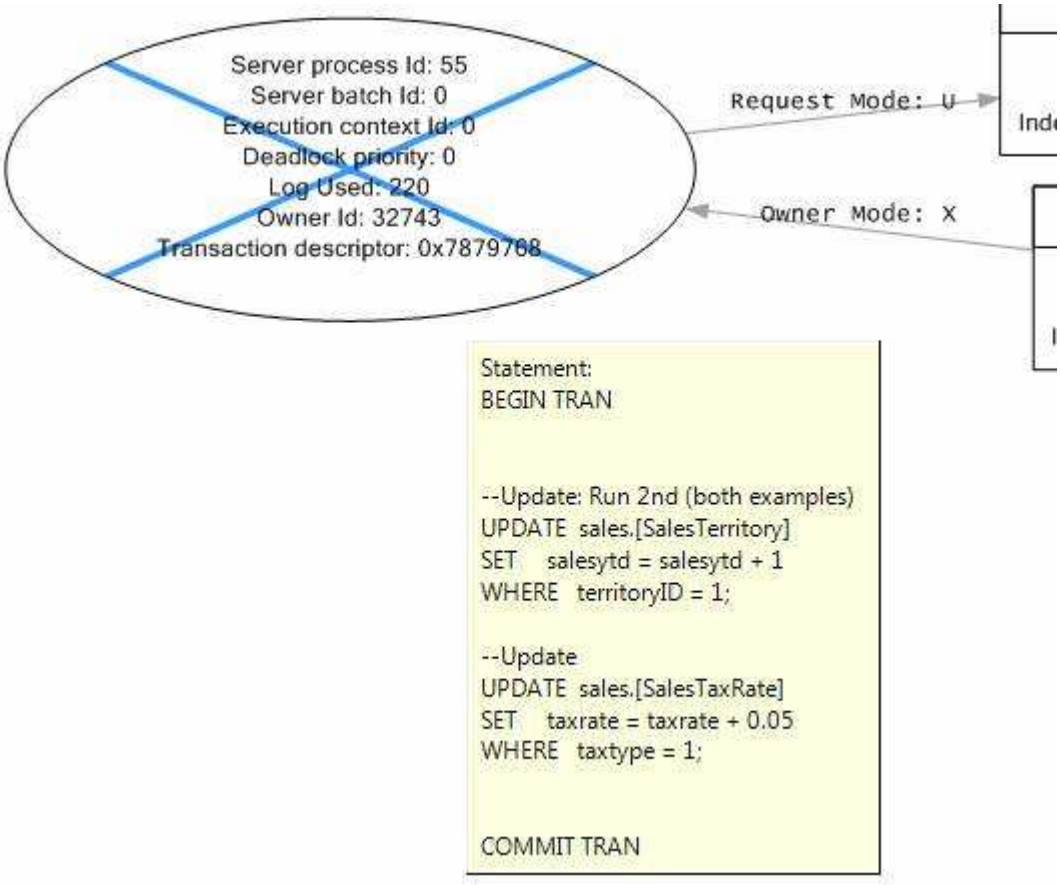


***Figure 5**: This transaction is the deadlock victim.*

On the left side of the Deadlock graph (figure 5 above) is the other Process Node. Like the winning Process Node, this node tells us the following: First, this was the losing transaction. Second, it provides the Transact-SQL code that contributed to the deadlock. Third, it tells us that is had an exclusive lock on the bottom Resource Node. Fourth, it tells us that it requested an update lock on the top Resource node. We'll talk more about the locking conflicts shortly, but for now, let's look at the two Resource Nodes.

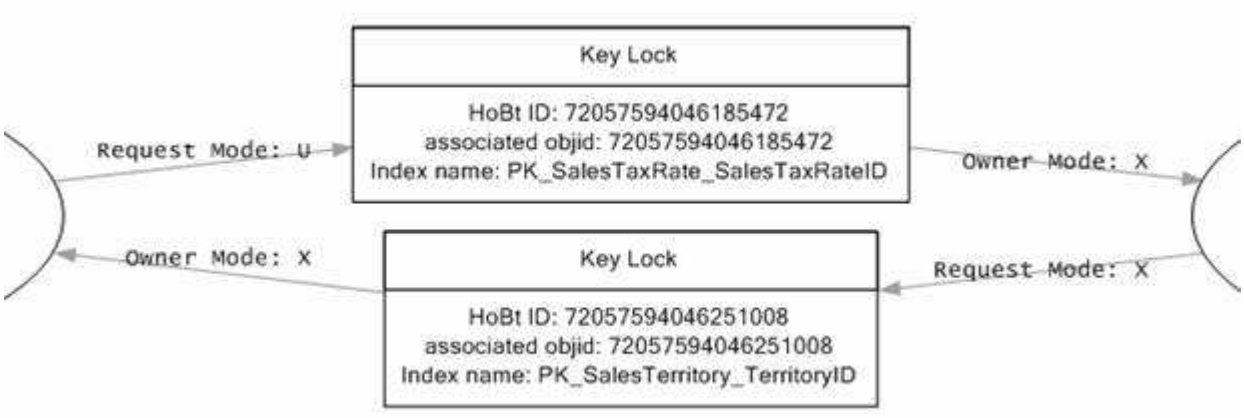


***Figure 6**: The resource nodes tell us what resources the transactions were fighting over.*

Both of these Resource Nodes represent indexes, which the two transactions needed access to in order to perform their requested work. Like Process Nodes, Resource Nodes have some definitions we need to learn.

- **HoBt ID:** This number refers to a subset of data/index pages within a single partition. These may be in the form of a heap or a B-Tree. In SQL Server 2005, the HoBt ID is identical to the Partition ID found in the sys.partitions table.
- **Associated Objid:** This is the object ID of the table associated with this index.
- **Index Name:** The name of the index.

The most useful information is the name of the index, which may be useful information when deciding how to

best reduce or eliminate the deadlocks in question.

The top Resource Node represents the PK_SalesTaxRate_SalesTaxRateID index and the bottom Resource Node represents the PK_SalesTerritory_TerritoryID index.

Now that we have discussed all the details of this Deadlock graph, let's bring all the pieces together.

1. SPID 54 started a transaction, then requested and received an Exclusive lock on the PK_SalesTaxRate_SalesTaxRateID index.
2. SPID 55 started a transaction, and then requested an Exclusive lock on the PK_SalesTerritory_TerritoryID index.
3. SPID 55, as part of the same transaction, then requested an Update lock on the PK_SalesTaxRate_SalesTaxRateID index. However, this lock was not granted because SPID 54 already had an Exclusive lock on the index. In most cases, this means that SPID 55 has to wait its turn before it can get an Update lock on PK_SalesTaxRate_SalesTaxRateID. At this point, SPID 54 is causing a blocking lock on SPID 55.
4. As the above blocking lock is continuing, SPID 54 wants to complete its transaction. In step 1 above, it had only started the transaction, it had not completed it. Now, SPID 54 wants to complete the transaction. In order to do this, it must get an Exclusive lock on PK_SalesTerritory_TerritoryID. The problem is that it can't get a lock on this index because SPID 55 already has an Exclusive lock on it. Now we have a deadlock. Neither SPID can continue because each transaction is locking out the other transaction from finishing. Because this is not a good thing, SQL Server looks at the two transactions and decides to kill the one that has used up the least amount of resources so far. In this case, SPID 55 has used up 220 units of the Log and SPID 54 has used 1612 units of the log. This indicates that SPID 55 should be killed because it has used fewer resources so far.
5. SQL Server kills SPID 55 and the transactions is rolled back, which releases the Exclusive lock on PK_SalesTerritory_TerritoryID, now allowing SPID 54 to get an Exclusive lock on it and to complete the transaction.

You may have to read this section several times in order to grasp all the activity that I have just described. It is not particular easy to follow. However, once you grasp what the Deadlock Graph is telling you, you are now in a better position to identify the code and/or objects that are contributing to the deadlocking problem, allowing an opportunity to fix it. In most cases, this will require developers to get involved. Fortunately, you now have the information you need to share with the developers so they can remedy the problem.

Reducing Deadlocking Problems

Many different things can contribute to the cause of a deadlock in SQL Server. Below are some suggestions on how to eliminate, or at least mitigate, deadlocking problems in your applications. This list is only a starting point and should not be considered a complete list of the options you have to prevent or reduce deadlocking. You may want to share this list with your developers, along with the information you identified during your trace.

- Ensure the database design is properly normalized.
- Have the application access database objects in the same order every time.
- Keep transactions as short as possible.
- During transactions, don't allow any user input.
- Avoid cursors.
- Consider reducing lock escalation by using the ROWLOCK or PAGLOCK hint.
- Consider using the NOLOCK hint to prevent locking.
- Use as low a level of isolation as possible for user connections.

Summary

As you can see, Profiler can be a very powerful tool to help DBAs identify deadlocking problems. Creating a deadlock Profiler trace is simple to create and run. So if you are currently experiencing any deadlocking issues in your databases, take the time now to apply what you have learned in this article. The sooner you give this a try,

the closer you will be to getting your deadlocking problems resolved.