SQL Server Deadlocks

--------------------

Deadlocking occurs when two user processes have locks on separate objects and each process is trying to acquire a lock on the object that the other process has. When this happens, SQL Server identifies the problem and ends the deadlock by automatically choosing one process and aborting the other process, allowing the other process to continue.

Deadlocks can use up SQL Server's resources, especially CPU power, wasting it unnecessarily.

The aborted transaction is rolled back and an error message is sent to the user of the aborted process. Generally, the transaction that requires the least amount of overhead to rollback is the transaction that is aborted.

When a deadlock occurs, by default, SQL Server choose a deadlock "victim" by identifying which of the two processes will use the least amount of resources to rollback, and then returns error message 1205.if you don't like default behavior then change it using following command.

**SET DEADLOCK\_PRIORITY { LOW | NORMAL | @deadlock\_var }**

LOW--> Low tells SQL Server that the current session should be the preferred deadlock victim, not the session that incurs the least amount of rollback resources. The standard deadlock error message 1205 is returned.

Normal--> Normal tells SQL Server to use the default deadlock method.

@deadlock\_var--> @deadlock\_var is a character variable specifying which deadlock method you want to use. Specify "3" for low, or "6" for normal.

Most well-designed applications, after receiving a deadlock message, will resubmit the aborted transaction, which most likely can now run successfully. This process, if it happens often on your server, can drag down performance. If the application has not been written to trap deadlock errors and to automatically resubmit the aborted transaction, users may very well become confused as to what is happening when they receive deadlock error messages on their computer.

Tips on how to avoid deadlocking on your SQL Server:

* Ensure the database design is properly normalized.
* During transactions, don't allow any user input. Collect it before the transaction begins.
* Avoid cursors.
* Keep transactions as short as possible.
* Reduce lock time. Try to develop your application so that it grabs locks at the latest possible time, and then releases them at the very earliest time.
* Reduce lock escalation by using the ROWLOCK or PAGLOCK.
* Consider using the NOLOCK hint to prevent locking if the data being locked is not modified often.

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

Selecting Events

----------------

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

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

Brief explanation of each of these events.

-------------------------------------------

Deadlock Graph: -->Of seven events, 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. This event 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

----------------------

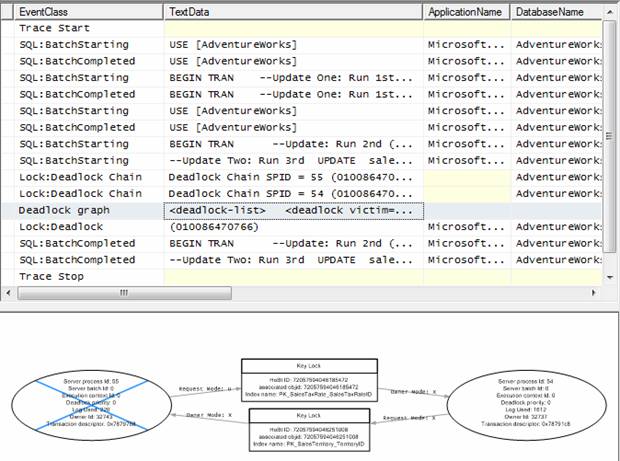
* Events
* TextData
* ApplicationName
* DatabaseName
* ServerName
* SPID
* LoginName
* BinaryData

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.

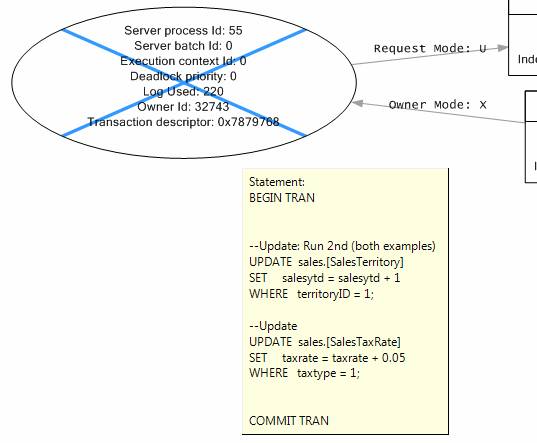


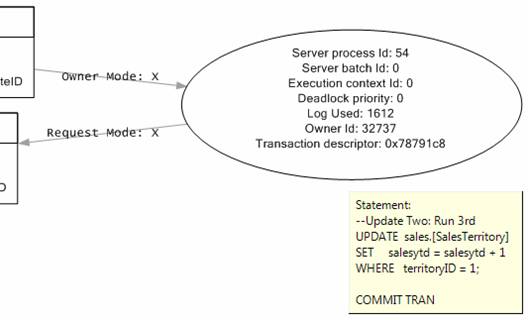
The left oval on the graph, with the blue cross, represents the transaction that was chosen as the deadlock victim by SQL Server. If you move the mouse pointer over the oval, a tooltip appears. This oval is also known as a Process Node as it represents a process that performs a specific task, such as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

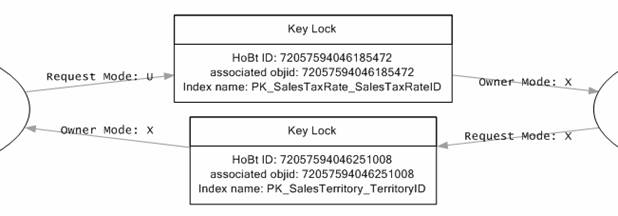
The arrows you see pointing from and to the ovals and rectangles are called Edges. An Edge represents a relationship between processes and resources. In this case, they represent types of locks each process has on each Resource Node.





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.



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.

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.