Control SQL Server locking with hints

<http://www.techrepublic.com/article/control-sql-server-locking-with-hints/5181472>

By Baseline Inc.

April 12, 2004, 7:00am PDT

Over the past few years, SQL Server has blossomed from a small office data store to an enterprise-level database server. The number of users concurrently accessing a database also increased with this upgrade. SQL Server 2000's standard approach to locking resources often seems inefficient, but thankfully it provides features to override the standard locking. Locking hints may be used to tell the server how to lock resources, but let's examine locking before covering them.  
  
What is a lock?  
Relational database systems like SQL Server use locks to prevent users from stepping on each other's toes. That is, locks prevent users from making conflicting data changes. When one user has a particular piece of data locked, no other user may modify it. In addition, a lock prevents users from viewing uncommitted data changes. Users must wait for the changes to be saved before viewing. Data may be locked using various methods. SQL Server 2000 uses locks to implement pessimistic concurrency control among multiple users performing modifications in a database at the same time.  
  
Deadlocks  
A database deadlock can occur when there is a dependency between two or more database sessions for some set of resources. A deadlock is a condition that can occur on any system with multiple threads, not just on a relational database management system. A thread in a multithreaded system may acquire one or more resources (for example, locks). If the resource being acquired is currently owned by another thread, the first thread may have to wait for the owning thread to release the target resource. The waiting thread is said to have a dependency on the owning thread for that particular resource. The following listing shows the text of an exception where a deadlock occurred:  
System.Data.SqlClient.SqlException: Transaction (Process ID 12) was deadlocked on lock resources with another process and has been chosen as the deadlock victim. Rerun the transaction.  
  
This exception was thrown when one SQL Server call conflicted with another resource that held a lock on the necessary resource. Consequently, one of the processes was terminated. This is a common error message for deadlocks with the process ID being unique to the system.  
  
Types of locks  
A database system may lock data items at one of many possible levels within the system hierarchy. The possibilities include:

* Rows—an entire row from a database table
* Pages—a collection of rows (usually a few kilobytes)
* Extents—usually a collection of a few pages
* Table—an entire database table
* Database—the entire database table is locked

Unless otherwise specified, the database uses its own judgment to determine the best locking approach based upon the scenario. Locking is a resource-intensive activity (with respect to memory), so this is not always the best approach. Thankfully, SQL Server does provide a way to circumvent the default behavior. This is accomplished with locking hints.  
  
Hints  
There are times when you need to override SQL Server's locking scheme and force a particular range of locks on a table. Transact-SQL provides a set of table-level locking hints that you can use with SELECT, INSERT, UPDATE, and DELETE statements to tell SQL Server how you want it to lock the table by overriding any other system-wide or transactional isolation levels. The available hints include the following:

* FASTFIRSTROW—The query is optimized to get the first row of the result set.
* HOLDLOCK—Hold a shared lock until the transaction has been completed.
* NOLOCK—Do not issue shared locks or recognize exclusive locks. This may result in data being returned that has been rolled back or has not been committed; therefore, working with dirty data is possible. This may only be used with the SELECT statement.
* PAGLOCK—Locks the table.
* READCOMMITTED—Read only data from transactions that have been committed. This is SQL Server's default behavior.
* READPAST—Rows locked by other processes are skipped, so the returned data may be missing rows. This may only be used with the SELECT statement.
* READUNCOMMITTED—Equivalent to NOLOCK.
* REPEATABLEREAD—Locks are placed on all data used in queries. This prevents other users from updating the data, but new phantom rows can be inserted into the data set by another user and are included in later reads in the current transaction.
* ROWLOCK—Locks the data at row level. SQL Server often locks at the page or table level to modify a row, so developers often override this setting when working with single rows.
* SERIALIZABLE—Equivalent to HOLDLOCK.
* TABLOCK—Lock at the table level. You may want to use this when performing many operations on table-level data.
* UPDLOCK—Use update locks instead of shared locks while reading a table, and hold locks until the end of the transaction. This has the advantage of allowing you to read data without locking and to update that data later knowing the data has not changed.
* XLOCK—Uses an exclusive lock on all resources until the end of the transaction.

Microsoft has two categories for the hints: granularity and isolation-level. Granularity hints include PAGLOCK, NOLOCK, ROWLOCK, and TABLOCK. On the other hand, isolation-level hints include HOLDLOCK, NOLOCK, READCOMMITTED, REPEATABLEREAD, and SERIALIZABLE. A maximum of one from each group may be used.  
  
These hints allow the consultant to control the locking used by SQL Server, and they are included in the Transact-SQL statement. They are placed in the FROM portion of the statement preceded by the WITH statement. The WITH statement is an option with SQL Server 2000, but Microsoft strongly urges its inclusion. This leads many to believe that it may be mandatory in future SQL Server releases. Here is the hint syntax as it applies to the FROM clause:  
[ FROM { < table\_source > } [ ,...n ] ]  
< table\_source > ::=  
table\_name [ [ AS ] table\_alias ] [ WITH ( < table\_hint > [ ,...n ] ) ]  
< table\_hint > ::=  
{ INDEX ( index\_val [ ,...n ] )  
| FASTFIRSTROW  
| HOLDLOCK  
| NOLOCK  
| PAGLOCK  
| READCOMMITTED  
| READPAST  
| READUNCOMMITTED  
| REPEATABLEREAD  
| ROWLOCK  
| SERIALIZABLE  
| TABLOCK  
| TABLOCKX  
| UPDLOCK  
| XLOCK    }  
  
While this syntax does show its usage, it's easier to show a real example. The following Transact-SQL statement selects all data from the Employees table of the Northwind database:  
SELECT \*  
FROM Employees WITH (nolock)  
  
This gives me all data regardless of what other processes are currently doing with it, so the data may be dirty, but this is not important to my task. Another example updates all rows in a table, setting a field to a certain value:  
UPDATE  
Employees WITH (tablock)  
SET Title='Test'  
  
This example is updating every row in the table, so a table lock is utilized.  
  
Alternate  
At this point, I must stress the fact that even though a table-level hint is specified in code, the query optimizer may ignore the hint. Table-level hints are ignored if the query optimizer does not choose the table and used in the subsequent query plan. Also, the query optimizer will often choose an indexed view over a table. Lastly, a hint may be ignored if the table contains computed columns.  
  
Use your discretion  
Using table hints in your applications depends upon what is required. Whether you use them at all will depend upon your needs. For example, many consultants love to use the FASTFIRSTROW hint to return the first row quickly. This gives them something to work with while the rest of the query completes. When the data is unlikely to change (e.g., archived data), the NOLOCK hint is a good choice since the data is basically static. On the other hand, this approach would not be good when doing financial applications, when accuracy is a must.