|  |  |
| --- | --- |
| **SQL Service**  **Avoid Locking Conflicts**  ***Reduce conflicts with a better understanding of SQL Server locking.***  **by Barry Fridley**  **FROM: https://msdn.microsoft.com/en-us/library/aa260979(v=vs.60).aspx**  Reprinted with permission from Visual Basic Programmer's Journal, April, Volume 10, Issue 4 Copyright 2000, Fawcette Technical Publications, Palo Alto, CA, USA. To subscribe, call 1-800-848-5523, 650-833-7100, visit [www.vbpj.com](http://www.vbpj.com), or visit [The Development Exchange](http://www.devx.com).  **S**QL Server uses locking to protect multiple transactions from trampling on one another. This is a critical feature for any multiuser database system. It's also a necessary one if you want to provide concurrency between multiple transactions. High concurrency means many users can experience good response time with little conflict from other users.  Locking is critical to implementing high concurrency, but using it incorrectly can impact performance significantly. Fortunately, you can manage locking effectively and create dependable applications that provide a high degree of concurrency by following a few simple programming practices. First, keep your transactions as short as possible to reduce the time locks are held. Second, design transactions to minimize deadlocks by consistently accessing shared resources in the same order. Finally, exercise caution when changing SQL Server default isolation levels and lock timeouts, test your code thoroughly, and include error handling to trap for errors.   |  | | --- | |  |   SQL Server provides several different types of locks that can be placed on specific resources (see [Table 1](https://msdn.microsoft.com/en-us/library/aa239809(v=vs.60).aspx)). In this article, I'll concentrate primarily on the shared and exclusive locks. You can find more information on these and other lock types in SQL Server Books Online (see Links).  Shared locks are acquired during read operations automatically and prevent the user from modifying data. Multiple transactions can hold shared locks on the same resource, and no other transaction can change data held with a shared lock. SQL Server releases shared locks as soon as the data has been read. It places exclusive locks on an object automatically when a user modifies the object using an Insert, Update, or Delete statement. Only one transaction at a time can hold an exclusive lock on a resource, and no locks of any kind can be placed on a resource that is already held by an exclusive lock. SQL Server holds exclusive locks for a transaction's duration. Data held with an exclusive lock is unavailable to any other transaction for the life of the original transaction.  Locks are incompatible if they cannot coexist on the same resource at the same time. Locking conflicts occur when a transaction holds a lock on a resource and another transaction attempts to place an incompatible lock on the same resource. For example, if a shared lock exists on a row, a request by another transaction for an exclusive lock on the same row is incompatible. However, a request for a shared lock on that row by another transaction is compatible and would be granted.  **Blocking Maintains Concurrency**  When a transaction requests an incompatible lock, the requesting transaction is blocked and must wait for the incompatible lock to be released before it can obtain a lock. Blocking is a normal process and necessary to maintain database concurrency. You can see this process at work by starting SQL Server's Query Analyzer and opening two separate connections to the Pubs database. Then start a transaction and modify the Authors table with the first connection:  BEGIN TRANSACTION  UPDATE AUTHORS  SET STATE='OH'  Next, run a query against the Authors table with the second connection:  SELECT \* FROM AUTHORS  SQL Server blocks the second transaction, which must wait for the first transaction to finish and release its locks. Now return to the first connection and roll back the transaction:  ROLLBACK TRANSACTION  As soon as the first transaction ends, the second transaction gets its lock and the query finishes.  Lock compatibility also enforces the hierarchy between the different levels of lockable resources. For example, assume Transaction A holds an exclusive lock on a row in Table\_X. SQL Server now prevents Transaction B from placing an exclusive lock on Table\_X as a whole, as well as the data page containing the row held by Transaction A. The exclusive lock at the row level prevents any incompatible lock from being placed on a resource at a higher level in the hierarchy.  By default, a blocked process waits indefinitely for any incompatible lock to be released. SQL Server takes an optimistic approach to locking and assumes that the locked resource will be released eventually, thereby letting the blocked process get its lock and continue. You can use the SET LOCK\_TIMEOUT statement to control how long your process waits for a lock to be released. You express the timeout value in milliseconds. For example, this code sets the timeout value to five seconds:  SET LOCK\_TIMEOUT 5000  Set options are valid only for the current connection and must be reissued every time you make a new connection to SQL Server. If the timeout value is exceeded, your transaction skips the blocked operation and receives trappable error number 1222 with the message "Lock request time out exceeded." Setting the lock timeout value to 0 tells SQL Server not to wait for any locks. In this situation, the blocked statement is cancelled and the rest of the transaction continues as soon as a locking conflict occurs.  A transaction isn't rolled back if you exceed the lock timeout. Exceeding the timeout skips the blocked operation, but any remaining SQL statements in the transaction still execute. You need to trap for error 1222 and take appropriate action if you want to rollback any user-defined transactions. This code segment demonstrates how to set the LOCK\_TIMEOUT to five seconds, trap for error 1222, and take appropriate action:  SET LOCK\_TIMEOUT 5000  BEGIN TRANSACTION  UPDATE PRODUCTS  SET QTY\_SOLD = QTY\_SOLD + 1  IF @@ERROR = 1222  GOTO BLOCKED\_ERROR  UPDATE INVENTORY  SET QTY\_ON\_HAND = QTY\_ON\_HAND - 1  IF @@ERROR = 1222  GOTO BLOCKED\_ERROR  COMMIT TRANSACTION  RETURN (0)  BLOCKED\_ERROR:  ROLLBACK TRANSACTION  RETURN (1)  **SQL Server Resolves Deadlocks**  A deadlock occurs when two transactions have locks on separate resources, and each transaction tries to place an incompatible lock on the other transaction's resource. For example, assume Transaction A obtains an exclusive lock on the Authors table and attempts to gain an exclusive lock on the Publishers table. At the same time, Transaction B acquires an exclusive lock on the Publishers table and attempts to get an exclusive lock on the Authors table. In this situation, both transactions block and wait indefinitely for the object it wants to be released.  SQL Server detects deadlocks automatically and takes appropriate action to resolve the issue. When SQL Server detects a deadlock, it selects one of the transactions as the victim automatically. SQL Server then kills the victim transaction, rolls back the entire transaction, and releases all locks the transaction holds. Terminating one of the blocking transactions enables the other transaction to acquire the necessary lock and continue. SQL Server notifies the victim by raising trappable error 1205. You should check regularly for error 1205 and resubmit the cancelled request, if necessary.  You can't always avoid deadlocks, but you can lessen their likelihood by following some simple programming rules. First, try to use resources in the same order within your transactions. For example, always reference the Authors table first and then the Publishers table. SQL Server holds exclusive locks for the transaction's duration, so always try to keep transactions as short as possible. Do this by minimizing the number of statements within the transaction and avoiding long running queries within an open transaction. Finally, use the SET DEADLOCK\_PRIORITY statement to cause a transaction to sacrifice itself as the victim when a deadlock is detected. Use the Set command to set the Deadlock Priority:  SET DEADLOCK\_PRIORITY {LOW | NORMAL}  Giving a transaction a low deadlock priority causes SQL Server to select the transaction as a victim automatically in a deadlock situation.  A transaction isolation level determines how aware a transaction is of changes made by other transactions. SQL Server provides four isolation levels: Read Committed (the default), Read Uncommitted, Repeatable Read, and Serializable. At the Read Committed level, SQL Server issues shared locks while reading data and respects exclusive locks. You cannot read uncommitted data (dirty read) in this state, and you are blocked by any outstanding exclusive locks.  A transaction under Read Uncommitted isolation neither issues any locks nor respects locks other transactions hold. In this case, it's possible for your transaction to read another transaction's uncommitted changes (dirty reads). Unfortunately, it's not uncommon for an application to receive misleading error messages when running under Read Uncommitted isolation. These messages result from the inconsistencies of working with uncommitted data. Note: You need to provide robust error handling when working at the Read Uncommitted level.  A transaction operating under Repeatable Read isolation holds all locks for the transaction's duration. Repeatable Read guarantees that another transaction cannot change any rows read in the transaction. In other words, repeating a Select statement forces SQL Server to return the same rows unaltered every time the operation is issued within the transaction.  **Maximize the Isolation**  Repeatable Read isolation protects a range of data from changes by other transactions, but it doesn't prohibit new rows from being inserted into the protected result set. Such rows are called *phantoms*. Serializable isolation is the most restrictive isolation level. It provides a simulated single-user environment within a multiuser database. Like Repeatable Read, Serializable holds all its locks for a transaction's duration. It also places *key-range* locks to prohibit another transaction from inserting new data (phantoms) into the range of data already read by a given transaction. In fact, Serializable isolation not only holds locks longer, it locks data that doesn't even exist!  You can guarantee predictable results with the more restrictive isolation levels, but this consistency comes at a cost. Locks are held longer, and the potential for blocking increases as the isolation levels become more restrictive. Use the Set command to change a transaction's isolation level:  SET TRANSACTION ISOLATION LEVEL {READ  COMMITTED | READ UNCOMMITTED |  REPEATABLE READ | SERIALIZABLE}  Now look at the effect of using Read Uncommitted isolation. Start SQL Server's Query Analyzer and open two separate connections to the Pubs database. Use the first connection to start a transaction and modify the Authors table:  BEGIN TRANSACTION  UPDATE AUTHORS  SET STATE='OH'  Next, change the isolation level and run a query against the Authors table with the second connection:  SET TRANSACTION ISOLATION LEVEL  READ UNCOMMITTED  SELECT \* FROM AUTHORS  All the rows returned have a value of OH for state field, even though the first transaction hasn't been committed. Now return to the first connection and roll back the transaction:  ROLLBACK TRANSACTION  Rerunning the second connection's Select statement causes all state values to return to their original values. However, be careful using the results of a Read Uncommitted query. Other actions taken using the results of a Read Uncommitted query are incorrect if the original transaction is rolled back.  SQL Server keeps information about all outstanding locks in the *Syslocksinfo* system table. Syslockinfo isn't a normal SQL Server table because it's maintained in memory, not on disk. The high level of activity surrounding locks would make maintaining this information on disk too slow and inefficient. You can view the current locking state within SQL Server using the SQL Server Enterprise Manager. Do this by selecting Current Activity from the Management folder. Current Activity lets you monitor Locks / Object and Locks / Process ID. Selecting Locks / Object lets you see all the locks placed on a particular lockable resource, while Selecting Locks / Process ID lets you view all processes holding locks, which processes are blocking, and which are blocked. Double clicking on a process in the right pane lets you see the last statement issued by that process (see [Figure 1](https://msdn.microsoft.com/en-us/library/aa239807(v=vs.60).aspx)). You can also get information on current locking activity using the sp\_lock stored procedure or by querying the Syslocksinfo table directly. See SQL Server Books Online for more information on sp\_lock and the Syslocksinfo table.  Some closing notes you might find useful for putting locks to good use: SQL Server uses a dynamic locking architecture to select the best lock for your transaction. It also escalates these locks to higher levels of granularity, as necessary. However, you can override SQL Server and request specific locking options in your transactions by using *lock hints* in your query. You should use lock hints only after you have a complete understanding of your app and only when you're sure the lock you ask for is better than the one SQL Server chooses. Monitoring the locking activity of your system gives you a better understanding of how locking works, making it easier to decide when and how to override the default locking behavior. |

*Barry Fridley is an MCSD, MCT developer and trainer for IS Research Inc. in Cincinnati, Ohio. He is also the manager the IS Research consulting division. Reach him at* [*bfridley@one.net*](mailto:bfridley@one.net)*.*