Locking in Microsoft SQL Server

One of the most challenging issues for developers who don’t live in RDBMS world is how to make the system working seamlessly in multi-user environment. The code which works perfectly in development and QA starts to fall apart when dozens of users access the system. There are timeouts, deadlocks and other issues that developer cannot even reproduce in house. It does not really matter that SQL Server uses row level locking, that transaction isolation level set to read uncommitted – locking, blocking and deadlocking still occurs.

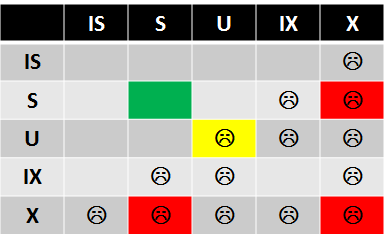
Today I’m going to start the series of the posts about locking in Microsoft SQL Server. I’ll try to explain why blocking and deadlocks occur in the system, how you can troubleshoot related problems and what should you do in order to minimize it. We will cover different transaction isolation levels and see how and why it affects behavior of the system. And talk about quite a few other things.

So let’s start with the lock types. What is the lock? In short, this is in-memory structure (64 bytes on 32 bit OS or 128 bytes on 64 bit OS). The structure has the owner, type and resource hash that links it to the resource it protects (row, page, table, file, database, etc). Obviously it’s more complicated and has quite a few other attributes, but for our practical purposes that level of details is enough.

SQL Server has more than 20 different lock types but for now let’s focus on the most important ones.

* Shared locks (S). Those locks acquired by readers during read operations such as SELECT. I’d like to mention that it happens in most part of the cases but not all the time. There are some cases when readers don’t acquire (S) locks. We will talk about it later.
* Exclusive locks (X). Those locks acquired by writers during data modification operators such as Insert, Update or Delete. Those locks prevent one object to be modified by the different sessions. Those locks are always acquired and held till end of transaction
* Update locks (U). Those locks are the mix between shared and exclusive locks. SQL Server uses them with data modification statements while searching for the rows need to be modified. For example, if you issue the statement like: “*update MyTable set Column1 = 0 where Column1 is null*” SQL Server acquires update lock for every row it processes while searching for*Column1 is null*. When eligible row found, SQL Server converts (U) lock to (X).
* Intent locks (IS, IX, IU, etc). Those locks indicate locks on the child objects. For example, if row has (X) lock, it would introduce (IX) locks on page, table and database level. Main purpose of those locks is optimization. This about situation when you need to have exclusive access to the database (i.e. (X) lock on database level). If SQL Server did not have intent locks, it would have to scan all rows in the all objects and see if there are any low level locks acquired.

Obviously the biggest question is lock compatibility. If you open [MSDN site](http://msdn.microsoft.com/en-us/library/ms186396.aspx) you’ll see nice and “easy to understand” matrix with more than 400 cells. But for our practical purpose let’s focus on the smaller version:



So what we need to remember are basically 3 things:

1. (S) locks are compatible with (S) and (U) locks.
2. (X) locks are incompatible with any other lock types
3. (U) locks are compatible with (S) but incompatible with (U)

Locks and transaction isolation levels

“What is transaction?”

Transaction is complete unit of work. Assuming you transfer money from checking account to saving, system should deduct money from the checking and add it to the saving accounts at once. Even if those are 2 independent operations, you don’t want it to “stop at the middle”, well at least in the case if bank deducts it from the checking first  If you don’t want to take that risk, you want them to work as one single action.

There is useful acronym – ACID – that describes requirements to transaction:

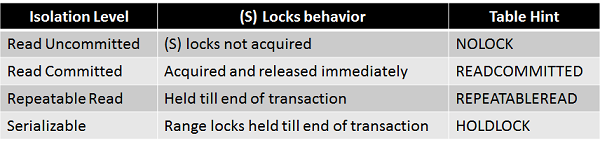
* (A) – Atomicity or “all or nothing”. Either all changes are saved or nothing changed.
* (C) – Consistency. Data remains in consistent stage all the time
* (I) – Isolation. Other sessions don’t see the changes until transaction is completed. Well, this is not always true and depend on the implementation. We will talk about it in a few minutes
* (D) – Durability. Transaction should survive and recover from the system failures

There are a few common myths about transactions in SQL Server. Such as:

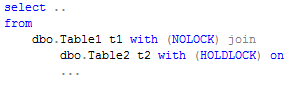
* *There are no transactions if you call insert/update/delete statements without begin tran/commit statements.* Not true. In such case SQL Server starts implicit transaction for every statement. It’s not only violate consistency rules in a lot of cases, it’s also extremely expensive. Try to run 1,000,000 insert statements within explicit transaction and without it and notice the difference in execution time and log file size.
* *There is no transactions for select statements.* Not true. SQL Server uses (lighter) transactions with select statements.
* *There is no transactions when you have (NOLOCK) hint.* Not true. (NOLOCK) hint downgrades the reader to read uncommitted isolation level but transactions are still in play.

Each transaction starts in specific transaction isolation level. There are 4 “pessimistic” isolation levels: Read uncommitted, read committed, repeatable read and serializable and 2 “optimisitic” isolation levels: Snapshot and read committed snapshot. With pessimistic isolation levels writers always block writers and typically block readers (with exception of read uncommitted isolation level). With optimistic isolation level writers don’t block readers and in snapshot isolation level does not block writers (there will be the conflict if 2 sessions are updating the same row). We will talk about optimistic isolation levels later.

Regardless of isolation level, exclusive lock (data modification) always held till end of transaction. The difference in behavior is how SQL Server handles shared locks. See the table below:

  
So, as you can see, in read uncommitted mode, shared locks are not acquired – as result, readers (select) statement can read data modified by other uncommitted transactions even when those rows held (X) locks. As result any side effects possible. Obviously it affects (S) lock behavior only. Writers still block each other.

In any other isolation level (S) locks are acquired and session is blocked when it tries to read uncommitted row with (X) lock. In read committed mode (S) locks are acquired and released immediately. In Repeatable read mode, (S) locks are acquired and held till end of transaction. So it prevents other session to modify data once read. Serializable isolation level works similarly to repeatable read with exception that locks are acquired on the range of the rows. It prevents other session to insert other data in-between once data is read.

You can control that locking behaviour with “set transaction isolation level” statement – if you want to do it in transaction/statement scope or on the table level with table hints. So it’s possible to have the statement like that:  
  
So you access Table1 in read uncommitted isolation level and Table2 in serializable isolation level.

It’s extremely easy to understand the difference between transaction isolation levels behaviour and side effects when you keep locking in mind. Just remember (S) locks behaviour and you’re all set.

# Questions About T-SQL Transaction Isolation Levels You Were Too Shy to Ask

Every time you access a relational database to make a query, you have an important decision to make: What is the appropriate isolation level for your query? If you get this wrong, the consequences can be serious. Deadlocks, Dirty reads, Non-repeatable reads, or poor performance. We're in luck, because Robert Sheldon once more answers those questions that are awkward to ask in public.

1. [“How do I set the transaction isolation level when connecting to a SQL Server database?”](https://www.red-gate.com/simple-talk/sql/t-sql-programming/questions-about-t-sql-transaction-isolation-levels-you-were-too-shy-to-ask/#first)
2. [“I still don’t get all this dirty reads and phantom reads business. What do they have to do with concurrency problems?”](https://www.red-gate.com/simple-talk/sql/t-sql-programming/questions-about-t-sql-transaction-isolation-levels-you-were-too-shy-to-ask/#second)
3. [“What is the Read Committed Snapshot isolation level and how do I enable it?”](https://www.red-gate.com/simple-talk/sql/t-sql-programming/questions-about-t-sql-transaction-isolation-levels-you-were-too-shy-to-ask/#third)
4. [“How do I set the default transaction isolation level on the current database?”](https://www.red-gate.com/simple-talk/sql/t-sql-programming/questions-about-t-sql-transaction-isolation-levels-you-were-too-shy-to-ask/#fourth)
5. [“What transaction isolation levels can I specify on a memory-optimized table?”](https://www.red-gate.com/simple-talk/sql/t-sql-programming/questions-about-t-sql-transaction-isolation-levels-you-were-too-shy-to-ask/#fifth)
6. [“How do I set the default transaction isolation level for write operations?”](https://www.red-gate.com/simple-talk/sql/t-sql-programming/questions-about-t-sql-transaction-isolation-levels-you-were-too-shy-to-ask/#sixth)
7. [“How do I enable the snapshot isolation level on my transactions?”](https://www.red-gate.com/simple-talk/sql/t-sql-programming/questions-about-t-sql-transaction-isolation-levels-you-were-too-shy-to-ask/#seventh)
8. [“The Serializable and Snapshot isolation levels appear to achieve the same results. What are the differences between them?”](https://www.red-gate.com/simple-talk/sql/t-sql-programming/questions-about-t-sql-transaction-isolation-levels-you-were-too-shy-to-ask/#eighth)
9. [“How do I verify which snapshot-related database options are enabled on my database?”](https://www.red-gate.com/simple-talk/sql/t-sql-programming/questions-about-t-sql-transaction-isolation-levels-you-were-too-shy-to-ask/#ninth)
10. [“How do I verify the transaction isolation levels that an application is using when connecting to the database?”](https://www.red-gate.com/simple-talk/sql/t-sql-programming/questions-about-t-sql-transaction-isolation-levels-you-were-too-shy-to-ask/#tenth)
11. [“What’s the difference between using the NOLOCK table hint and the Read Uncommitted transaction level?”](https://www.red-gate.com/simple-talk/sql/t-sql-programming/questions-about-t-sql-transaction-isolation-levels-you-were-too-shy-to-ask/#eleventh)
12. [“Why would I ever use the Read Uncommitted isolation level?”](https://www.red-gate.com/simple-talk/sql/t-sql-programming/questions-about-t-sql-transaction-isolation-levels-you-were-too-shy-to-ask/#twelveth)

### “How do I set the transaction isolation level when connecting to a SQL Server database?”

To set the isolation level, you can issue a SET TRANSACTION ISOLATION LEVEL statement after you connect to SQL Server. The isolation level will apply to the rest of that session, unless you explicitly change the level again.

Within the SET TRANSACTION ISOLATION LEVEL statement, you must specify one of the following five isolation levels:

* READ UNCOMMITTED: A query in the current transaction can read data modified within another transaction but not yet committed. The database engine does not issue shared locks when Read Uncommitted is specified, making this the least restrictive of the isolation levels. As a result, it’s possible that a statement will read rows that have been inserted, updated or deleted, but never committed to the database, a condition known as *dirty reads*. It’s also possible for data to be modified by another transaction between issuing statements within the current transaction, resulting in *nonrepeatable reads* or *phantom reads*.
* READ COMMITTED: A query in the current transaction cannot read data modified by another transaction that has not yet committed, thus preventing dirty reads. However, data can still be modified by other transactions between issuing statements within the current transaction, so nonrepeatable reads and phantom reads are still possible. The isolation level uses shared locking or row versioning to prevent dirty reads, depending on whether the READ\_COMMITTED\_SNAPSHOT database option is enabled. Read Committed is the default isolation level for all SQL Server databases.
* REPEATABLE READ: A query in the current transaction cannot read data modified by another transaction that has not yet committed, thus preventing dirty reads. In addition, no other transactions can modify data being read by the current transaction until it completes, eliminating nonrepeatable reads. However, if another transaction inserts new rows that match the search condition in the current transaction, in between the current transaction accessing the same data twice, phantom rows can appear in the second read.
* SERIALIZABLE: A query in the current transaction cannot read data modified by another transaction that has not yet committed. No other transaction can modify data being read by the current transaction until it completes, and no other transaction can insert new rows that would match the search condition in the current transaction until it completes. As a result, the Serializable isolation level prevents dirty reads, nonrepeatable reads, and phantom reads. However, it can have the biggest impact on performance, compared to the other isolation levels.
* SNAPSHOT: A statement can use data only if it will be in a consistent state throughout the transaction. If another transaction modifies data after the start of the current transaction, the data is not visible to the current transaction. The current transaction works with a snapshot of the data as it existed at the beginning of that transaction. Snapshot transactions do not request locks when reading data, nor do they block other transactions from writing data. In addition, other transactions writing data do not block the current transaction for reading data. As with the Serializable isolation level, the Snapshot level prevents dirty reads, nonrepeatable reads and phantom reads. However, it is susceptible to concurrent update errors. (not ANSI/ISO SQL standard)

There is much more to the isolation levels than what I’ve covered here, and the differences between them can be quite subtle, so be sure to refer to SQL Server documentation for more information. In the meantime, let’s look at how to actually specify the isolation level after making a connection. As already noted, you must use the SET TRANSACTION ISOLATION LEVELstatement, as shown in the following T-SQL statement:

|  |  |
| --- | --- |
|  | SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;    SELECT \* FROM EmployeeInfo  WHERE EmpID=1; |

Notice that we simply specify the isolation level in our SET TRANSACTION ISOLATION LEVELstatement, in this case, Read Uncommitted. We can then run our query under that isolation level. Afterwards, we can return our session to the default level by issuing the following statement:

|  |  |
| --- | --- |
|  | SET TRANSACTION ISOLATION LEVEL READ COMMITTED; |

That’s all there is to setting the isolation level. The trick is in understanding how the isolation levels work and the implications of each one. Tread carefully when changing the isolation level. You can easily impact performance if you use a level too restrictive, or create application problems by implementing an isolation level that leads to transactions prone to concurrency issues.

### “I still don’t get all this dirty reads and phantom reads business. What do they have to do with concurrency problems?”

In the previous question, we touched upon the issue of dirty reads, nonrepeatable reads and phantom reads and how they’re tied to transaction isolation levels. Perhaps a different take on each of these would be helpful:

* Dirty read: Session 1 begins a transaction and modifies data. Session 2 reads the modified data before Session 1 commits the transaction. Session 2 is reading data that exists only in theory because the Session 1 transaction might be rolled back. If that occurs, the data read by Session 2 is no longer valid.
* Nonrepeatable read: Session 1 begins a transaction and retrieves a row of data from a table. Session 2 updates that row. Session 1 tries to retrieve the row once more, still within the original transaction, but receives different results because of the Session 2 update.
* Phantom read: Session 1 begins a transaction and retrieves several rows of data from a table. Session 2 inserts a row into the same table, and the insert coincides with the search criteria of the Session 1 statement. When Session 1 reruns the query, the new row added by Session 2 is now included in the results.

Notice the similarities between nonrepeatable reads and phantom reads. In fact, you can think of a phantom read as a variation of the nonrepeatable read. The problem with these definitions, however, is that it can be difficult to get agreement on any one being the “correct” one, particularly when it comes to phantom reads and nonrepeatable reads. Even within Microsoft documentation the definitions are not consistent. However, the scenarios described here should at least give you some sense of the differences between these concurrency problems.

The important point to remember is that the possibility of running into any of these issues depends on the isolation levels you’re using when you run your queries. The following table shows the concurrency issues that each isolation level is susceptible to:

|  |  |  |  |
| --- | --- | --- | --- |
| Isolation level | Dirty read | Nonrepeatable read | Phantom read |
| Read uncommitted | Y | Y | Y |
| Read committed | N | Y | Y |
| Repeatable read | N | N | Y |
| Serializable | N | N | N |
| Snapshot | N | N | N |

Let’s look at a few examples to better illustrate how the isolation levels can help prevent concurrency problems. The examples are based on the test EmployeeInfo table, which includes the EmpID, FirstName, and LastName columns. However, you can use whatever table you like, as long as you can update data in it.

In the first example, we simulate a dirty read against the EmployeeInfo table. (If you want to try this out, you’ll need to establish two different connections to your database.) Let’s start with Session 1. First, we begin a transaction and run an UPDATE statement, as shown in the following T-SQL:

|  |  |
| --- | --- |
|  | -- session 1  BEGIN TRANSACTION;    UPDATE EmployeeInfo  SET FirstName='Frank'  WHERE EmpID=1;    WAITFOR DELAY '00:00:05'    ROLLBACK TRANSACTION; |

Notice that the transaction includes a WAITFOR DELAY statement, with five seconds specified. This is included to give us time to run our statements in the second session. You can take this approach, adjusting the seconds as necessary, or simply run the BEGIN TRANSACTION and UPDATE statements and stop there. You can then run the statements in Session 2, and then return to Session 1 to run the ROLLBACK statement. The goal is to make sure we run the statements in the second session in between the UPDATE and ROLLBACK statements in the first session.

When we hit the delay, we can run the following the following T-SQL statements in Session 2:

|  |  |
| --- | --- |
|  | --session 2  SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;    SELECT FirstName FROM EmployeeInfo  WHERE EmpID=1; |

We start by specifying that the Read Uncommitted isolation level be used. We then run the SELECT statement, which immediately returns the value Frank, the new value inserted into the table in Session 1. However, if we run the SELECT statement a second time, after the Session 1 transaction has been rolled back, we’ll receive the value Ken, the original value in the table. The returned value Frank in Session 2 is an example of a dirty read.

Now let’s repeat our experiment, only this time with a different isolation level. In Session 1, we again start our transaction and do our update:

|  |  |
| --- | --- |
|  | -- session 1  BEGIN TRANSACTION;    UPDATE EmployeeInfo  SET FirstName='Frank'  WHERE EmpID=1;    WAITFOR DELAY'00:00:05'    ROLLBACK TRANSACTION; |

This time in Session 2, we set the transaction level to Read Committed:

|  |  |
| --- | --- |
|  | --session 2  SET TRANSACTION ISOLATION LEVEL READCOMMITTED;    SELECT FirstName FROM EmployeeInfo  WHERE EmpID=1; |

When we run the SELECT statement, it waits until the Session 1 transaction has been rolled back and then returns the value Ken. As you can see, no dirty read. But another problem is lurking: the nonrepeatable read.

To demonstrate the nonrepeatable read, we take a different approach. In Session 1, we set the isolation level to Read Committed (which is the default), start a transaction, and retrieve a row from the EmployeeInfo table:

|  |  |
| --- | --- |
|  | --session 1  SET TRANSACTION ISOLATION LEVEL READCOMMITTED;    BEGIN TRANSACTION;    SELECT FirstName FROM EmployeeInfo  WHER EEmpID=1;    WAITFOR DELAY'00:00:05'    SELECT FirstName FROM EmployeeInfo  WHERE EmpID=1;    ROLLBACK TRANSACTION; |

During the delay, we run an UPDATE statement in Session 2:

|  |  |
| --- | --- |
|  | --session 2  UPDATE EmployeeInfo  SET FirstName='Frank'  WHERE EmpID=1; |

This time around, the first SELECT statement in Session 1 returns the value Ken, the original value. However, the second time the SELECT statement runs-after the five-second delay and the Session 2 update, the statement returns the value Frank, thus demonstrating a nonrepeatable read.

Let’s do our experiment again, only this time specify the Repeatable Read isolation level in Session 1:

|  |  |
| --- | --- |
|  | --session 1SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;    BEGIN TRANSACTION;    SELECT FirstName FROM EmployeeInfo  WHERE EmpID=1;    WAITFOR DELAY'00:00:05'    SELECT FirstName FROM EmployeeInfo  WHERE EmpID=1;    ROLLBACK TRANSACTION; |

In Session 2, during the Session 1 delay, we run the same UPDATE statement as in the previous example:

|  |  |
| --- | --- |
|  | --session 2  UPDATE EmployeeInfo  SET FirstName='Ken'  WHERE EmpID=1; |

In this case, both executions of the SELECT statement in Session 1 return the value Frank, the last value to be inserted (from the preceding example). The UPDATE statement will not run until the Session 1 transaction has been rolled back. If we were to then run the SELECT statement again, after the update, it would return Ken, as expected.

We could create more examples to demonstrate various scenarios, but you get the picture. The isolation level determines which read concurrency problems can occur. It’s up to you to decide how strictly you want to control access. Just remember, the stricter your isolation policy, the longer the database engine will lock your data and the greater the impact on performance.

### “What is the Read Committed Snapshot isolation level and how do I enable it?”

In some SQL Server documentation, you’ll see references to the Read Committed Snapshot isolation level as a sixth type. What this actually refers to is the READ\_COMMITTED\_SNAPSHOTdatabase option. When a database is first created, this option is set to OFF, so it has no impact on the database or the existing isolation levels. When set to ON, however, the option modifies the Read Committed isolation level to behave more like the Snapshot isolation level, thus the label Read Committed Snapshot.

By default, when the Read Committed transaction level is enabled, the database uses shared locks to prevent other transactions from modifying rows when the current transaction is reading those rows. At the same time, the locks block the current transaction from reading rows modified by other running transactions until those transactions have completed.

When the READ\_COMMITTED\_SNAPSHOT database option is set to ON, the database engine uses row versioning for Read Committed to ensure transactionally consistency, rather than using shared locks. The engine stores the row versions in the tempdb system database as long as necessary to facilitate this process. In this way, the current read operation does not block other transactions, or vice versa.

To enable the READ\_COMMITTED\_SNAPSHOT option on a database, you can run an ALTERDATABASE statement similar to the one shown in the following example:

|  |  |
| --- | --- |
|  | ALTER DATABASE AdventureWorks2014  SET READ\_COMMITTED\_SNAPSHOTON; |

Notice that we need only set the option to ON. To disable the option, simply set it to OFF:

|  |  |
| --- | --- |
|  | ALTER DATABASE AdventureWorks2014  SET READ\_COMMITTED\_SNAPSHOTOFF; |

The READ\_COMMITTED\_SNAPSHOT option can help avoid locking contention and subsequently improve query performance by minimizing wait times, but it does not eliminate the risks of non-repeatable reads and phantom reads. For that, you need the Snapshot isolation level. The challenge with Snapshot, however, is that it can result in update conflict errors. With Read Committed Snapshot, the database engine will block a connection until the active connection releases its locks, and then it will update the data.

Also be aware that enabling the READ\_COMMITTED\_SNAPSHOT option-and the subsequent row versioning that goes with it-incurs additional overhead in maintaining the versioned rows in tempdb, as is the case with the Snapshot isolation level. You must also ensure that tempdb has plenty of room to accommodate all the versioned data.

### “How do I set the default transaction isolation level on the current database?”

You can’t. The default isolation level for all SQL Server databases is Read Committed, and your only option is to set the isolation level within a session, if you want to use a level other than the default. Workarounds have been suggested, such as creating a logon trigger that sets the isolation level, but these approaches tend not to deliver the expected results.

That said, SQL Server does support several database options that can impact the isolation levels: READ\_COMMITTED\_SNAPSHOT, ALLOW\_SNAPSHOT\_ISOLATION, and MEMORY\_OPTIMIZED\_ELEVATE\_TO\_SNAPSHOT. All of these are disabled by default, but you can easily use T-SQL to turn them on, as shown in the following examples:

|  |  |
| --- | --- |
|  | ALTER DATABASE AdventureWorks2014  SET READ\_COMMITTED\_SNAPSHOT ON;    ALTER DATABASE AdventureWorks2014  SET ALLOW\_SNAPSHOT\_ISOLATION ON;    ALTER DATABASE AdventureWorks2014  SET MEMORY\_OPTIMIZED\_ELEVATE\_TO\_SNAPSHOTON; |

The READ\_COMMITTED\_SNAPSHOT option affects the behavior of the Read Committed isolation level, as previously mentioned, and the ALLOW\_SNAPSHOT\_ISOLATION option controls whether the Snapshot isolation level can be enabled during a session. When you set either of these options to ON, the database engine maintains row versions in the tempdb database to meet the requirements of transactions running under the Read Committed Snapshot or Snapshot isolation level.

The MEMORY\_OPTIMIZED\_ELEVATE\_TO\_SNAPSHOT option is specific to memory-optimized tables, new in SQL Server 2014. When the option is enabled, the database engine uses Snapshot isolation for all interpreted T-SQL operations on memory-optimized tables, whether or not the isolation level is set explicitly at the session level. (Interpreted T-SQL refers to batches or stored procedures other than natively compiled procedures.)

You can disable any of these options by setting them to OFF, as shown in the following ALTERDATABASE statements:

|  |  |
| --- | --- |
|  | ALTERDATABASEAdventureWorks2014  SETREAD\_COMMITTED\_SNAPSHOTOFF;    ALTERDATABASEAdventureWorks2014  SETALLOW\_SNAPSHOT\_ISOLATIONOFF;    ALTERDATABASEAdventureWorks2014  SETMEMORY\_OPTIMIZED\_ELEVATE\_TO\_SNAPSHOTOFF; |

Other than these options, there’s not much else you can do about the default behavior. You’re stuck with Read Committed whether or not row versioning is used. To get around this, you must use the SET TRANSACTION ISOLATION LEVEL statement at the session level, or use a table hint at the statement level, if you want your change to apply only to that statement. For example, the following SELECT statement specifies the TABLOCK table hint:

|  |  |
| --- | --- |
|  | SELECTEmpID,FirstName,LastName  FROMEmployeeInfoWITH(TABLOCK)  WHEREEmpID>99  ORDERBYLastName; |

The TABLOCK table hint directs the database engine to lock the data at the table level, rather than the row or page level. The table hint will apply only to the table targeted in this statement and will not impact the rest of the session, as would a SET TRANSACTION ISOLATION LEVEL statement.

You can also use table hints for a specific table in a join. For example, the following SELECTstatement specifies the SERIALIZABLE table hint for the SalesOrderDetail table:

|  |  |
| --- | --- |
|  | SELECTd.ProductID,d.OrderQty,h.OrderDate  FROMSales.SalesOrderHeaderhINNERJOIN   Sales.SalesOrderDetaildWITH(SERIALIZABLE)   ONh.SalesOrderID=d.SalesOrderID  ORDERBYd.ProductID,h.OrderDateDESC; |

This tells the database engine to apply the Serializable isolation level only to the SalesOrderDetail table for this query. The Read Committed isolation level (the default) still applies to the SalesOrderHeader table. As a result, a shared lock will be held on the SalesOrderDetail table for the entire transaction, rather than just the initial table read.

### “What transaction isolation levels can I specify on a memory-optimized table?”

Memory-optimized tables are part of the new In-Memory OLTP technology introduced in SQL Server 2014 to help improve application performance. Memory-optimized tables support the Read Committed, Repeatable Read, Serializable, and Snapshot isolation levels, with Read Committed the default level, as with disk-based tables. Notice that the Read Uncommitted isolation level is not included.

All isolation levels supported for memory-optimized tables are based on row versioning. The tables rely on multiple row versions to guarantee isolation, providing optimistic concurrency control. As a result, the database engine does not issue shared data locks. Row versions are not stored in tempdb like the Snapshot isolation level for disk-based tables, but are instead part of the memory-optimized table itself.

Isolation levels used for memory-optimized tables provide the same guarantees as those used for disk-based tables; only the processes used to get there are different. Because the isolation levels implement row versioning instead of shared locks, a write conflict will occur when two transactions attempt to update the same row concurrently, rather than one transaction waiting for the other to complete before attempting the update.

If you’re working with memory-optimized tables, there are a number of guidelines you should follow when specifying an isolation level. For example, you should avoid long-running transactions and implement retry logic in your apps to deal with conflicts, validation errors, and commit-dependency failures. You can find details about the various guidelines in the MSDN article “[Guidelines for Transaction Isolation Levels with Memory-Optimized Tables](http://msdn.microsoft.com/en-us/library/dn133187.aspx).”

Keep in mind, however, that SQL Server’s in-memory technologies are still in their infancy. Consequently, the available information about the technology is often confusing and inconsistent. If you plan to implement in-memory tables, be sure to test your operations thoroughly and compare the behavior for different isolation levels. No doubt we’ll see changes in the technology and guidelines going forward.

### “How do I set the default transaction isolation level for write operations?”

It’s best to think of the isolation levels in terms of read operations, rather than write operations. The database engine issues exclusive locks on all write operations and holds the locks until the transactions have completed, regardless of which isolation level is used. This ensures that your write operations are always protected. Concurrency issues such as dirty reads and phantom reads, as their names suggest, apply to read operations, not write. The database engine governs the locking behavior of all write operations, and you cannot change that behavior at the database level.

Even for read operations, you have few options for changing the default behavior, as noted earlier. When a transaction initiates a read option, the database engine uses either shared locks or row versions (to support Snapshot isolation) in order to handle concurrency issues. The way in which the database engine issues shared locks depends on the isolation level. For example, in a transaction running at the Read Uncommitted isolation level, the database engine issues no shared locks. However, if the transaction is running at the Serializable level, the database engine will lock the selected data throughout the transaction.

Multiple shared locks can exist on the selected data at any one time, but only one exclusive lock can exist on data at a time. Exclusive locks are not compatible with shared locks. Once a transaction has taken an exclusive lock, the database engine cannot issue a shared lock on that data until the transaction has completed.

There are a number of other rules that govern locking, but the point is, you cannot change the database’s default behavior when it comes to write operations. You can, however, override that behavior by using table hints, just like you can use table locks for read operations. For example, the following UPDATE statement specifies the TABLOCKX table hint:

|  |  |
| --- | --- |
|  | UPDATEEmployeeInfoWITH(TABLOCKX)  SETFirstName='Ken'  WHEREEmpID=1; |

The TABLOCKX table hint tells the database engine to issue an exclusive lock on the entire table, rather than at the row or page level, as would normally be the case for this type of statement.

### “How do I enable the snapshot isolation level on my transactions?”

As noted previously, the Snapshot isolation level relies on row versions being maintained in the tempdb database to avoid locking contention issues. This approach incurs overhead to maintain those versioned rows. For this reason, you must explicitly enable snapshot isolation at the database level before you can implement it at the session level.

To enable snapshot isolation, you can run an ALTER DATABASE statement to set the ALLOW\_SNAPSHOT\_ISOLATION option to ON, as shown in the following example:

|  |  |
| --- | --- |
|  | ALTERDATABASEAdventureWorks2014  SETALLOW\_SNAPSHOT\_ISOLATIONON; |

When you enable the ALLOW\_SNAPSHOT\_ISOLATION option, you activate a mechanism in your database for storing the versioned rows in tempdb. You can then set the Snapshot isolation level within your session for specific transactions:

|  |  |
| --- | --- |
|  | SETTRANSACTIONISOLATIONLEVELSNAPSHOT;    BEGINTRANSACTION;    SELECTFirstNameFROMEmployeeInfo  WHEREEmpID=1;    COMMITTRANSACTION; |

A transaction using the Snapshot isolation level sees the data as it existed at the beginning of the transaction. As a result, if you try to update data that has changed during the course of the transaction, the database engine rolls back the transaction and raises an error.

### “The Serializable and Snapshot isolation levels appear to achieve the same results. What are the differences between them?”

Although the Serializable and Snapshot isolation levels work differently, they both protect against dirty reads, nonrepeatable reads, and phantom reads. The Serializable isolation level relies on shared data locks to address concurrency issues, and the Snapshot isolation level relies on row versioning. (It’s true that all isolation levels use row versioning for memory-optimized tables, but for now, let’s stick with disk-based tables.)

The key to understanding the differences between the two isolation levels is to look at what happens when data gets updated. With the Serializable isolation level, the database engines issues a shared lock on the accessed data and holds that lock until the transaction has completed. As a result, no other transactions can modify the data as long as the current transaction is active.

With the Snapshot isolation level, the database engine does not lock the data. Other transactions can update the data at the same time as the current transaction, resulting in update conflicts. Let’s try another little experiment to demonstrate how this works. In Session 1, we set the connection to the Serializable isolation level, begin a transaction and issue several statements:

|  |  |
| --- | --- |
|  | --session 1  SETTRANSACTIONISOLATIONLEVELSERIALIZABLE;    BEGINTRANSACTION;    SELECTFirstNameFROMEmployeeInfo  WHEREEmpID=1;    WAITFORDELAY'00:00:05'    UPDATEEmployeeInfo  SETFirstName='Roger'  WHEREEmpID=1;    SELECTFirstNameFROMEmployeeInfo  WHEREEmpID=1;    COMMITTRANSACTION; |

As in our previous examples, we’re using a WAITFOR DELAY statement to give us time to run the second session. Before the delay, we retrieve the data (Session 1). After the delay, we update the data and retrieve it once again. During the delay, we run our second session:

|  |  |
| --- | --- |
|  | --session 2  UPDATEEmployeeInfo  SETFirstName='Harold'  WHEREEmpID=1;    SELECTFirstNameFROMEmployeeInfo  WHEREEmpID=1; |

In Session 2, we’re simply updating the same row as in Session 1 and then retrieving the results. Because we’re using the Serializable isolation level in Session 1, the statements in Session 2 will not run until the Session 1 transaction has completed. Consequently, the first SELECT statement in Session 1 returns Ken, the original value, and the second SELECT statement returns the value Roger. The SELECT statement in Session 2 then returns the value Harold.

If we repeat the experiment, but instead specify the Snapshot isolation level in Session 1, we will receive different results. The first SELECT statement in Session 1 will still return the value Ken(assuming we reset that value to the original), and the SELECT statement in Session 2 will return the value Harold. However, when we run the UPDATE statement in Session 1, we receive the following error:

|  |  |
| --- | --- |
|  | Msg3960,Level16,State6,Line297  Snapshotisolationtransactionabortedduetoupdateconflict.Youcannotusesnapshotisolationtoaccesstable'dbo.EmployeeInfo'directlyorindirectlyindatabase'AdventureWorks2014'toupdate,delete,orinserttherowthathasbeenmodifiedordeletedbyanothertransaction.Retrythetransactionorchangetheisolationlevelfortheupdate/deletestatement. |

As you can see, the Snapshot isolation level doesn’t handle concurrent updates as gracefully as the Serializable isolation level. However, Snapshot can offer improved query performance because the queries don’t have to wait for locks to be released before being able to access the data. If trying to decide which of the two to use, you’ll have to weigh query performance against tempdb overhead, as well as take into consideration the type of queries you’ll be running against your tables.

### “How do I verify which snapshot-related database options are enabled on my database?”

You can verify the snapshot-related database options by using the sys.databases catalog view to retrieve the information on a specific database. For example, if you want to know the setting for the ALLOW\_SNAPSHOT\_ISOLATION option, you would query the snapshot\_isolation\_state\_desc column in the sys.databases view, as shown in the following SELECT statement:

|  |  |
| --- | --- |
|  | SELECTsnapshot\_isolation\_state\_desc  FROMsys.databases  WHEREname='AdventureWorks2014' |

The statement will return the value ON if the option is enabled or OFF if it’s disabled. You can just as easily retrieve the current setting for the READ\_COMMITTED\_SNAPSHOT option by querying the is\_read\_committed\_snapshot\_on column:

|  |  |
| --- | --- |
|  | SELECTis\_read\_committed\_snapshot\_on  FROMsys.databases  WHEREname='AdventureWorks2014' |

In this case, the statement will return the value 1 if the option is enabled or 0 if it’s disabled. To retrieve the current setting of the MEMORY\_OPTIMIZED\_ELEVATE\_TO\_SNAPSHOT option, use the is\_memory\_optimized\_elevate\_to\_snapshot\_on column:

|  |  |
| --- | --- |
|  | SELECTis\_memory\_optimized\_elevate\_to\_snapshot\_on  FROMsys.databases  WHEREname='AdventureWorks2014' |

Once again, the statement will return the value 1 if the option is enabled or 0 if it’s disabled.

### “How do I verify the transaction isolation levels that an application is using when connecting to the database?”

You can use the sys.dm\_exec\_sessions dynamic management view (DMV) to verify a connection’s isolation level. For example, suppose you have an application that issues the following statements:

|  |  |
| --- | --- |
|  | SETTRANSACTIONISOLATIONLEVELREADUNCOMMITTED;    SELECTFirstNameFROMEmployeeInfo  WHEREEmpID=1; |

The application sets the isolation level to Read Uncommitted and then retrieves data from the EmployeeInfo table. You can run the following statement (within the same session) to return the isolation level:

|  |  |
| --- | --- |
|  | SELECTtransaction\_isolation\_level  FROMsys.dm\_exec\_sessions  WHEREsession\_id=@@SPID; |

The SELECT statement retrieves the transaction\_isolation\_level column from the DMV. The statement also includes a WHERE clause that uses the @@SPID system variable to specify the current session ID.

In this case, the SELECT statement returns a value of 1. SQL Server uses the following values to represent the isolation levels available through the sys.dm\_exec\_sessions view:

* 0 = Unspecified
* 1 = Read Uncommitted
* 2 = Read Committed
* 3 = Repeatable
* 4 = Serializable
* 5 = Snapshot

If you can’t query the sys.dm\_exec\_sessions view within the session whose isolation level you want to verify, you’ll need to take a different approach to identify the application and its session. For example, the view supports the host\_name, login\_name, and program\_namecolumns. Perhaps querying one of these will give you the information you need.

### “What’s the difference between using the NOLOCK table hint and the Read Uncommitted transaction level?”

Both approaches tell the database engine not to issue shared locks when reading the requested data; however, the two approaches differ in scope, within the context of the current session. For example, suppose we issue the following T-SQL statements:

|  |  |
| --- | --- |
|  | SETTRANSACTIONISOLATIONLEVELREADUNCOMMITTED;    SELECTFirstName,LastName  FROMEmployeeInfo  WHEREEmpID=1; |

In this case, we’re setting the connection to the Read Uncommitted isolation level before running our statement. The entire session will use this isolation level until the session ends or we explicitly change the level. That means, for every table we access throughout the session, no shared locks are placed on any of the tables for our read operations. Consequently, our transactions are not protected from dirty reads as well as nonrepeatable or phantom reads.

However, we might decide that only one or two of our target tables require the “lock-less” state, in which case, we can use the NOLOCK table hint to limit that isolation level to the specific query, as shown in the following example:

|  |  |
| --- | --- |
|  | SELECTFirstName,LastName  FROMEmployeeInfoWITH(NOLOCK)  WHEREEmpID=1; |

The NOLOCK table hint is the equivalent of the READUNCOMMITTED table hint. The database engine ignores the hint if used in the FROM clause of a DELETE or UPDATE statement.

### “Why would I ever use the Read Uncommitted isolation level?”

When the Read Uncommitted isolation level is used, the database engine does not issue shared locks on the accessed data, which can lead to the types of concurrency issues described earlier. That said, the Read Uncommitted isolation level can be useful in certain situations, particularly if you’re trying to avoid heavy lock contention.

For example, if you’re pulling historical data from a data warehouse or reporting database, your queries might benefit from the minimal locking, especially those complex joins that can easily escalate to table locks. In such a scenario, the primary DML operations might be inserts only and often not part of the queries. Plus, you might also be able to schedule your data loads to occur at a time when the data is not being accessed for reporting and analytics.

Another way to look at this is that the Read Uncommitted isolation level can be handy if you’re querying data that is not likely to change or you’re aggregating large quantities of data that permit an acceptable margin for error. On the other hand, if the data is time-sensitive or requires drill-down precision, you should forego Read Uncommitted, especially when an application frequently writes against the data. Your typical OLTP operation, for example, is usually a lousy candidate for Read Uncommitted, except for simple queries and small tables with fairly static data.

Developers can also benefit from the Read Uncommitted isolation level because they can use it to help debug a transaction by allowing them to see what is happening within the transaction while it’s still running. In this case, the dirty read can provide line-by-line insight into a transaction, which can be especially useful for those overly complex stored procedures.

Despite the advantages in performance that Read Uncommitted can offer in certain situations, you should use this isolation level judiciously. Make sure you understand your data and your queries, and that you’re not using Read Uncommitted to cover up for other problems.