Isolation levels in SQL Server control the way locking works between transactions.

SQL Server 2008 supports the following isolation levels

* Read Uncommitted
* Read Committed (The default)
* Repeatable Read
* Serializable
* Snapshot

Before I run through each of these in detail you may want to create a new database to run the examples, run the following script on the new database to create the sample data. **Note** : You’ll also want to drop the IsolationTests table and re-run this script before each example to reset the data.

|  |
| --- |
| CREATE TABLE IsolationTests (  Id INT IDENTITY,  Col1 INT,  Col2 INT,  Col3 INT  )    INSERT INTO IsolationTests(Col1,Col2,Col3)  SELECT 1,2,3  UNION ALL SELECT 1,2,3  UNION ALL SELECT 1,2,3  UNION ALL SELECT 1,2,3  UNION ALL SELECT 1,2,3  UNION ALL SELECT 1,2,3  UNION ALL SELECT 1,2,3 |

Also before we go any further it is important to understand these two terms….

1. **Dirty Reads** – This is when you read uncommitted data, when doing this there is no guarantee that data read will ever be committed meaning the data could well be bad.
2. **Phantom Reads** – This is when data that you are working with has been changed by another transaction since you first read it in. This means subsequent reads of this data in the same transaction could well be different.

**Read Uncommitted**

This is the lowest isolation level there is. Read uncommitted causes no shared locks to be requested which allows you to read data that is currently being modified in other transactions. It also allows other transactions to modify data that you are reading.

As you can probably imagine this can cause some unexpected results in a variety of different ways. For example data returned by the select could be in a half way state if an update was running in another transaction causing some of your rows to come back with the updated values and some not to.

To see read uncommitted in action lets run Query1 in one tab of Management Studio and then quickly run Query2 in another tab before Query1 completes.

Query1

|  |
| --- |
| BEGIN TRAN  UPDATE Tests SET Col1 = 2  --Simulate having some intensive processing here with a wait  WAITFOR DELAY '00:00:10'  ROLLBACK |

Query2

|  |
| --- |
| SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED  SELECT \* FROM IsolationTests |

Notice that Query2 will not wait for Query1 to finish, also more importantly Query2 returns dirty data. Remember Query1 rolls back all its changes however Query2 has returned the data anyway, this is because it didn’t wait for all the other transactions with exclusive locks on this data it just returned what was there at the time.

There is a syntactic shortcut for querying data using the read uncommitted isolation level by using the NOLOCK table hint. You could change the above Query2 to look like this and it would do the exact same thing.

|  |
| --- |
| SELECT \* FROM IsolationTests WITH(NOLOCK) |

**Read Committed**

This is the default isolation level and means selects will only return committed data. Select statements will issue shared lock requests against data you’re querying this causes you to wait if another transaction already has an exclusive lock on that data. Once you have your shared lock any other transactions trying to modify that data will request an exclusive lock and be made to wait until your Read Committed transaction finishes.

You can see an example of a read transaction waiting for a modify transaction to complete before returning the data by running the following Queries in separate tabs as you did with Read Uncommitted.

Query1

|  |
| --- |
| BEGIN TRAN  UPDATE IsolationTests SET Col1 = 2  --Simulate having some intensive processing here with a wait  WAITFOR DELAY '00:00:10'  ROLLBACK |

Query2

|  |
| --- |
| SELECT \* FROM IsolationTests |

Notice how Query2 waited for the first transaction to complete before returning and also how the data returned is the data we started off with as Query1 did a rollback. The reason no isolation level was specified is because Read Committed is the default isolation level for SQL Server. If you want to check what isolation level you are running under you can run “DBCC useroptions”. Remember isolation levels are Connection/Transaction specific so different queries on the same database are often run under different isolation levels.

**Repeatable Read**

This is similar to Read Committed but with the additional guarantee that if you issue the same select twice in a transaction you will get the same results both times. It does this by holding on to the shared locks it obtains on the records it reads until the end of the transaction, This means any transactions that try to modify these records are force to wait for the read transaction to complete.

As before run Query1 then while its running run Query2

Query1

|  |
| --- |
| SET TRANSACTION ISOLATION LEVEL REPEATABLE READ  BEGIN TRAN  SELECT \* FROM IsolationTests  WAITFOR DELAY '00:00:10'  SELECT \* FROM IsolationTests  ROLLBACK |

Query2

|  |
| --- |
| UPDATE IsolationTests SET Col1 = -1 |

Notice that Query1 returns the same data for both selects even though you ran a query to modify the data before the second select ran. This is because the Update query was forced to wait for Query1 to finish due to the exclusive locks that were opened as you specified Repeatable Read.

If you rerun the above Queries but change Query1 to Read Committed you will notice the two selects return different data and that Query2 does not wait for Query1 to finish.

One last thing to know about Repeatable Read is that the data can change between 2 queries if more records are added. Repeatable Read guarantees records queried by a previous select will not be changed or deleted, it does not stop new records being inserted so it is still very possible to get Phantom Reads at this isolation level.

**Serializable**

This isolation level takes Repeatable Read and adds the guarantee that no new data will be added eradicating the chance of getting Phantom Reads. It does this by placing range locks on the queried data. This causes any other transactions trying to modify or insert data touched on by this transaction to wait until it has finished.

You know the drill by now run these queries side by side…

Query1

|  |
| --- |
| SET TRANSACTION ISOLATION LEVEL SERIALIZABLE  BEGIN TRAN  SELECT \* FROM IsolationTests  WAITFOR DELAY '00:00:10'  SELECT \* FROM IsolationTests  ROLLBACK |

Query2

|  |
| --- |
| INSERT INTO IsolationTests(Col1,Col2,Col3)  VALUES (100,100,100) |

You’ll see that the insert in Query2 waits for Query1 to complete before it runs eradicating the chance of a phantom read. If you change the isolation level in Query1 to repeatable read, you’ll see the insert no longer gets blocked and the two select statements in Query1 return a different amount of rows.

**Snapshot**

This provides the same guarantees as serializable. So what’s the difference? Well it’s more in the way it works, using snapshot doesn’t block other Queries from inserting or updating the data touched by the snapshot transaction. Instead it creates it’s own little snapshot of the data being read at that time, if you then read that data again in the same transaction it reads it from its snapshot, This means that even if another transaction has made changes you will always get the same results as you did the first time you read the data.

So on the plus side your not blocking anyone else from modifying the data whilst you run your transaction but…. You’re using extra resources on the SQL Server to allocate each snapshot transaction the additional resources to store the snapshot data which can be quite significant if your transaction is working with a large amount of data.

To use the snapshot isolation level you need to enable it on the database by running the following command

|  |
| --- |
| ALTER DATABASE IsolationTests  SET ALLOW\_SNAPSHOT\_ISOLATION ON |

If you rerun the examples from serializable but change the isolation level to snapshot you will notice that you still get the same data returned but Query2 no longer waits for Query1 to complete.

**Summary**

You should now have a good idea how each of the different isolation levels work. You can see how the higher the level you use the less concurrency you are offering and the more blocking you bring to the table. You should always try to use the lowest isolation level you can which is usually read committed.

**TRY CATCH**

SQL Server 2005 also introduced the TRY...CATCH construct, which you can use within T-SQL code to provide a more graceful mechanism for exception handling than was available in previous versions of SQL Server. In versions prior to 2005, error handling was typically done by checking @@ERROR after each SQL statement and often using the GOTO statement to branch to an error-handling routine.

A TRY...CATCH construct consists of two parts: a TRY block and CATCH block. When an error condition is detected in a T-SQL statement that is inside a TRY block, control is immediately passed to the CATCH block, where the error is processed. T-SQL statements in the TRY block that follow the statement that generated the error are not executed.

If an error occurs and processing is passed to the CATCH block, after the statements in the CATCH block are executed, control is transferred to the first T-SQL statement that follows the END CATCH statement. If there are no errors inside the TRY block, control is passed to the statement immediately after the associated END CATCH statement, essentially skipping over the statements in the CATCH block.

A TRY block is initiated with the BEGIN TRY statement and ended with the END TRY statement and can consist of one or more Transact-SQL statements between the BEGIN TRY and END TRY statements. The TRY block must be followed immediately by a CATCH block. A CATCH block is indicated with the BEGIN CATCH statement and ended with the END CATCH statement and can consist of one or more SQL statements. In SQL Server, each TRY block can be associated with only one CATCH block.

The syntax of the TRY...CATCH construct is as follows:

BEGIN TRY

--one\_or\_more\_sql\_statements

END TRY

BEGIN CATCH

--one\_or\_more\_sql\_statements

END CATCH

In a CATCH block, you can use the following error functions to capture information about the error that invoked the CATCH block:

* ERROR\_NUMBER()— Returns the error number
* ERROR\_MESSAGE()— Returns the complete text of the error message
* ERROR\_SEVERITY()— Returns the error severity
* ERROR\_STATE()— Returns the error state number
* ERROR\_LINE()— Returns the line number inside the procedure that caused the error
* ERROR\_PROCEDURE()— Returns the name of the stored procedure or trigger where the error occurred

Unlike @@ERROR, which is reset by each statement that is executed, the error information retrieved by the error functions remains constant anywhere within the scope of the CATCH block of a TRY...CATCH construct. Error functions can also be referenced from within a stored procedure invoked within a CATCH block. This allows you to modularize the error handling into a single stored procedure so you do not have to repeat the error-handling code in every CATCH block. Listing 1 shows an example of an error-handling procedure that you can use in your CATCH blocks.

**Listing 1. An Example of a Standard Error Handler Procedure**

|  |
| --- |
| use bigpubs2008  go  create proc dbo.error\_handler  as  begin  Declare @errnum int,  @severity int,  @errstate int,  @proc nvarchar(126),  @line int,  @message nvarchar(4000)  -- capture the error information that caused the CATCH block to be invoked  SELECT @errnum = ERROR\_NUMBER(),  @severity = ERROR\_SEVERITY(),  @errstate = ERROR\_STATE(),  @proc = ERROR\_PROCEDURE(),  @line = ERROR\_LINE(),  @message = ERROR\_MESSAGE() |

Listing 2 provides an example of the use of the TRY...CATCH construct in a T-SQL batch. Note that this CATCH block uses the dbo.error\_handler procedure defined in Listing 1.

**Listing 2. Using a TRY...CATCH Construct for Error Handling in a T-SQL Batch**

|  |
| --- |
| ----------------------------------------------------------  -----------1st Run this Statement  ----------------------------------------------------------  --NEXT RUN THIS STATEMENT WHICH WILL FAIL BECAUSE 'ali' is not an INT  use bigpubs2008  go  BEGIN TRY  INSERT INTO bigpubs2008.dbo.publishers  (pub\_id, pub\_name, city, state, country)  VALUES(9999, 'Sams Publishing', 'Indianapolis', 'IN', 'USA')  -- if no error occurs, we should see this print statement  print 'New Publisher added'  END TRY  BEGIN CATCH  -- invoke the error\_handler procedure  exec error\_handler  -- return a non-zero status code  END CATCH  ----------------------------------------------------------------------  ---NEXT RUN THIS STATEMENT WHICH WILL FAIL BECAUSE 'ali' is not an INT  ----------------------------------------------------------------------  use bigpubs2008  go  BEGIN TRY  INSERT INTO bigpubs2008.dbo.publishers  (pub\_id, pub\_name, city, state, country)  VALUES('ali', 'Sams Publishing', 'Indianapolis', 'IN', 'USA')  -- if no error occurs, we should see this print statement  print 'New Publisher added'  END TRY  BEGIN CATCH  -- invoke the error\_handler procedure  exec error\_handler  -- return a non-zero status code  END CATCH |

If you want to capture and handle any errors that may occur within a CATCH block, you can incorporate another TRY...CATCH block within the CATCH block.

Note

Some errors with severity 20 or higher that would cause SQL Server to close the user connection cannot be handled by the TRY...CATCH construct. However, severity level 20 or higher errors that do not result in the connection being closed can be captured and handled by the CATCH block. Any errors with a severity level of 10 or less are considered only warnings or informational messages and not really errors, and thus they are not handled by the TRY...CATCH construct. Also, any compile errors (such as syntax errors) or object name resolution errors that happen during deferred name resolution also do not invoke a CATCH block. These errors are returned to the application or batch that called the error-generating routine.