SQL Server: Concurrency Control Models, ACID Properties and Transaction Isolation Levels

Table of Contents

Introduction

Concurrency, by definition, means two or more events or circumstances happening at the same time.

In SQL Server terminology, this can be translated to multiple user transactions accessing or changing the shared data at the same time without disturbing each other. Concurrency always comes with some associated cost which we will see later in the article. This goal of this article is to be able to understand how SQL Server processes transactions and how concurrency is maintained, what options does SQL Server provide in terms of concurrency & consistency and the pros and cons of running the database under a specific model, known as Isolation level and finally, the differences between the Read-Committed Snapshot Isolation and Snapshot Isolation level. Knowledge of isolation levels in SQL Server is critical to understanding how SQL behaves under a particular level which would in-turn help in making key business decisions.

Before we dive into the types of isolation levels SQL Server offers, let us go through the properties that any transaction should guarantee. These properties are known by the acronym ACID where A stands for Atomicity, C for Consistency, I for isolation and D for Durability.

ACID Properties

The basic unit of work in SQL Server is a transaction, either implicit or explicit. An implicit transaction is the one without “BEGIN TRAN” prior to the SQL statements whereas, an explicit transaction is the one that starts with “BEGIN TRAN” and ends with either “COMMIT TRAN” or “ROLLBACK TRAN”. SQL Server ensures that transactions are performed as a single unit of work. Regardless implicit or explicit, SQL Server always ensures that every transaction adheres to the ACID properties.

To better understand what each of the ACID properties means, let us take an example of a bank where the following transaction takes place. We will reference this same example in each of the ACID properties explanation below.

Transaction Example: John Doe transfers $500 from his checking account to his savings account. His balance was $1000 in each of the two accounts prior to the transaction. The expected balance post-transfer is $500 in checking and $1500 in savings.

Now let’s look at the ACID properties:

Atomicity

Atomicity means that a transaction either fully completes or not at all. It cannot be left in the middle in an inconsistent state i.e. it either commits or rolls back completely. In the previous example, Atomicity guarantees that either the amount of $500 is transferred to John’s savings account or the transaction aborts and the balances remain unchanged in both accounts. Let’s say if the transaction partially completed i.e. it deducted $500 from his checking account but never credited his Savings account. The balances after the partial transaction would be $500 and $1000 in checking and savings respectively. This leads to undesirable and incorrect results. Thus, SQL Server guarantees Atomicity and either fully commits a transaction or rolls it back.

Consistency

Consistency ensures that a transaction will always bring the database from one consistent state to another. It ensures that data is always logically correct. It prevents the transaction from an incorrect logical state. In the preceding example, a new $500 cannot be added or removed out of nowhere, it has to be first removed from his checking account and reflect in the savings account to be called a consistent transaction. Consistency also ensures that none of the database rules and constraints is violated by the transaction.

Isolation

Isolation means that a transaction is isolated from other concurrent processes. It prevents concurrent processes from seeing the updates of an incomplete transaction. As in our example, the balances cannot be viewed by other concurrent transactions until the transaction completes. Otherwise, if another process sees the checking account’s balance after $500 is deducted but before the transaction completed, then that process is reading the balance that is not guaranteed to be correct i.e. what if the transaction aborts and the balance in checking go back to $1000? Since the other process read “dirty” data, it would lead to inconsistency problems. SQL Server ensures isolation among transactions by locking data so other transactions are blocked from seeing an update of an incomplete transaction. If run under a specific isolation level (called Snapshot isolation, explained later), SQL Server writes row versions to allow concurrent transactions to see the committed data at the same time making sure that consistency is not compromised.

Durability

This property ensures that, as and when a transaction commits, it is made persistent so that even in a case of power failure or system crash after it is committed, the transaction can be replayed to bring it back to the consistent state it was prior to the failure. Durability also ensures that if the system crash occurs while the transaction is in the middle, all the changes made by the partially completed transaction are undone or rolled back. Durability in SQL Server is taken care of by a mechanism known as Write-Ahead logging wherein, all log records are written to the log file on disk prior to the corresponding data pages are written to the MDF file on disk. The internals and the behavior of WAL or T-log file are outside the scope of this article.

Tip: Read this article to understand the T-log internals and how Write-Ahead logging is guaranteed in SQL Server.

Concurrency

Back to concurrency, there are two approaches in any database to managing the concurrent transactions.

1. Pessimistic
2. Optimistic

Before SQL Server 2005, SQL Server supported only pessimistic concurrency model. SQL Server 2005 introduced the concept of optimistic concurrency and showed how to set it at the transaction level.

Pessimistic concurrency model

This is the default concurrency model in SQL Server. As the name suggests, this concurrency model assumes that data modifications have been made and any read operation is going to be affected by that modification. Fearing a conflict, it acquires locks on the data being read so that any transactions intending to write are blocked until the read is complete and conversely, it acquires locks on the data being modified/updated so that no other transaction can read it. This is the default model where readers block writers and writer's block readers.

Optimistic concurrency model

This model assumes the opposite of pessimistic model i.e. it assumes that the system has few conflicting modifications and that any transaction is unlikely to change data that is being modified by another transaction. This model uses the concept of row versioning wherein versions of rows are saved into tempdb (in version store) so that any read transaction sees the version of data as it existed at the time the read initiated. Changes can be made to the data being read and are not affected by the read operation similarly, reads can be performed on the data being modified and are not affected by the modification transaction. In short, readers don’t block writers and writers don’t block readers.

Note that writers will always block writers no matter what the model or isolation level is, to avoid conflicts.

Transaction behaviors

Now that we have looked at the ACID properties of a transaction and the two concurrency models in SQL Server, let us get familiar with a few behaviors a transaction might adopt before diving into the different isolation levels. Knowing these terms or rather behaviors is essential to understanding the isolation levels. Some isolation levels in SQL Server allow some of the following behaviors and some do not allow any of them.

Remember, these are undesirable behaviors and hence, lesser these behaviors, efficient is the Isolation level.

Dirty Reads

Put simply, when a transaction reads uncommitted data, then such reads are called dirty reads. By default, SQL Server doesn’t allow dirty reads by using locks i.e. the transaction that is making changes locks the data being changed so no read can be performed against it. It is to be noted that the reading transaction can still choose to read uncommitted or dirty data (for example, by using NOLOCK hint) i.e. the transaction making changes has no control over whether to allow a read transaction to proceed or not before the changes are committed.

Back to our bank example, if dirty reads were allowed, as soon as the balance in checking account updated to $500 from the original $1000, another transaction B, reads that $500 (dirty read) from the checking account’s balance and now, if the update transaction is rolled back, the transaction B would have a dirty read and may lead to bigger problems.

Non-repeatable reads

Non-repeatable reads are nothing but read that give different values to the process within the same transaction on subsequent read attempts. In other words, if two reads are performed one after the other within the same transaction and they both give different values, then such reads are known as non-repeatable reads. If any process changes the data after the first read and before the second read could start, the second read will read a different value and this gives rise to inconsistencies.

Phantom reads

A phantom read occurs when subsequent reads within the same transaction return new rows. The new rows are considered as phantom rows. This happens when another transaction inserts data in between the reads and the second read reads ghost records that were inserted after the first read. It happens with queries with predicates such as “WHERE” clause.

Duplicated reads

This occurs when a query returns the same row multiple times. This happens in Read-Uncommitted transaction isolation level (may also occur in Read-Committed level) which we will see later in this article. What happens in this type of read is, when a full table is scanned, SQL may choose to do an allocation order scan and if any concurrent transaction performs updates, it may move the already read rows to a higher location in the table and since the read operation is following the allocation order scan, it will read the same row twice.

Skipped rows

Similar to duplicated reads, if a row on a high page number is updated and moved to a lower page (by an update operation) that has already been read by a read operation, that row is skipped because the read operation followed the allocation order scan (using page number) instead of logical pointers and won’t go back to the lower page number hence skipping that row.

Transaction Isolation levels

A transaction behaves as per the isolation level it runs under. The command SET TRANSACTION ISOLATION LEVEL <IsolationlevelName> sets the desired isolation level for the transaction being run. An isolation level could be optimistic or pessimistic. SQL Server supports five isolation levels, of which three are pessimistic, one is optimistic and one supports either.

1. Read Uncommitted

As the name implies, this isolation level reads uncommitted data and may end up having inconsistencies due to dirty reads. SQL Server allows a read operation to not take any locks and it won’t be blocked. Again, under this isolation level, a process is allowed to read uncommitted data i.e. data that is not guaranteed to be committed. As mentioned above, this isolation level also leads to “Duplicated reads” and “Skipped rows”.

1. When doing a table scan, if SQL Server chooses to do allocation order scan i.e. in the order of page number instead of page pointers scan (logical order scan) and if there is any concurrent transaction that moved “already read” rows to a new location (higher), the read operation in progress will read this row twice- the first time the row was read and second time when the row was moved to a higher page number by a concurrent operation. Thus causing duplicated reads.
2. During the same table scan, when a concurrent transaction moves an unread row from a higher page number that has not been covered yet by the read transaction to a lower page number that has already been read, the row will never be read and will be skipped. Thus causing skipped rows.

Seeing the undesirable behaviors of this isolation level, one may not see this to be of good use but the main advantage of this isolation level is that the read operation is not blocked and don’t acquire any locks. This comes at the cost of compromising consistency.

Let’s see which of the above discussed (undesirable) behaviors are allowed under the Read Uncommitted transaction isolation level.

1. Dirty Reads = Yes
2. Non-repeatable Reads = Yes
3. Phantom Reads = Yes
4. Duplicated Reads = Yes
5. Skipped Rows = Yes

Now, let's see this isolation level's behavior in action. The following actions occur in the sequence order.

1. A database named "Sample" is created and a table named "Iso_level" is created within it. The table is then populated with 1 dummy record.
2. An update transaction is started in a new session in an attempt to change the value "John" to "James" for row ID=1.
3. A Read transaction is run in another session under "Read Uncommitted" isolation level to read the row being changed (dirty) by the concurrent update transaction.

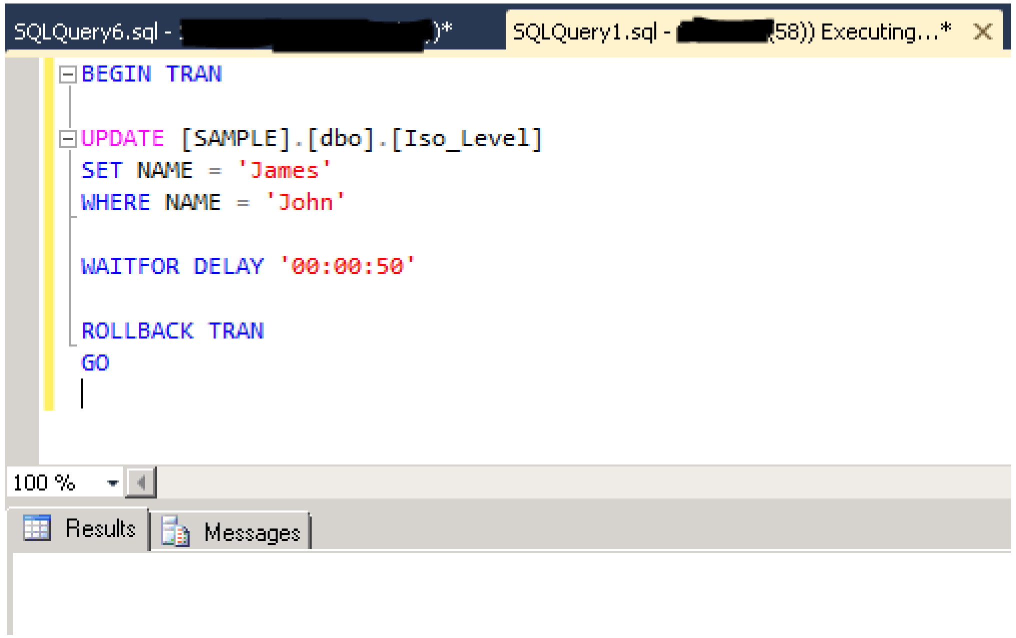
```
USE MASTER
GO
CREATE DATABASE [SAMPLE]
GO

USE [SAMPLE]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
SET ANSI_PADDING ON
GO
```

```
CREATE TABLE [dbo].[Iso_Level](
    [ID] [int] NULL,
    [Name] [varchar](50) NULL
) ON [PRIMARY]

GO
SET ANSI_PADDING OFF
GO
INSERT [dbo].[Iso_Level] ([ID], [Name]) VALUES (1, N'John')
GO
```

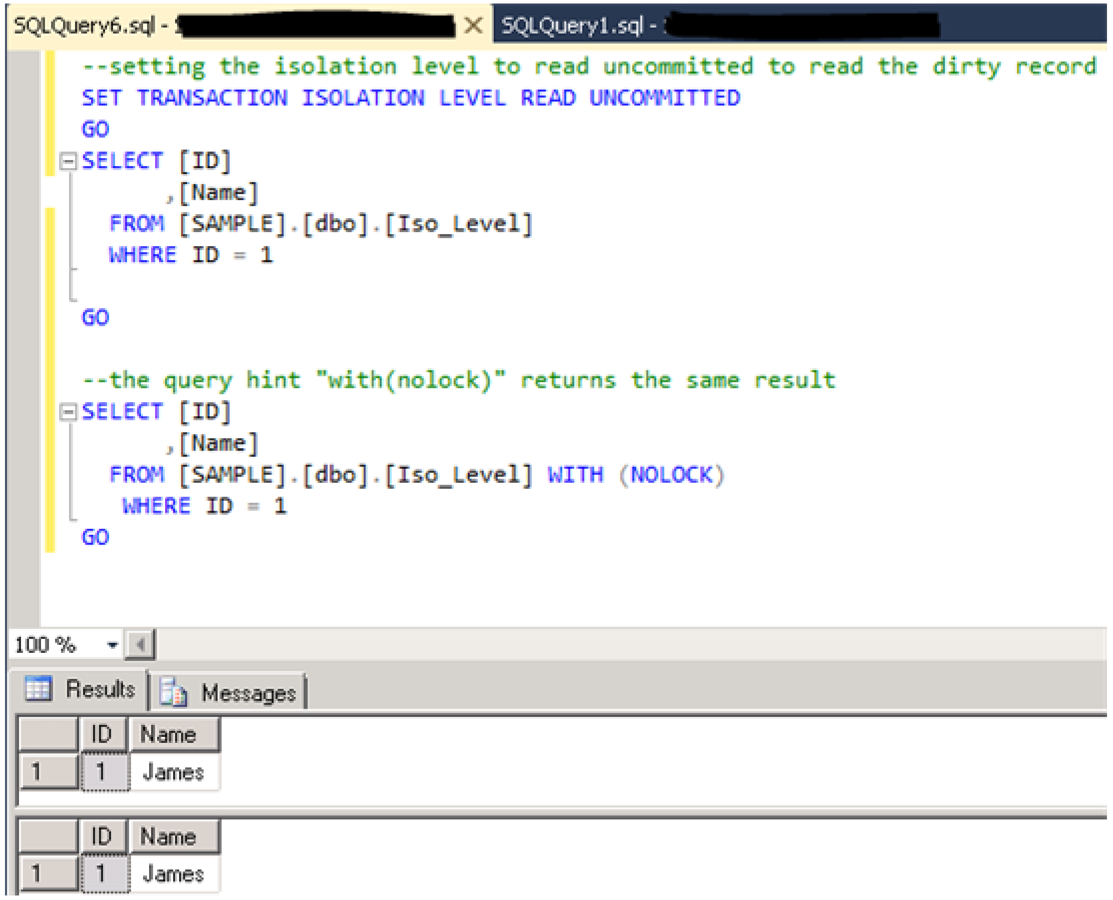
```
BEGIN TRAN

UPDATE [SAMPLE].[dbo].[Iso_Level]
SET NAME = 'James'
WHERE NAME = 'John'
AND ID =1

WAITFOR DELAY '00:00:50'

ROLLBACK TRAN
GO
```

While the update was being executed and waiting to lapse 50 seconds, the following "read" transaction is executed in another session.

```
--setting the isolation level to read uncommitted to read the dirty record
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
GO
SELECT [ID]
      ,[Name]
  FROM [SAMPLE].[dbo].[Iso_Level]
  WHERE ID = 1

GO

--the query hint "with(nolock)" returns the same result
SELECT [ID]
      ,[Name]
  FROM [SAMPLE].[dbo].[Iso_Level] WITH (NOLOCK)
   WHERE ID = 1
GO
```


SQLQuery6.sql - [redacted])* SQLQuery1.sql - [redacted]58)) Executing...*

```
BEGIN TRAN

UPDATE [SAMPLE].[dbo].[Iso_Level]
SET NAME = 'James'
WHERE NAME = 'John'

WAITFOR DELAY '00:00:50'

ROLLBACK TRAN
GO
```



```
--setting the isolation level to read uncommitted to read the dirty record
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
GO
SELECT [ID]
      ,[Name]
  FROM [SAMPLE].[dbo].[Iso_Level]
  WHERE ID = 1

GO

--the query hint "with(nolock)" returns the same result
SELECT [ID]
      ,[Name]
  FROM [SAMPLE].[dbo].[Iso_Level] WITH (NOLOCK)
   WHERE ID = 1
GO
```

	ID	Name
1	1	James

	ID	Name
1	1	James

As we can see, the read transaction read the dirty record. The update was eventually rolled back but because the read transaction was executed under "Read Uncommitted" isolation level, it read the uncommitted value.

2. Read Committed

Read committed is the default isolation level. It comes in two flavors i.e. it can either be optimistic or pessimistic, contingent upon a database option “READ_COMMITTED_SNAPSHOT”. If this option is set to ON, this isolation level adopts optimistic behavior and if OFF, it becomes pessimistic. The default setting is OFF, hence making the default behavior of this isolation level as pessimistic.

a) Read Committed (Pessimistic)

The pessimistic behavior comes from the fact that if a transaction locks down the rows exclusively (exclusive locks) for an update operation, a read transaction against the same rows is blocked until the first transaction releases exclusive locks. Similarly, any read operation must acquire shared locks on the data being read so that the rows are not available to other transaction intending to update. Note that a shared lock still allows other concurrent transaction to read data but any update transaction would be blocked by the read transaction that acquired shared locks.

One key thing to note here is that a shared lock, by default, is released as soon as the data is processed i.e. the lock does not have to wait till the completion of the whole transaction (or even the statement) to be released. As soon as the row is processed, the row lock can be released but the transaction or statement itself might still be processing other rows.

Recall the definition of non-repeatable reads and relate it to this default shared lock behavior. That is, if a transaction consists of two read statements then there is a chance of non-repeatable reads under this isolation level because of the default shared locking behavior i.e. the rows that are processed by the first read could possibly be changed by another concurrent transaction before the second read and the second read will see a different value.

Let’s see which of the behaviors are allowed under the Read Committed (Pessimistic, locking) transaction isolation level.

1. Dirty Reads = No
2. Non-repeatable Reads = Yes
3. Phantom Reads = Yes
4. Duplicated Reads = Yes
5. Skipped Rows = Yes

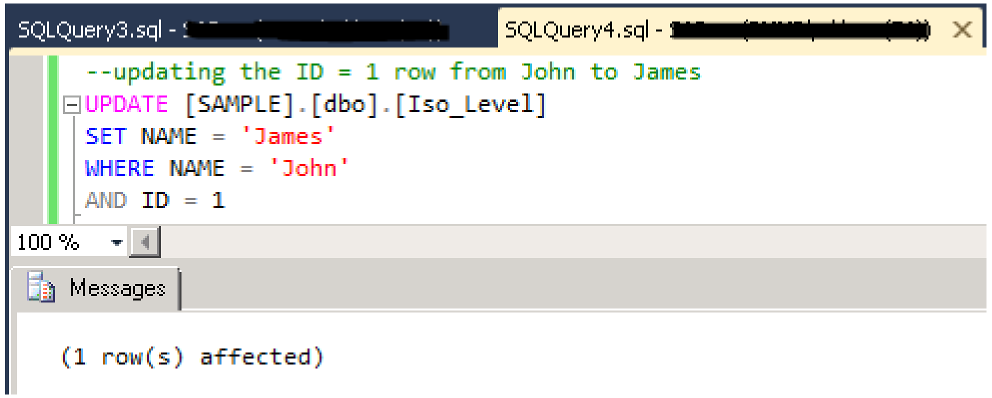
The following example shows the behavior under this isolation level. The following series of actions occur in sequence.

1. A transaction with two read statements is started in a new session. The transaction waits for 30 seconds before initiating the second read within the same transaction.
2. While the read transaction waits, a concurrent "update" statement is executed in another session. This is to update the row that has already been read by the first "select" statement in the on-going read transaction. The update succeeds.
3. When the read transaction finally lapses 30 seconds and re-reads the same row, it reads a different value because it has been updated by a concurrent update transaction. Hence causing non-repeatable reads.

```
--This shows the read committed (locking) behavior
--This is the default isolation level in SQL Server
--No setting needs to be changed
--two Non-repeatable READs within the same transaction
BEGIN TRAN

SELECT [ID]
      ,[Name]
  FROM [SAMPLE].[dbo].[Iso_Level]
  WHERE ID = 1
--at this point, there is a very quick shared lock on row ID 1 and releases as soon as row is read
```

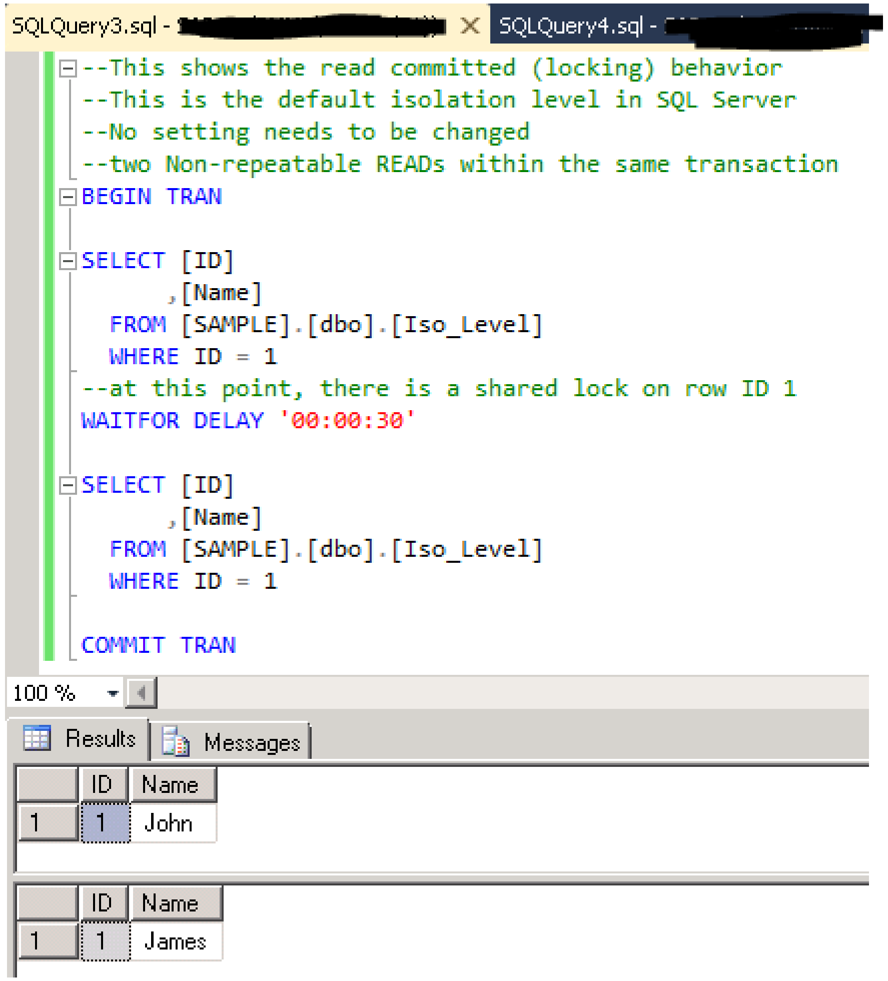
```
WAITFOR DELAY '00:00:30'

SELECT [ID]
      ,[Name]
  FROM [SAMPLE].[dbo].[Iso_Level]
  WHERE ID = 1

COMMIT TRAN
```

```
--updating the ID = 1 row from John to James
UPDATE [SAMPLE].[dbo].[Iso_Level]
SET NAME = 'James'
WHERE NAME = 'John'
AND ID = 1
```


SQLQuery3.sql - [redacted] SQLQuery4.sql - [redacted]

```
--updating the ID = 1 row from John to James
UPDATE [SAMPLE].[dbo].[Iso_Level]
SET NAME = 'James'
WHERE NAME = 'John'
AND ID = 1
```

```
(1 row(s) affected)
```



```
--This shows the read committed (locking) behavior
--This is the default isolation level in SQL Server
--No setting needs to be changed
--two Non-repeatable READs within the same transaction
BEGIN TRAN

SELECT [ID]
      ,[Name]
  FROM [SAMPLE].[dbo].[Iso_Level]
  WHERE ID = 1
--at this point, there is a shared lock on row ID 1
WAITFOR DELAY '00:00:30'

SELECT [ID]
      ,[Name]
  FROM [SAMPLE].[dbo].[Iso_Level]
  WHERE ID = 1

COMMIT TRAN
```

```
   ID  Name
1  1   John

   ID  Name
1  1   James
```

b) Read Committed (Optimistic)

This isolation level is known as Read-Committed Snapshot due to the database setting “READ_COMMITTED_SNAPSHOT” being turned ON. This one is similar to the Read Committed (locking) in terms of reading only committed data, but the main difference is that this one doesn’t cause blocking. When a row is changed, its previous version (i.e. the committed value) is generated by SQL Server and stored in tempdb so that any other read operation that depends on the old version of the row can see it, thus maintaining transactional consistency.

Exclusive locks are still acquired by update transactions but because we now have a previous committed version of the row, a concurrent read transaction can proceed without being blocked.

The main advantage this isolation level has over the default one (with locking) is, readers don’t block writers and writers don’t block readers.

Let’s see which behaviors are allowed under the Read Committed Snapshot transaction isolation level.

1. Dirty Reads = No
2. Non-repeatable Reads = Yes
3. Phantom Reads = Yes
4. Duplicated Reads = No
5. Skipped Rows = No

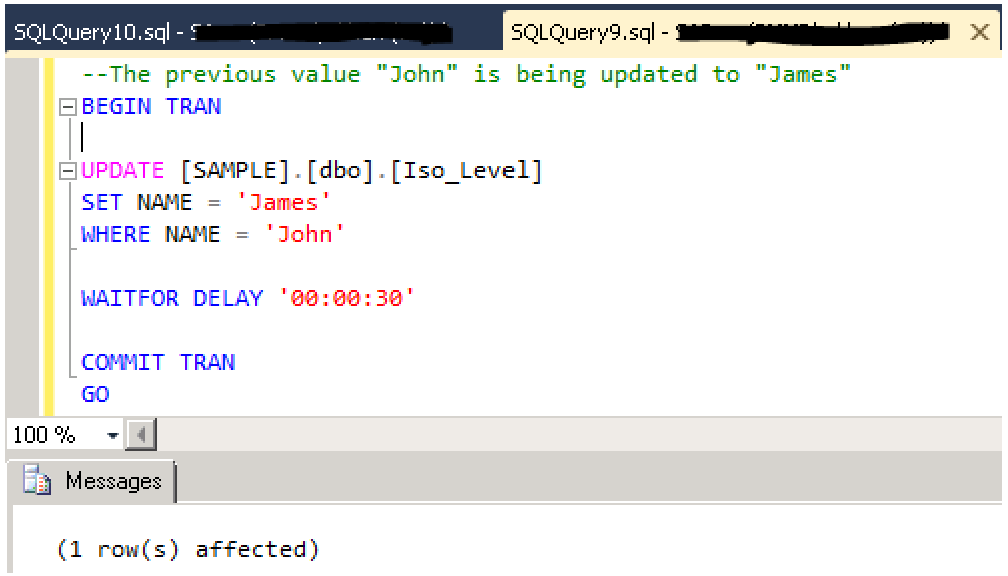
Each query will read the database at a consistent point-in-time using row versions. If a query encounters a row that has been updated since the query began, the previous version of the row is read from the Version Store. This eliminates Duplicated Reads and Skipped Rows. You can see different versions of a row over the course of a Transaction. The Snapshot isolation level provides a transaction-level point-in-time view of the database.

The below example shows that the behavior under this isolation level is pretty much same as in the case of Read Committed (locking) with an exception that the update transaction is not blocked by the concurrent read transaction i.e readers and writers do not block each other.

```
--This shows the read committed (Snapshot) behavior
--set READ_COMMITTED_SNAPSHOT ON on the database before
--Two Non-repeatable READs within the same transaction

USE [master]
GO
ALTER DATABASE [SAMPLE] SET READ_COMMITTED_SNAPSHOT ON WITH NO_WAIT
GO

BEGIN TRAN

SELECT [ID]
      ,[Name]
  FROM [SAMPLE].[dbo].[Iso_Level]
  WHERE ID = 1

WAITFOR DELAY '00:00:35'

SELECT [ID]
      ,[Name]
  FROM [SAMPLE].[dbo].[Iso_Level]
  WHERE ID = 1
```

```
COMMIT TRAN
```

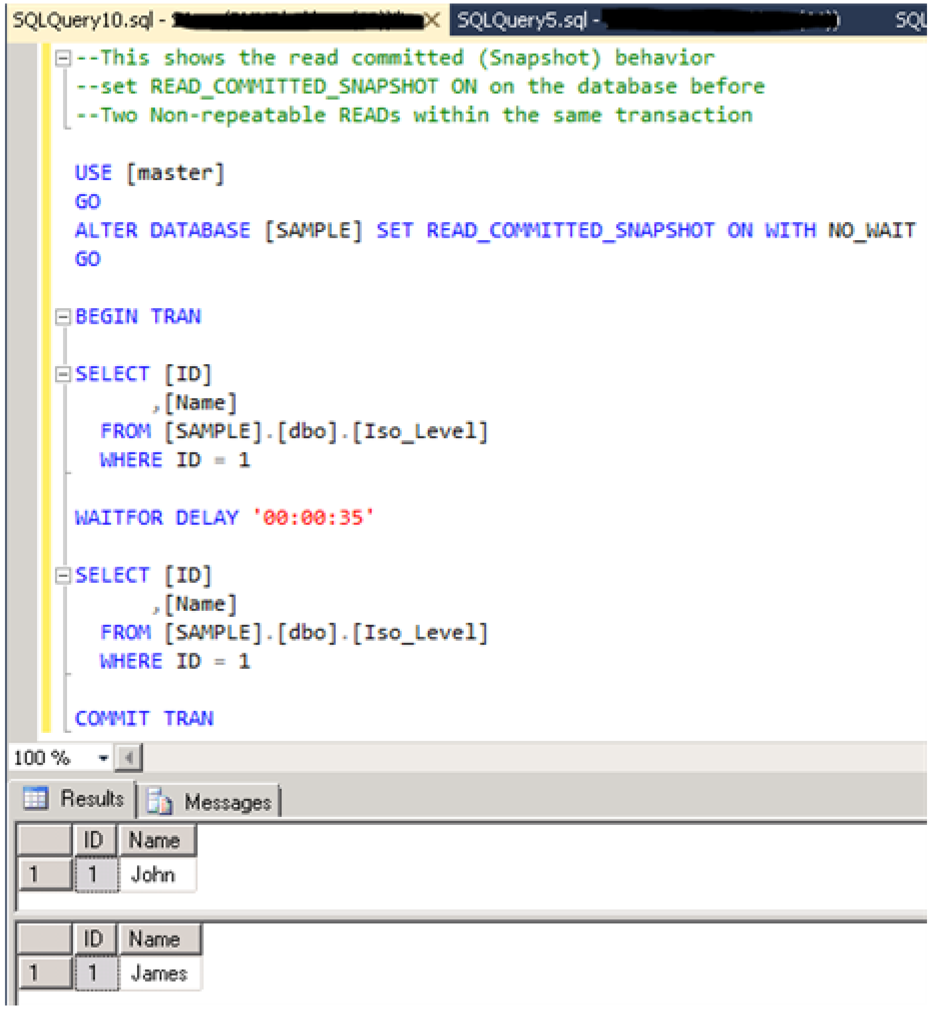
```
--The previous value "John" is being updated to "James"
BEGIN TRAN

UPDATE [SAMPLE].[dbo].[Iso_Level]
SET NAME = 'James'
WHERE NAME = 'John'

WAITFOR DELAY '00:00:30'

COMMIT TRAN
GO
```



SQLQuery10.sql - [redacted] SQLQuery9.sql - [redacted]

```
--The previous value "John" is being updated to "James"
BEGIN TRAN

UPDATE [SAMPLE].[dbo].[Iso_Level]
SET NAME = 'James'
WHERE NAME = 'John'

WAITFOR DELAY '00:00:30'

COMMIT TRAN
GO
```

```
(1 row(s) affected)
```



SQLQuery10.sql - [redacted] SQLQuery5.sql - [redacted]

```
--This shows the read committed (Snapshot) behavior
--set READ_COMMITTED_SNAPSHOT ON on the database before
--Two Non-repeatable READs within the same transaction

USE [master]
GO
ALTER DATABASE [SAMPLE] SET READ_COMMITTED_SNAPSHOT ON WITH NO_WAIT
GO

BEGIN TRAN

SELECT [ID]
      ,[Name]
  FROM [SAMPLE].[dbo].[Iso_Level]
  WHERE ID = 1

WAITFOR DELAY '00:00:35'

SELECT [ID]
      ,[Name]
  FROM [SAMPLE].[dbo].[Iso_Level]
  WHERE ID = 1

COMMIT TRAN
```

100 %

Results | Messages

```
   ID  Name
1  1   John

   ID  Name
1  1   James
```

3. Repeatable Read

As the name implies, this isolation level assures that any subsequent reads within a transaction always give same data and the data doesn’t change. This is pessimistic in behavior. No changes can be made by other transaction to the data while the read transaction is in progress, hence any number of reads would return same values. The main drawback, however, is that phantom reads can still happen.

The idea behind Repeated Read is that shared locks are held for the entire duration of the transaction as opposed to Read Committed isolation where locks are released as soon as the rows are processed even though the transaction itself has not finished. This behavior of acquiring shared locks throughout the transaction guarantees repeatable reads.

Note: The above behavior is for shared locks only as Exclusive locks are always acquired and held until the transaction ends irrespective of the isolation level the transaction runs under.

This isolation level prevents other users from modifying data that the read operation accessed as long as the read transaction is active. This comes at the cost of reducing the concurrency.

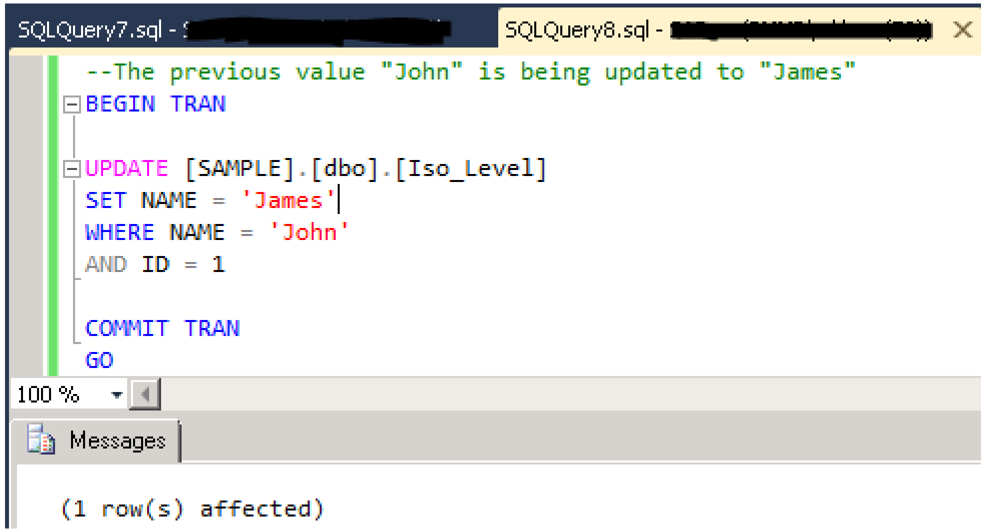
Let's see which behaviors are allowed under the Repeatable Read transaction isolation level.

1. Dirty Reads = No
2. Non-repeatable Reads = No
3. Phantom Reads = Yes
4. Duplicated Reads = No
5. Skipped Rows = Yes

The example below shows the behavior under "Repeatable Read" isolation level. Notice the two read statements within the read transaction return the same value (the committed value). This comes at the cost of holding the shared lock and blocking the concurrent update transaction until the read finishes.

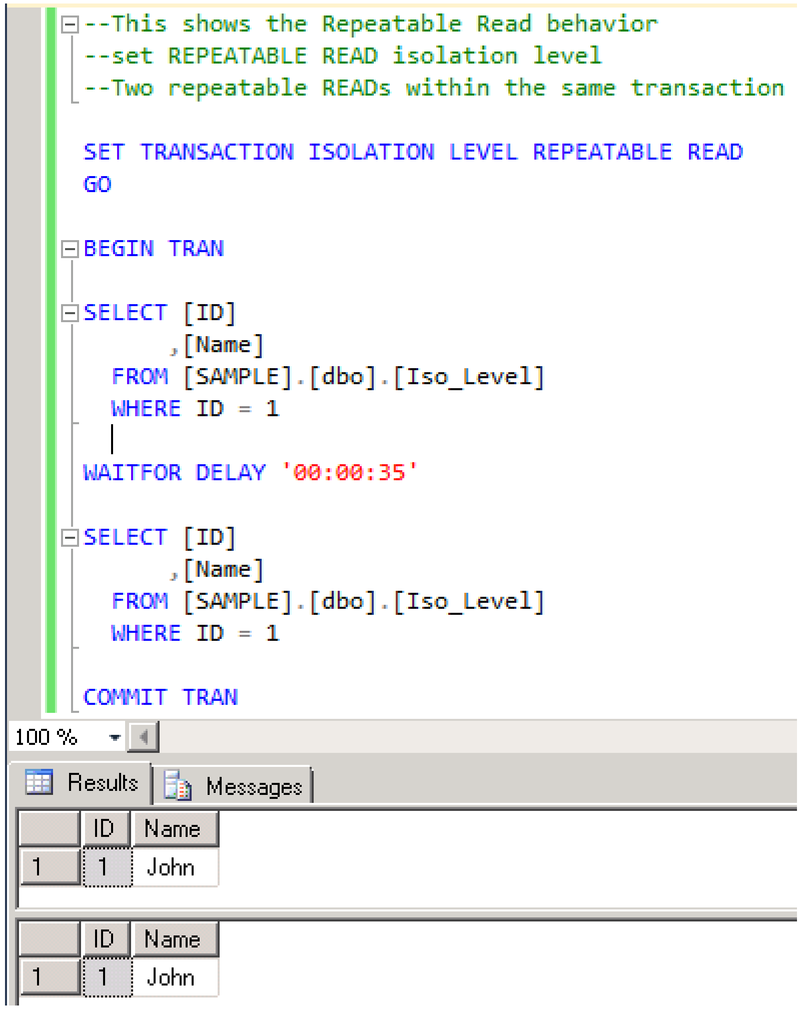
```sql
--This shows the Repeatable Read behavior
--set REPEATABLE READ isolation level
--Two repeatable READs within the same transaction

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
GO

BEGIN TRAN

SELECT [ID]
      ,[Name]
  FROM [SAMPLE].[dbo].[Iso_Level]
  WHERE ID = 1

WAITFOR DELAY '00:00:35'

SELECT [ID]
      ,[Name]
  FROM [SAMPLE].[dbo].[Iso_Level]
  WHERE ID = 1

COMMIT TRAN
```

```
--The previous value "John" is being updated to "James"
BEGIN TRAN

UPDATE [SAMPLE].[dbo].[Iso_Level]
SET NAME = 'James'
WHERE NAME = 'John'
AND ID = 1

COMMIT TRAN
GO
```



```
--The previous value "John" is being updated to "James"
BEGIN TRAN

UPDATE [SAMPLE].[dbo].[Iso_Level]
SET NAME = 'James'
WHERE NAME = 'John'
AND ID = 1

COMMIT TRAN
GO
```

```
(1 row(s) affected)
```



```
--This shows the Repeatable Read behavior
--set REPEATABLE READ isolation level
--Two repeatable READs within the same transaction

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
GO

BEGIN TRAN

SELECT [ID]
      ,[Name]
  FROM [SAMPLE].[dbo].[Iso_Level]
  WHERE ID = 1

WAITFOR DELAY '00:00:35'

SELECT [ID]
      ,[Name]
  FROM [SAMPLE].[dbo].[Iso_Level]
  WHERE ID = 1

COMMIT TRAN
```

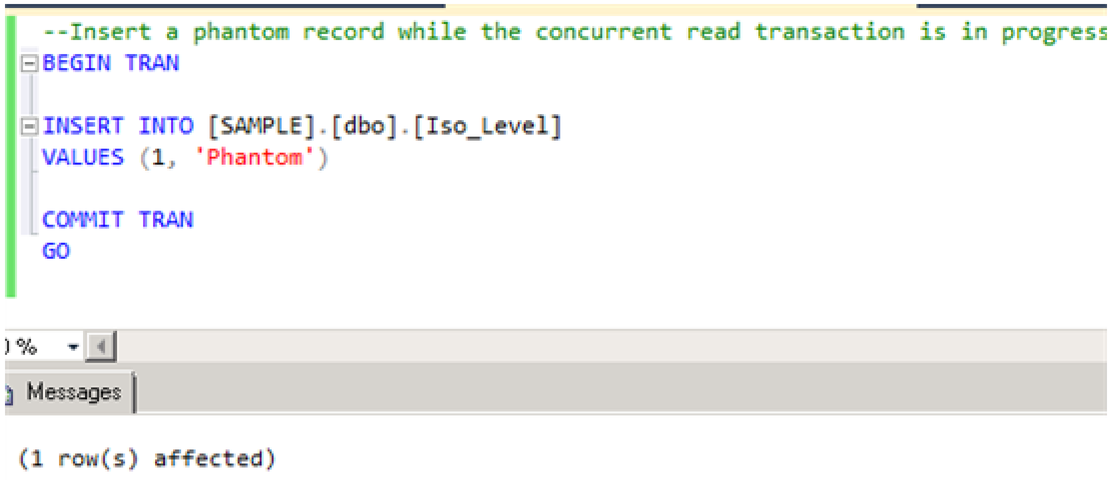
```
   ID  Name
1  1   John

   ID  Name
1  1   John
```

Now, the example below proves that "Phantom rows" are possible under Repeatable Read isolation level. A phantom record is successfully inserted while the first "select" in the read transaction finished but before the second read started and hence, the second read returns the phantom record.

```
--This shows how phantom record shows up in repeatable read
--set REPEATABLE READ isolation level
--Two READs within the same transaction, one showing a phantom

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
GO
```

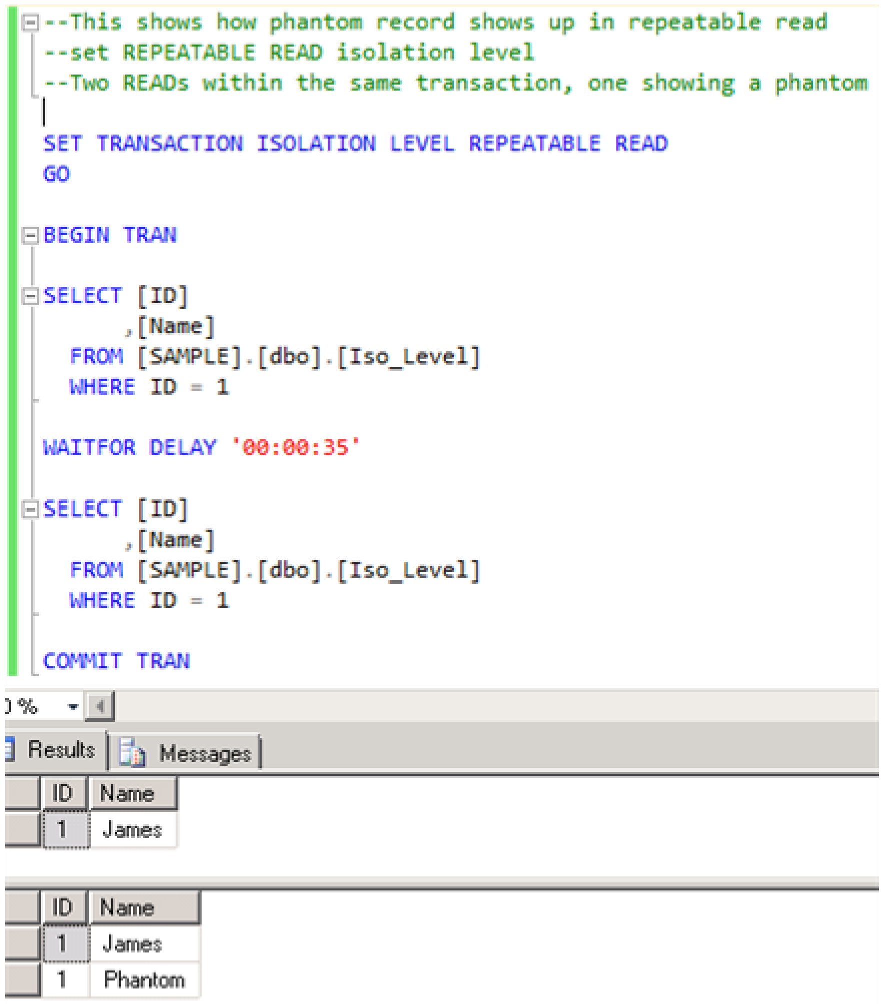
```
BEGIN TRAN

SELECT [ID]
      ,[Name]
  FROM [SAMPLE].[dbo].[Iso_Level]
  WHERE ID = 1

WAITFOR DELAY '00:00:35'

SELECT [ID]
      ,[Name]
  FROM [SAMPLE].[dbo].[Iso_Level]
  WHERE ID = 1

COMMIT TRAN
```

```
--Insert a phantom record while the concurrent read transaction is in progress
BEGIN TRAN

INSERT INTO [SAMPLE].[dbo].[Iso_Level]
VALUES (1, 'Phantom')

COMMIT TRAN
GO
```



```
--Insert a phantom record while the concurrent read transaction is in progress
BEGIN TRAN

INSERT INTO [SAMPLE].[dbo].[Iso_Level]
VALUES (1, 'Phantom')

COMMIT TRAN
GO
```

```
(1 row(s) affected)
```



```
--This shows how phantom record shows up in repeatable read
--set REPEATABLE READ isolation level
--Two READs within the same transaction, one showing a phantom

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
GO

BEGIN TRAN

SELECT [ID]
      ,[Name]
  FROM [SAMPLE].[dbo].[Iso_Level]
  WHERE ID = 1

WAITFOR DELAY '00:00:35'

SELECT [ID]
      ,[Name]
  FROM [SAMPLE].[dbo].[Iso_Level]
  WHERE ID = 1

COMMIT TRAN
```

ID	Name
1	James

ID	Name
1	James
1	Phantom

4. Serializable

This pessimistic isolation level is similar to Repeatable Read but doesn't allow phantom reads. In fact, this isolation level doesn't allow any of the five undesirable behaviors discussed above. It is the only pessimistic level that doesn't incur any data inconsistency problems but has concurrency issues due to the fact that it uses locking and blocking.

Under Serializable isolation level, there are no phantoms records i.e. when a query is re-run within the same transaction, same rows are returned and there are new rows added. It uses the similar locking behavior as Repeatable read i.e. shared locks are held for the life of the transaction. An added property in Serializable isolation level is that it locks the data that doesn't exist.

As an example, if a query to get the employees whose salary is between $5000 and $7000 returns no rows, Serializable isolation level locks that range of rows (even though the query returns nothing) i.e. rows with salaries between $5000 to $7000. This ensures that if the same query is run again, it continues to return no rows so there are no phantom rows. SQL does this by using a special lock called "key-range lock" These locks require the table to have an index on the column referenced in the where clause (salary in our example) otherwise, it acquires a table lock.

The name Serializable is for a reason, if there are multiple Serializable transactions running simultaneously, it is just as running them in serial order. The order is completely random. If there are four update transactions working on the same data range, whichever gets the lock first is the first to execute and so on. Hence, the results are all same whether they are run in parallel or in the serial order in which the lock was acquired.

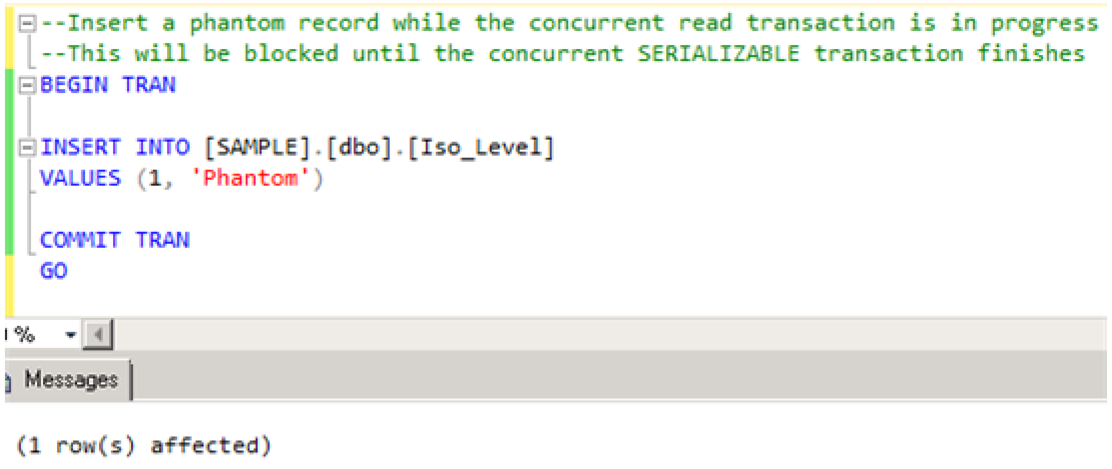
Let's see which behaviors are allowed under the Serializable transaction isolation level.

1. Dirty Reads = No
2. Non-repeatable Reads = No
3. Phantom Reads = No
4. Duplicated Reads = No
5. Skipped Rows = No

The example below shows how "Serializable" isolation level prevents phantom records. Similar to previous examples, the same actions occur in sequence i.e. the read transaction is started and a concurrent "insert" transaction is run from a new session in an attempt to insert a phantom record. The attempt fails i.e. the insert transaction is blocked until the concurrent read transaction is finished.

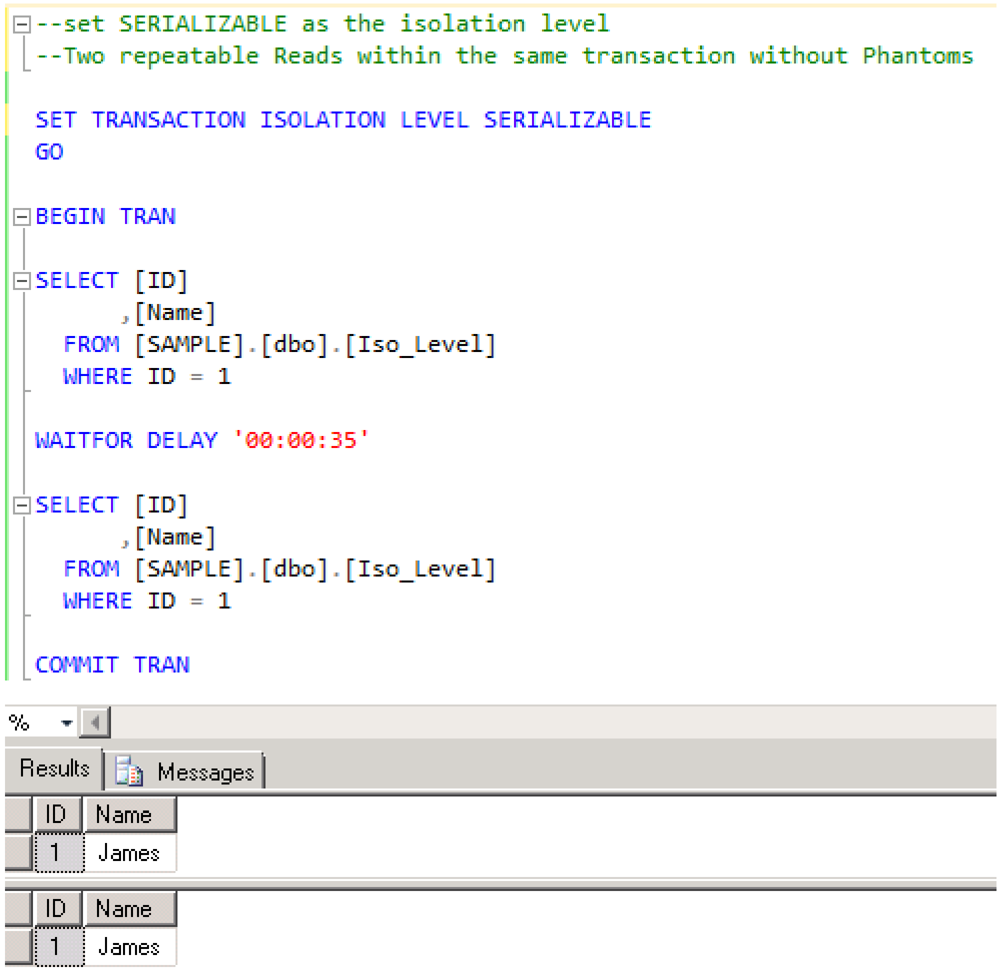
```
--set SERIALIZABLE as the isolation level
--Two repeatable Reads within the same transaction without Phantoms

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
GO

BEGIN TRAN

SELECT [ID]
      ,[Name]
  FROM [SAMPLE].[dbo].[Iso_Level]
  WHERE ID = 1

WAITFOR DELAY '00:00:35'

SELECT [ID]
      ,[Name]
  FROM [SAMPLE].[dbo].[Iso_Level]
  WHERE ID = 1

COMMIT TRAN
```

```
--Insert a phantom record while the concurrent read transaction is in progress
--This will be blocked until the concurrent SERIALIZABLE transaction finishes
BEGIN TRAN

INSERT INTO [SAMPLE].[dbo].[Iso_Level]
VALUES (1, 'Phantom')

COMMIT TRAN
GO
```



```
--Insert a phantom record while the concurrent read transaction is in progress
--This will be blocked until the concurrent SERIALIZABLE transaction finishes
BEGIN TRAN

INSERT INTO [SAMPLE].[dbo].[Iso_Level]
VALUES (1, 'Phantom')

COMMIT TRAN
GO
```

```
(1 row(s) affected)
```



```
--set SERIALIZABLE as the isolation level
--Two repeatable Reads within the same transaction without Phantoms

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
GO

BEGIN TRAN

SELECT [ID]
      ,[Name]
  FROM [SAMPLE].[dbo].[Iso_Level]
  WHERE ID = 1

WAITFOR DELAY '00:00:35'

SELECT [ID]
      ,[Name]
  FROM [SAMPLE].[dbo].[Iso_Level]
  WHERE ID = 1

COMMIT TRAN
```

Results | Messages

ID	Name
1	James

ID	Name
1	James

5. Snapshot Isolation

This optimistic concurrency isolation level uses a method called row versioning (explained shortly) to allow concurrent transactions to access the data i.e. readers don't block writers and writers don't block readers. A transaction can still read the committed version (earlier row version) of the data that is currently being exclusively locked by an update transaction, similarly, a transaction can acquire an exclusive lock and perform update on the data and won't block a read transaction (the read operation sees the previous committed version of the data).

The version store is the place in tempdb where all row versions are stored i.e. the older versions of rows. These versions are kept so long as any transaction might need to access the older versions for consistency. Concurrency is guaranteed in this isolation level because multiple sessions could work together with their respective versions without interfering with each other. The version of the row they see is consistent as of the time the transaction fired. Since tempdb is heavily used to store multiple versions, it must be managed efficiently or it runs the risk of having contention.

Let’s see which behaviors are allowed under the Snapshot transaction isolation level.

1. Dirty Reads = No
2. Non-repeatable Reads = No
3. Phantom Reads = No
4. Duplicated Reads = No
5. Skipped Rows = No

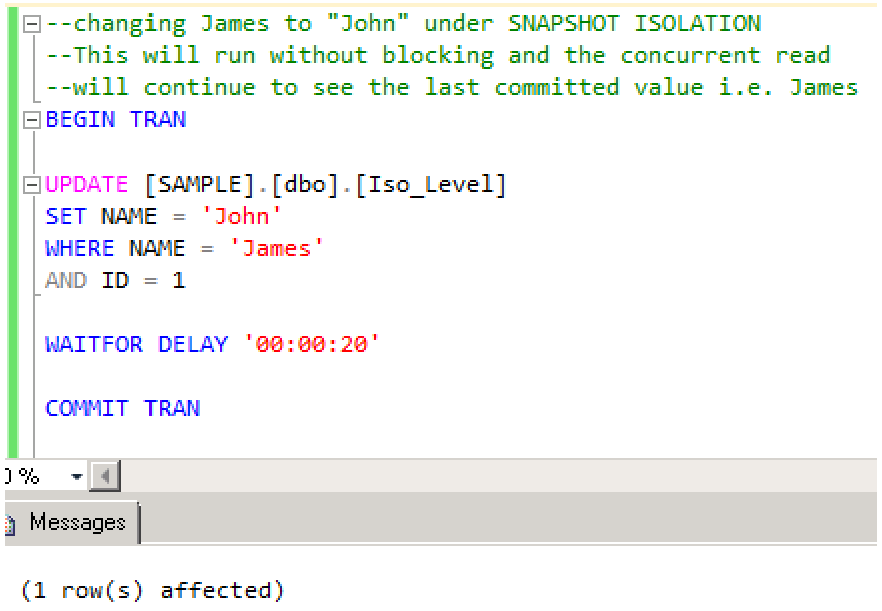
The example below shows the "Snapshot" isolation behavior.

1. There are two read transactions in the first session. The first transaction in-turn has two select statements returning the same value hence there are no non-repeatable reads.
2. There is a concurrent update transaction that runs in another session. This is not blocked by the earlier read transaction. The read operation sees the last committed value and continues fine and so does the update transaction. Hence there is no blocking.
3. After the update transaction finishes, the second read transaction in the first session starts and reads the updated value (changed as a result of the update in the middle). Hence making the new value as the latest/current version.

```
--set SNAPSHOT as the isolation level
--Two repeatable Reads within the same transaction without Phantoms
--Readers don't block writers and writers don't block readers

ALTER DATABASE [SAMPLE] SET ALLOW_SNAPSHOT_ISOLATION ON
GO
SET TRANSACTION ISOLATION LEVEL SNAPSHOT
GO

--This FIRST TRANSACTION will return last committed James as the
--concurrent update transaction has not committed yet
--notice there are no non-repeatable reads
BEGIN TRAN

SELECT [ID]
      ,[Name]
  FROM [SAMPLE].[dbo].[Iso_Level]
  WHERE ID = 1

WAITFOR DELAY '00:00:35'

SELECT [ID]
```

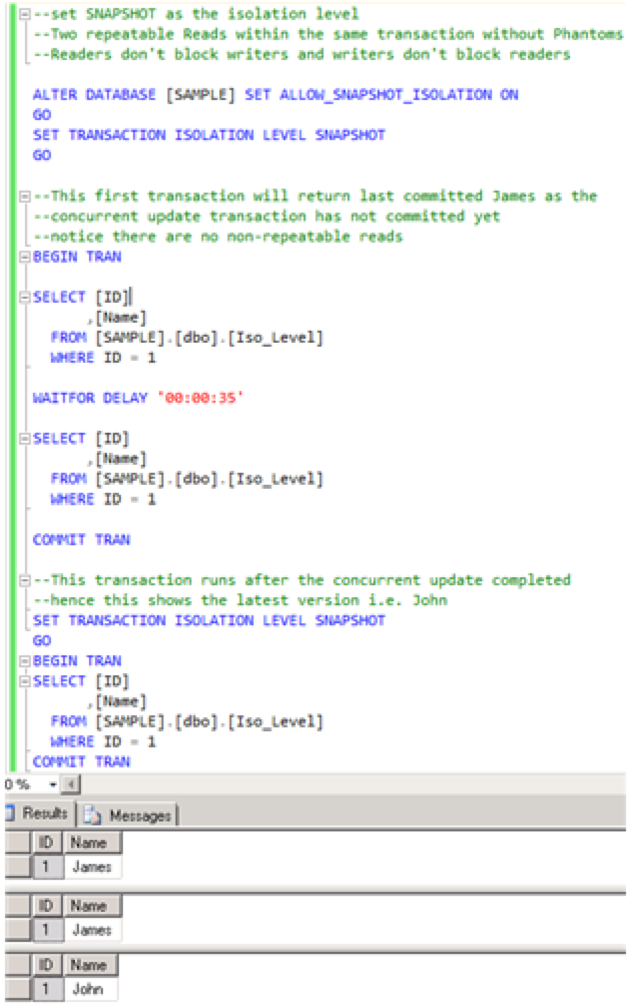
```
      ,[Name]
  FROM [SAMPLE].[dbo].[Iso_Level]
  WHERE ID = 1

COMMIT TRAN

--This SECOND TRANSACTION runs after the concurrent update completed
--hence this shows the latest version i.e. John
SET TRANSACTION ISOLATION LEVEL SNAPSHOT
GO
BEGIN TRAN
SELECT [ID]
      ,[Name]
  FROM [SAMPLE].[dbo].[Iso_Level]
  WHERE ID = 1
COMMIT TRAN
```

```
--changing James to "John" under SNAPSHOT ISOLATION
--This will run without blocking and the concurrent read
--will continue to see the last committed value i.e. James
BEGIN TRAN

UPDATE [SAMPLE].[dbo].[Iso_Level]
SET NAME = 'John'
WHERE NAME = 'James'
AND ID = 1

WAITFOR DELAY '00:00:20'

COMMIT TRAN
```



```
--changing James to "John" under SNAPSHOT ISOLATION
--This will run without blocking and the concurrent read
--will continue to see the last committed value i.e. James
BEGIN TRAN

UPDATE [SAMPLE].[dbo].[Iso_Level]
SET NAME = 'John'
WHERE NAME = 'James'
AND ID = 1

WAITFOR DELAY '00:00:20'

COMMIT TRAN
```

```
(1 row(s) affected)
```



```
--set SNAPSHOT as the isolation level
--Two repeatable Reads within the same transaction without Phantoms
--Readers don't block writers and writers don't block readers

ALTER DATABASE [SAMPLE] SET ALLOW_SNAPSHOT_ISOLATION ON
GO
SET TRANSACTION ISOLATION LEVEL SNAPSHOT
GO

--This first transaction will return last committed James as the
--concurrent update transaction has not committed yet
--notice there are no non-repeatable reads
BEGIN TRAN

SELECT [ID]
      ,[Name]
  FROM [SAMPLE].[dbo].[Iso_Level]
  WHERE ID = 1

WAITFOR DELAY '00:00:35'

SELECT [ID]
      ,[Name]
  FROM [SAMPLE].[dbo].[Iso_Level]
  WHERE ID = 1

COMMIT TRAN

--This transaction runs after the concurrent update completed
--hence this shows the latest version i.e. John
SET TRANSACTION ISOLATION LEVEL SNAPSHOT
GO
BEGIN TRAN
SELECT [ID]
      ,[Name]
  FROM [SAMPLE].[dbo].[Iso_Level]
  WHERE ID = 1
COMMIT TRAN
```

ID	Name
1	James

ID	Name
1	James

ID	Name
1	John

How does Row Versioning work?

Whenever a row in a table (or index) changes, the new row is recorded with a unique sequence number called “Transaction Sequence Number” or “XSN”. This number represents the corresponding transaction that changed the row. As soon as a row is updated, its previous image (committed value) is stored in the version store in tempdb and the updated row has a pointer link to the previous version and the previous version in-turn has pointers to even older versions (if there exist any). As we can imagine, a transaction might have to traverse through several pointer links in order to locate the version current to the calling transaction. Hence, the current version is the only one that resides on the regular data page and all the earlier versions are kept in the pages in the version store in tempdb.

The main benefit row versioning provides is the ability for concurrent transactions to proceed without blocking each other. As mentioned before, writers always block writers i.e. there can't be two concurrent transactions changing the same data at the same time and if SQL Server allowed that, it would lead to inconsistencies because both will change their respective versions ending up having an undesirable behavior of "update conflicts". SQL automatically prevents this from happening by failing the other transaction (explained later in the article).

As a note of caution, since snapshot isolation level heavily uses tempdb for version keeping, one has to carefully evaluate the performance gains in using Snapshot Isolation vs. the overhead of managing tempdb. There is always a trade-off of using this model such as slower updates etc.

Read Committed Snapshot Isolation (RCSI) Vs. Snapshot Isolation (SI)

We might now be wondering, how is the Read Committed Snapshot (recall the optimistic one) different from the normal Snapshot Isolation level. Don't they both have "Snapshot" in their names? The answer is, No, they aren't same. Though they both use the concept of row versioning but they are not same. Given below are the differences between them.

1. Enabling the isolation level

To enable RCSI, we are only required to turn "READ_COMMITTED_SNAPSHOT" setting ON. By default, this database setting is OFF and SQL runs under the pessimistic version of Read-Committed isolation (the locking one discussed earlier) and not RCSI. There is no need to add "SET" option in the session.

To turn Snapshot isolation ON, we would need to first enable "ALLOW_SNAPSHOT_ISOLATION" option and use the statement "SET TRANSACTION ISOLATION LEVEL SNAPSHOT" in the respective transaction.

2. Different Snapshot Behaviors

An important distinction between RCSI and SI is, **RCSI** is a statement based i.e. a statement only sees the most recent committed values as of the beginning of the statement. If any update happens after the first read and before the second read within the same transaction, the second read will reflect the change and shows the new data. This causes non-repeatable reads due to the statement based behavior. This was demonstrated in the RCSI example above.

In contrast, **Snapshot Isolation** is, transaction-based i.e. it provides a consistent view of data across the entire transaction, unlike RCSI, which is statement based. Just like RCSI, the rows read by a read operation are the most recent committed version but what distinguishes SI from RCSI is the fact that the row version read at the beginning of the transaction remains constant throughout the transaction and do not reflect any changes even though there may be updates occurring on them by a concurrent committed transaction. Hence avoiding non-repeatable reads. This was demonstrated in the Snapshot Isolation example above.

3. Update Conflicts (Lost Updates)

An update conflict is nothing but two processes reading the same piece of data and both change the data and then both update the original data to the new value. This leads to one process completely overwriting the update of another and hence losing the updates. By default, SQL Server prevents lost updates by blocking one of the two processes and letting the other one proceed and only after first one finishes will the second one be allowed to run.

Snapshot Isolation can potentially end up causing update conflicts because it works off the row version at the beginning of the transaction (but SQL won't let that happen, read on). To better understand this behavior, see the example below.

Let's say we have transaction 1 "TX1" and transaction 2 "TX2" and the following actions occur in sequence:

1. TX1 sees the value of price as $25 and changes it to $30 but doesn't commit yet.
2. TX2 also sees the price value as $25 (recall TX2 only sees the last committed value).
3. TX2 tries to add $20 and change the price to $45 but gets blocked as TX1 is holding an exclusive lock.
4. TX1 commits.

Let's stop here for a second and discuss the behaviors of RCSI and SI after step 4.

Under **Snapshot Isolation**, we would imagine that after step 4, TX2 would unblock and update the value to $45, completely overwriting the update done by TX1 (because TX2 saw it as $25 at the beginning of the transaction and hence added $20 to original $25 making the new value as $45). This causes lost updates. Fortunately, SQL Server has a mechanism which prevents lost updates. Under Snapshot Isolation, as soon as TX1 commits (step 4), TX2 aborts and SQL cancels the transaction with an error message "*Snapshot Isolation transaction aborted due to update conflict*". Hence, there are no lost updates.

Under **Read Committed Snapshot Isolation**, as soon as TX1 commits (step 4), TX2 unblocks and completes fine adding $20 to the last committed value of $30 (after TX1 commits) i.e. it wouldn't use the old value of $25. Hence, the final price value becomes $50. The key point to note here is, under RCSI, TX2 sees the latest committed value and is statement-based, unlike SI which sees the same row version throughout the transaction.

Hence, there may be chances of update conflicts (but no lost updates) in Snapshot Isolation leading to transactions abort just like deadlocks but we get the benefit of repeatable reads and no phantom reads.

In contrast, RCSI gives us the benefit of not aborting any transactions at the cost of non-repeatable and phantom reads.

Conclusion

All transactions Isolation levels in SQL Server are for a reason. One needs to decide which one suits the application needs and go with it. Snapshot isolation level protects from all undesirable behaviors and provides the multi-version concurrency control but adds the overhead of managing tempdb for version-store and may also cause transaction aborts upon encountering an update conflict situation.

Serializable, on the other hand, also protects from all undesirable situations but uses excessive locking and has concurrency issues. Therefore, an appropriate isolation level should be chosen considering the pros and cons each one has.

See Also

- Transaction Isolation Levels
- Understanding Isolation Levels