

SQLskills Immersion Event

IE1: Internals and Performance

Module 4: Versioning

Kimberly L. Tripp
Kimberly@SQLskills.com



Overview

- Understanding isolation levels
- Isolation in SQL Server
 - By default, uses locking
 - Optionally, can use versioning (and locking)
- Controlling isolation levels
- Statement-level read consistency
- Transaction-level read consistency
- Isolation summary
- Overhead/monitoring



2

© SQLskills, All rights reserved.
<http://www.SQLskills.com>



Availability, What About [B]locking?

- **ACID transaction design requirements**
 - Atomicity Consistency **Isolation** Durability
- **Isolation levels (session setting shown in sys.dm_exec_sessions)**
 - Read uncommitted (1)
 - Read committed (2)
 - Repeatable reads (3)
 - Serializable (4)
 - Snapshot (5)
- **Default isolation level in every release is ANSI/ISO read committed**
- **SQL Server implementation uses locking for all levels**

ACID Properties

- **Atomicity**
 - A transaction must be an atomic unit of work; either all of its modifications are performed, or none
- **Consistency**
 - When completed, a transaction must leave all data and all related structures in a consistent state
- **Isolation**
 - *A transaction either sees data in the state it was in before another concurrent transaction modified it, or it sees the data after the second transaction has completed, but it does not see an intermediate state*
- **Durability**
 - Transaction should persist despite system failure

Isolation Level: Read Uncommitted

Phenomenon: Dirty Reads

- A read transaction can read another transaction's uncommitted (or in-flight) changes – resulting in "dirty reads"
- DML statements always use exclusive locking
- SQL Server implementation:
 - Row locks are not used (SCH_S locks are used) for the dirty read transaction and locks against data being accessed are not honored
- Resulting phenomenon:
 - Statements execute with the possibility of inaccurate data since the "in-flight" data read may continue to change or even be invalidated (e.g. rolled back)

Isolation Level: Read Committed

Phenomenon: Inconsistent Analysis

- The default behavior in ALL releases
- In-flight transaction's data cannot be read by a read committed transaction – only committed changes are visible
- DML statements always use exclusive locking
- In implementation:
 - Locks are released (for readers – not modifiers) as resources are read, a row may be read more than once in some scenarios
- Resulting phenomenon:
 - Reads are not repeatable through the life of a transaction – as a result a row may not be read consistently during the life of a transaction
 - Don't have a DEFINABLE point in time to which a query reconciles...

Isolation Level: Repeatable Read

Phenomenon: Phantoms

- In-flight transaction's data cannot be read and data modified is accessible only to the repeatable read transaction
- Data read, but not modified is accessible to other transactions for reads, but not DML
- In implementation:
 - Locks are held for the life of a transaction, rows which are read are locked and can be repeatably read during the life of a transaction
- Resulting phenomena:
 - Rows which were not present at the beginning of the transaction can appear – in the result

Isolation Level: Serializable

Phenomenon: None

- In-flight transaction's data cannot be read and data modified is accessible only to the serializable transaction
- Data read, but not modified is accessible to other transactions for reads, but not DML
- In implementation:
 - Locks are held for the life of a transaction and held at higher levels within indexes to prevent rows from entering the "set"
- Implementation side effect:
 - To prevent rows from entering the "set" of data, the "set" of data needs to be locked
 - If appropriate indexes do not exist then higher levels of locking might be necessary (i.e. table-level locking)

Understanding Isolation Levels In the Context of a Multi-Statement Transaction

```
BEGIN TRAN
sql
Q1 = SELECT count(*)
      FROM dbo.tname
      WHERE country = 'USA'
sql
...
sql
Q2 = SELECT count(*)
      FROM dbo.tname
      WHERE country = 'USA'
sql
COMMIT TRAN
```

Read Uncommitted

- Q1 > Q2
- Q1 < Q2
- Q1 = Q2
- The data accessed for Q1 and Q2 is not guaranteed to be committed at the time the row is read
- Locks are neither acquired nor honored by Q1 or Q2 and therefore may change between the two as well as be in-flight (or dirty) during either read
- Because locks are not honored, the reader does not have to wait to read data that's in flight
- Trading off accuracy for concurrency

Understanding Isolation Levels In the context of a Multi-Statement Transaction

```
BEGIN TRAN
sql
Q1 = SELECT count(*)
      FROM dbo.tname
      WHERE country = 'USA'
sql
...
sql
Q2 = SELECT count(*)
      FROM dbo.tname
      WHERE country = 'USA'
sql
COMMIT TRAN
```

Read Committed

- Q1 > Q2
- Q1 < Q2
- Q1 = Q2
- The data accessed for Q1 and Q2 is guaranteed to be ONLY committed data
- Rows accessed during Q1 are *not* locked and therefore may be read inconsistently even in Q1 (non-repeatable reads)
- Because only committed data can be read, the reader may have to wait if a writer is modifying rows
- Allowing *some* accuracy while allowing *some* concurrency

Understanding Isolation Levels In the Context of a Multi-Statement Transaction

```
BEGIN TRAN
sql
Q1 = SELECT count(*)
      FROM dbo.tname
      WHERE country = 'USA'

sql
...
sql
Q2 = SELECT count(*)
      FROM dbo.tname
      WHERE country = 'USA'

sql
COMMIT TRAN
```

Repeatable Read

- Q1 < Q2
- Q1 = Q2
- As the rows in Q1 are read, the shared locks remain
- Only rows accessed by Q1 are locked; this does not prevent new rows from entering set
- Q2 will have at least the same number of rows as Q1
- Locked rows are not accessible to other transactions
- Increasing accuracy while reducing concurrency

Understanding Isolation Levels In the Context of a Multi-Statement Transaction

```
BEGIN TRAN
sql
Q1 = SELECT count(*)
      FROM dbo.tname
      WHERE country = 'USA'

sql
...
sql
Q2 = SELECT count(*)
      FROM dbo.tname
      WHERE country = 'USA'

sql
COMMIT TRAN
```

Serializable

- Q1 = Q2
- Rows accessed by Q1 are locked and cannot change between Q1 and Q2
- Serializable protects the entire set and does not allow the data returned from Q1 to change by any other process – guaranteeing the state of the data
- Uses locks (indexes/tables) to guarantee consistency
- Locked rows are not accessible for modifications to other transactions – this creates the most blocking
- Absolute accuracy while reducing concurrency the most

Isolation in Implementation

- All versions of SQL Server use a locking-based implementation by default
- In the default environment, and to reduce the possibility of various anomalies, transactional duration and possibly higher levels of locking are necessary
 - This results in blocking
- There are benefits of locking, if true “isolation” is desired

Isolation in Implementation

- SQL Server 2005 added an optional row versioning-based isolation, in combination with the locking implementation, and now you can control isolation in essentially 4 different configurations by defining the point in time to which you want your statements to reconcile
- Two end results (and database options) for implementing row-level versioning:
 - Statement-level read consistency
 - “Read Committed Isolation Using Row Versioning” (often referred to as RCSI)
 - Database option: [read_committed_snapshot](#)
 - Transaction-level read consistency
 - “Snapshot Isolation”
 - Database option: [allow_snapshot_isolation](#)

Controlling Isolation Behavior

- Default behavior
- Statement-level read consistency OR
"Read Committed Isolation Using Row Versioning"

```
ALTER DATABASE <database_name>  
SET READ_COMMITTED_SNAPSHOT ON  
WITH ROLLBACK AFTER 5
```

- Transaction-level read consistency OR "Snapshot Isolation"

```
ALTER DATABASE <database_name>  
SET ALLOW_SNAPSHOT_ISOLATION ON
```

- Both database options can be turned on as well:

```
ALTER DATABASE <database_name> SET READ_COM...  
ALTER DATABASE <database_name> SET ALLOW_SNA...
```

Versioning: Impact and Overhead

- Versioning overhead is the same in ALL three configurations:
 - Read_committed_snapshot
 - Allow_snapshot_isolation
 - Or, with both turned on (the versioning is the same with one or both turned on)
- Overhead in tempdb can vary
 - Always tied to the transactions that require the versions
 - Possible for "transaction-level" to require the versions to stick around longer
- Monitor with sys.dm_db_file_space_usage

Read Committed Default Behavior

- All phenomena, except dirty reads, are possible, even in the bounds of a single select query
- In volatile databases a long-running query may produce inconsistent results
 - Can increase isolation to remove phenomena
 - Increasing isolation requires locks to be held longer
 - This can create blocking

Read Committed Snapshot Isolation Database Changed to READ_COMMITTED_SNAPSHOT

- No phenomena are possible in the bounds of a single **read committed** statement
- In volatile databases, a multi-statement transaction may yield different results for different access of the same data
- Each statement is consistent but only for the execution of that statement, not for the life of the transaction (if the transaction has multiple statements)
- Each time data is read by a new statement the latest version is used

Understanding Isolation Levels In the Context of a Multi-Statement Transaction

```
BEGIN TRAN
sql
Q1 = SELECT count(*)
      FROM dbo.tname
      WHERE country = 'USA'

sql
...
sql
Q2 = SELECT count(*)
      FROM dbo.tname
      WHERE country = 'USA'

sql
COMMIT TRAN
```

Statement-Level Read Consistency

- Q1 > Q2
- Q1 < Q2
- Q1 = Q2
- Q1 and Q2 are both guaranteed to be accurate as of the beginning of the *statement* and the “version” of the row cannot change for the life of the *statement*
- RCSI guarantees the accuracy of the *statement* but the data can change and read differently by Q2

Snapshot Isolation Database Changed to ALLOW_SNAPSHOT_ISOLATION

- Setting **ALLOW**s users to ask for snapshot isolation
 - NOT on by default
- **ALL** phenomena and **ALL** default locking behaviors are **EXACTLY** the same unless you explicitly ask for snapshot isolation
- Once requested, no phenomena are possible in the bounds of a transaction running under snapshot isolation
- In volatile databases, a multi-statement transaction will always see the transactionally accurate version which existed when the transaction started
- Versions must stick around longer, multi-statement transactions may have conflicts

Understanding Isolation Levels In the Context of a Multi-Statement Transaction

```
SET TRANSACTION ISOLATION
  LEVEL SNAPSHOT
BEGIN TRAN
sql
Q1 = SELECT count(*)
      FROM dbo.tname
      WHERE country = 'USA'

sql
...
sql
Q2 = SELECT count(*)
      FROM dbo.tname
      WHERE country = 'USA'

sql
COMMIT TRAN
```

Transaction-Level Read Consistency (Snapshot Isolation)

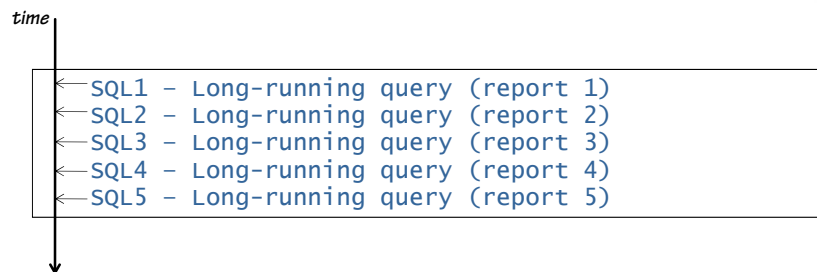
- Q1 = Q2
- Rows accessed by Q1 are consistent at start of transaction
- Data isolated at start of transaction and ensures that the transaction sees the same data at Q2, even if changed by other transactions
- Uses the row version store in tempdb
- Rows are accessible to other transactions

Read Consistency for Multiple Statements

- Imagine running 5 queries (5 large sales reports)
- To what point in time do you want all of the reports to reconcile?
 - The point in time the statement starts
 - The point in time the transaction starts

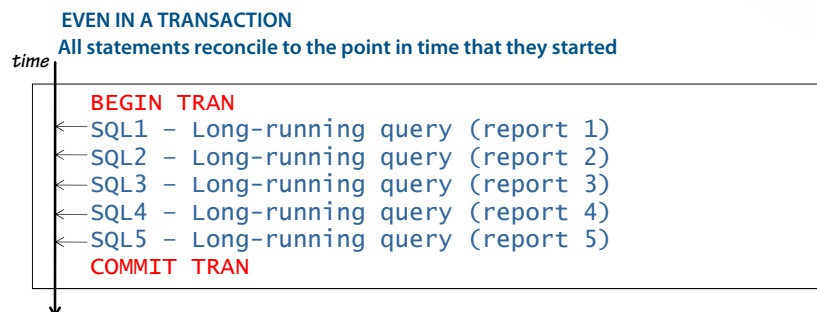
Statement-Level Read Consistency

- Set the read_committed_snapshot database option
- Do nothing else...
- Each query reconciles to the point in time at which it starts



Statement-Level Read Consistency

- Set the read_committed_snapshot database option
- Do nothing else...
- Each query reconciles to the point in time at which it starts

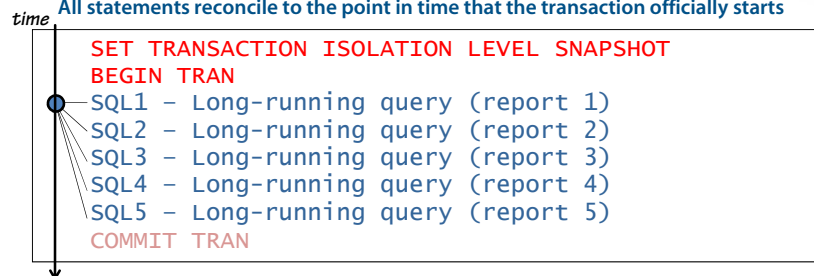


Transaction-Level Read Consistency

- Set the `allow_snapshot_isolation` database option
- Request snapshot isolation (set transaction isolation...)
- All queries reconcile to the point in time the transaction starts (the first real statement)

REQUIRES THE EXPLICIT TRANSACTION (BEGIN/COMMIT) DEFINITION

All statements reconcile to the point in time that the transaction officially starts



Isolation Levels: In Summary

- **Read uncommitted**
 - "Dirty reads" – an option ONLY for readers
 - Any data (even that which is in-flight/locked) can be viewed
- **Read committed using locking (default)**
 - Only committed changes are visible
 - Data in an intermediate state cannot be accessed
 - The point-in-time to which your statement reconciles is not guaranteed, only the state of each row as it's accessed is guaranteed (committed rows only)
- **Read committed using versioning (RCSI)**
 - Statement-level read consistency
 - New non-blocking, non-locking (ex. `SCH_S`), version-based statements (in read committed ONLY)
 - The point-in-time to which your statement reconciles is the time it began

Isolation Levels: In Summary

- **Repeatable reads (uses locking)**
 - All reads are consistent for the life of a transaction
 - Shared locks are NOT released after the data is processed – does not allow writers (does allow other readers)
 - Does not protect entire set (phantoms may occur)
- **Serializable (uses locking)**
 - All reads are consistent (from the time the resource is locked) for the life of the transaction
 - Avoids phantoms – no new records
 - The point-in-time to which your transaction reconciles is not guaranteed, only the state of each set (from when it's first accessed) is guaranteed
- **Snapshot isolation (if requested) uses versioning**
 - Transaction-level consistency using versioning
 - Non-blocking, non-locking, version-based transactions
 - The point-in-time to which your transaction reconciles is the time it began

Allowing Read Committed Using Statement-Level Snapshot

- **Database option**

```
ALTER DATABASE <database_name>
SET READ_COMMITTED_SNAPSHOT ON
WITH ROLLBACK AFTER 5
```
- **No other changes necessary...**
- **If you use READ COMMITTED – no changes to your queries or your applications:**
 - If you “hint” with lock hints (NOLOCK, READUNCOMMITTED, SERIALIZABLE, etc.) then you must remove these hints
 - If you depend on locking – re: queues, readers to wait for change then you might need to use READCOMMITTEDLOCK
- **Changes to blocking... might be extreme!**
- **However, if this is NOT your performance problem (meaning concurrency isn't your bottleneck) then you may hinder performance**
- **Expect this change in behavior at a cost (10-15%)**

Allowing Snapshot Isolation

- Database option

```
ALTER DATABASE <database_name>  
SET ALLOW_SNAPSHOT_ISOLATION ON
```

- Session setting

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT
```

- Changes to applications:

- Request snapshot isolation
- Test for conflict detection (mandatory)

- Expect this change in behavior at a higher cost

Row Length Changes

- Cost in row overhead

- When version is needed, 14 bytes added to row
- When indexes are rebuilt, **versions are removed** (may need FILLFACTOR to reduce possible fragmentation)

- If snapshot, do you depend on locking?

- OK, most of you will say no but if you use queues...
- Tip: Use **READCOMMITTEDLOCK** hint for queues

- If transaction-level is needed and you have multiple writers (with snapshot isolation), you can have conflicts?

- Be sure to have error handling/conflict detection, see Snapshot Isolation whitepaper for details and examples

Management/Monitoring

- Version store in tempdb
- Versions removed when no longer needed
- Read committed using statement-level snapshot won't hold versions as long, in theory, because only statement-level
- Snapshot isolation may have more impact on tempdb as versions held for life of transaction
- Lots of long-running transactions may stress tempdb
- See whitepaper for examples of queries that can help you monitor the version store through DMVs

Review

- Understanding isolation levels
- Isolation in SQL Server
 - By default, uses locking
 - Optionally, can use versioning (and locking)
- Controlling isolation levels
- Statement-level read consistency
- Transaction-level read consistency
- Isolation summary
- Overhead/monitoring

Questions!

