SQLskills Immersion Event

IEPTO1: Performance Tuning and Optimization

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
- Overhead/monitoring
- Isolation summary

Statement Accuracy vs. Data Accessibility/Concurrency

- **Read committed (with locking) does not provide a point in time to which a statement reconciles**
  - This will make sense as we discuss the different isolation levels as well as how they're enforced (and how long locks are held)
- **Sybase (originally) designed "read committed" using the ANSI/ISO standards**
  - Read committed also known as "Inconsistent Analysis"
  - Their design has both pros/cons:
    - Positive: concurrency (data that has not YET been modified, can still be read)
    - Negative: the only accuracy is that a row cannot be read if it's been modified; there's no relationship of that read to the overall statement
      - Leads to inaccuracies: non-repeatable reads and phantoms
- **Oracle is "read committed" by default but they do NOT adhere to the standards and do not have inconsistent analysis in read committed (we CAN achieve this behavior)**

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)
    - Uses locking (when read_committed_snapshot is not set)
    - Uses versioning (when read_committed_snapshot is set)
  - Repeatable reads (3)
  - Serializable (4)
  - Snapshot (5)
- **Default isolation level is ANSI/ISO read committed**
  - NOTE: Default isolation level for Windows Azure SQL Database is read committed using row versioning (RCSI/read_committed_snapshot)
- **SQL Server implementation uses locking for all levels (for writers)**

Isolation Level: Read Uncommitted

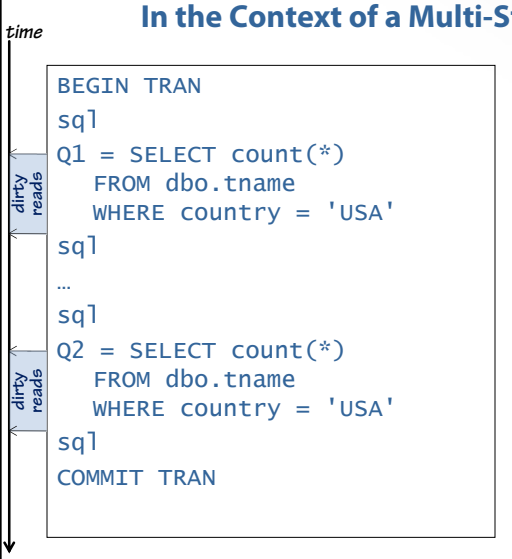
Phenomenon: Dirty Reads

- 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)

SQLskills

Isolation Level: Read Uncommitted

In the Context of a Multi-Statement Transaction



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 the data 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 inflight
- Trading off accuracy for concurrency

SQLskills

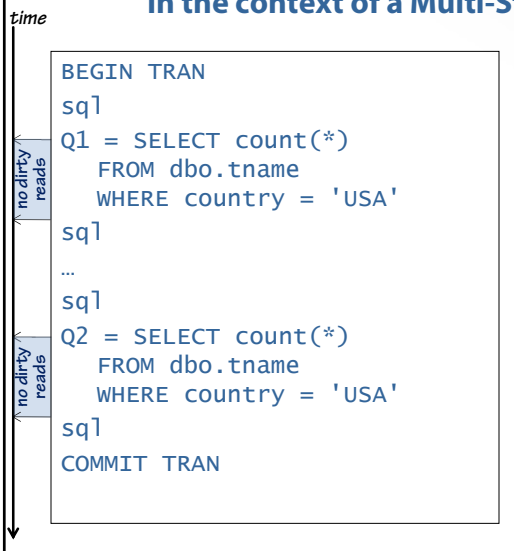
Isolation Level: Read Committed (using Locking)

Phenomenon: Inconsistent Analysis

- Read committed "using locking" is the default behavior in ALL releases
- In-flight transaction's data cannot be read by a read committed transaction – only committed changes are visible
- Modification statements always use locking
- In implementation:
  - Locks are released (for readers – not writers) 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 a reconciles…

Isolation Level: Read Committed

In the context of a Multi-Statement Transaction



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**

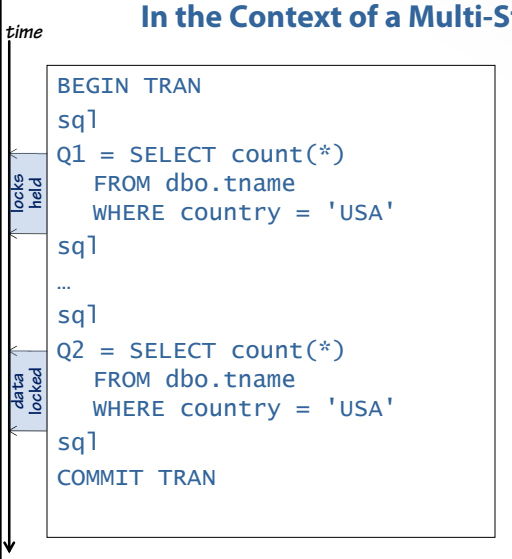
Isolation Level: Repeatable Read

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: Repeatable Read

In the Context of a Multi-Statement Transaction



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 (phantoms)
- 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

Isolation Level: Serializable

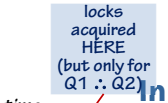
Phenomenon: None

- 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)

SQLskills

Understanding Isolation Levels

In the Context of a Multi-Statement Transaction



time

```
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

SQLskills

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 four 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 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**
  - Prior to SQL Server 2012 this DMV only worked for tempdb
    - RETURNS: database_id, file_id, filegroup_id, total_page_count allocated_extent_page_count, unallocated_extent_page_count, **version_store_reserved_page_count**, user_object_reserved_page_count, internal_object_reserved_page_count, mixed_extent_page_count
  - In SQL Server 2012, this can be used in any database and returns nulls for version_store_reserved_page_count, user_object_reserved_page_count, and internal_object_reserved_page_count

Isolation Level: Read Committed (using Locking)
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

Isolation Level: Read Committed (using Versioning)

Database Changed to READ_COMMITTED_SNAPSHOT

- No phenomena are possible in the bounds of a single read committed statement
- Only used by statements that are in READ COMMITTED isolation
  - Statements cannot have lock hints
    - Statement-level lock / isolation hints override this
    - If your query uses NOLOCK then you do NOT use versions; you must remove lock hints
- In volatile databases, a multi-statement transaction may yield different results for subsequent 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

Isolation Level: Read Committed (using Versioning)

In the Context of a Multi-Statement Transaction

time

```
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

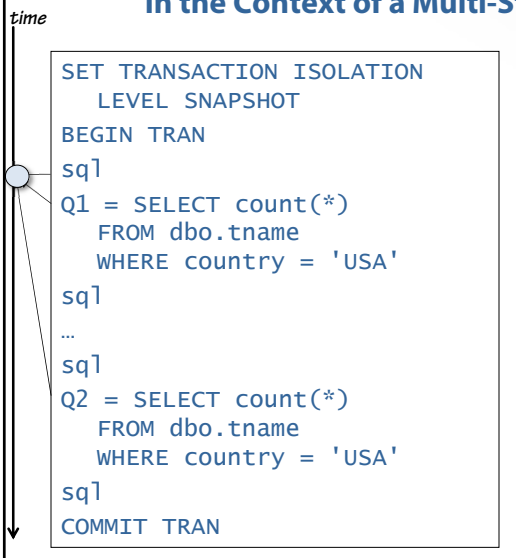
Isolation Level: Snapshot Isolation

Database Changed to ALLOW_SNAPSHOT_ISOLATION

- Setting ALLOWS users to ask for versioning by requesting snapshot isolation
  - NOT on by default
- ALL phenomena and ALL default locking behaviors are EXACTLY the same unless you explicitly ask for versioning through 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

Isolation Level: Snapshot Isolation

In the Context of a Multi-Statement Transaction



Transaction-Level Read Consistency (Snapshot Isolation)

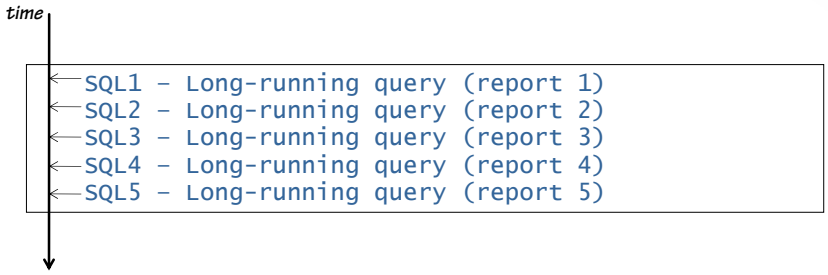
- Q1 = Q2
- Rows accessed by Q1 are consistent at start of transaction
- Data isolated at start of transaction and ensures that the transaction sees the same data at Q2, even if changed by other transactions
- Uses the row version store in tempdb
- Rows are accessible to other transactions

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

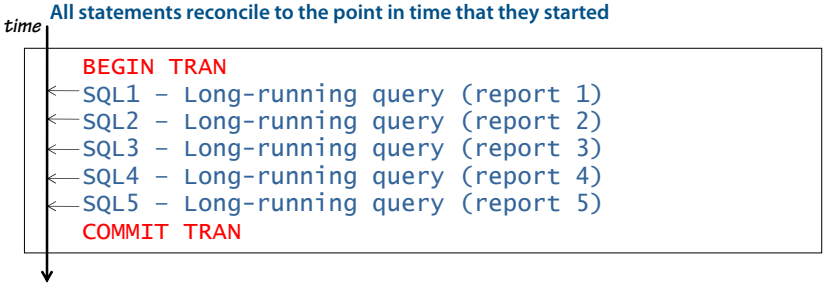
- Set the `READ_COMMITTED_SNAPSHOT` database option
- Do nothing else…
- Each query reconciles to the point in time at which it starts



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

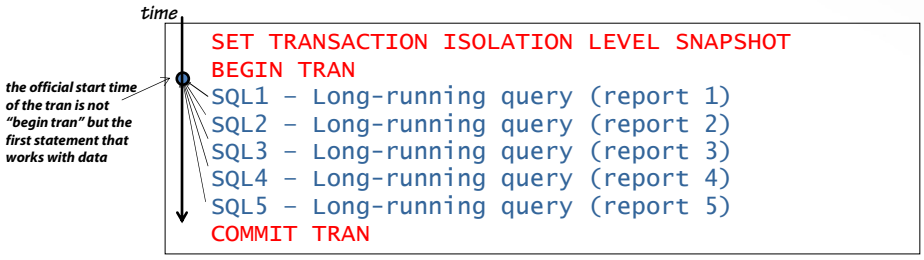
EVEN IN A TRANSACTION
All statements reconcile to the point in time that they started



Transaction-Level Read Consistency (1 of 2)

- 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)

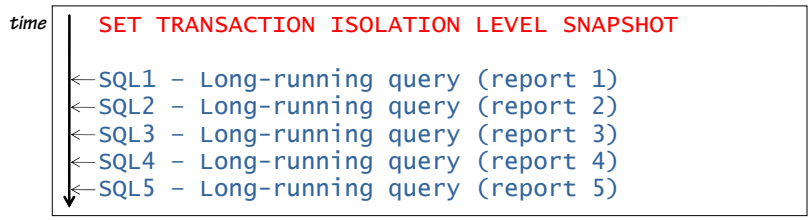
REQUIRES THE EXPLICIT TRANSACTION (BEGIN/COMMIT) DEFINITION
All statements reconcile to the point in time that the transaction officially starts



Transaction-Level Read Consistency (2 of 2)

- Set the ALLOW_SNAPSHOT_ISOLATION database option
- Request snapshot isolation (SET TRANSACTION ISOLATION…)
- All queries reconcile to the point in time the transaction starts
- But… you didn't say BEGIN TRAN

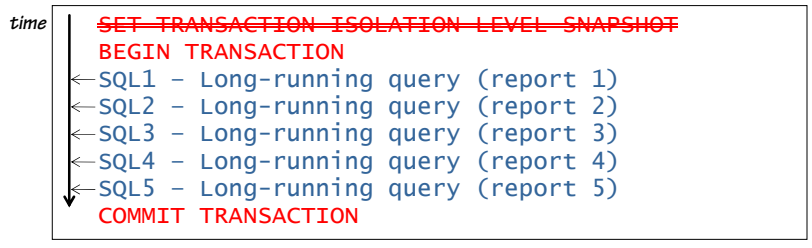
All statements reconcile to the point in time that the STATEMENT starts!



Read Consistency

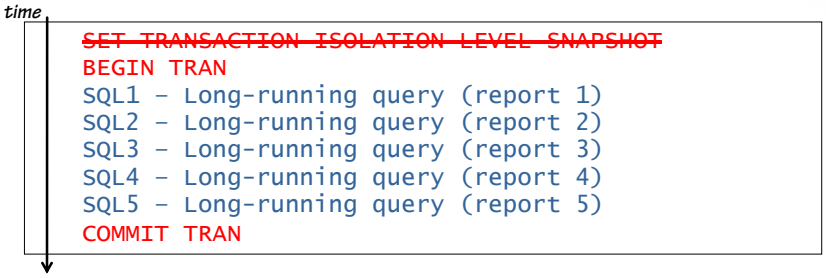
Without the Request for Snapshot Isolation

- Set the ALLOW_SNAPSHOT_ISOLATION database option ON
- WHAT IF YOU FORGET to request snapshot isolation
  - If the database option: READ_COMMITTED_SNAPSHOT is ON then, all queries reconcile to the point in time the statement starts
    - This is both with or without an explicit transaction defined
  - If the database option: READ_COMMITTED_SNAPSHOT is OFF and you FORGET to ask for snapshot isolation then, then all queries use **read committed with locking**



Read Consistency Without the Request for Snapshot Isolation

- Set the ALLOW_SNAPSHOT_ISOLATION database option ON
- WHAT IF YOU FORGET to request snapshot isolation
  - If the database option: READ_COMMITTED_SNAPSHOT is ON then, all queries reconcile to the point in time the statement starts
  - If the database option: READ_COMMITTED_SNAPSHOT is OFF and you FORGET to ask for snapshot isolation then, then all queries use **read committed with locking**



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
  - Make sure your transactions have error handling (use TRY…CATCH)
    - Test for conflict detection (mandatory = you don't have a choice, SQL will error/kill the 2nd updater)
- 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 versioning, 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
  - **IMPORTANT:** versions can only be removed when the low-watermark SERVERWIDE has moved up (in time). If you do server consolidation (with multiple databases on the same SERVER) you need to be very careful that you don't have one database with really long transactions forcing other database's OLTP activity to be KEPT in the version store longer than it was before consolidation!
- 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 IN ANY DATABASE THAT ALLOWS VERSIONING may stress tempdb
- See whitepaper for examples of queries that can help you monitor the version store through DMVs

Isolation: Key Points (1 of 2)

- Consider turning on `ALLOW_SNAPSHOT_ISOLATION` to determine OLTP overhead and monitor version-store activity (without changing any other behaviors)
- Most common to use statement-level read consistency / RCSI / "Read Committed Isolation Using Row Versioning"
  - No problems with update conflicts because every statement uses versions that are applicable to that statement's starting time
  - Little to no code changes required for statements **currently** using read committed with locking; this database option seemlessly makes those statements switch to using versioning (only statements with hints need to be changed)
- Often, `ALLOW_SNAPSHOT_ISOLATION` is also ON but more limited in use
  - Only use "snapshot isolation" for multi-statement reporting "transactions"
    - SET TRANSACTION ISOLATION LEVEL SNAPSHOT must be added to the code…
    - NO update conflicts if you're not setting transaction isolation level snapshot with transactions that make modifications

Isolation: Key Points (2 of 2)

- **Using locking**
  - You are always trading off between accuracy and concurrency
- **Using versioning**
  - You increase concurrency and accuracy with overhead in tempdb
- **If you are trying to do real-time reporting in an OLTP environment**
  - You will get inconsistencies/inaccuracies/anomalies in your long running reads unless you increase locks (which creates more and more blocking)
  - You should consider versioning
    - You'll want to make sure you optimize tempdb
      - Whitepaper: **Working with tempdb in SQL Server 2005**
      - http://technet.microsoft.com/en-us/library/cc966545.aspx
    - You'll want to make sure you understand versioning
      - Whitepaper: **SQL Server 2005 Row Versioning-Based Transaction Isolation**
      - http://msdn.microsoft.com/en-us/library/ms345124.aspx

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**
- **Overhead/monitoring**
- **Isolation summary**

Questions!

SQLskills
immerse yourself in sql server

Isolation Levels: Quick Review (1 of 2)

- **(1): Read uncommitted**
  - "Dirty reads" – an option ONLY for readers
  - Any data (even that which is in-flight/locked) can be viewed
- **(2): 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)
- **(2): Read committed using versioning (read_committed_snapshot)**
  - Statement-level read consistency
  - New non-blocking, non-locking (i.e. 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: Quick Review (2 of 2)

- **(3): 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)
- **(4): 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
- **(5): Snapshot isolation uses versioning (db: allow_snapshot_isolation)**
  - Transaction-level consistency using versioning
  - Non-blocking, non-locking, version-based transactions
  - The point-in-time to which your transaction reconciles is the time the transaction began (this must be requested with SET TRANSACTION ISOLATION SNAPSHOT)

"What If" Scenarios

- **Scenario 1: Code changes for transaction-level read consistency made but the database option allow_snapshot_isolation gets turned off**
  - Client application code:

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT
<executes without error>
BEGIN TRAN
<executes without error>
SELECT * FROM [dbo].[member];
Msg 3952, Level 16, State 1, Line 5
Snapshot isolation transaction failed accessing database
'Credit' because snapshot isolation is not allowed in this
database. Use ALTER DATABASE to allow snapshot isolation.
```

"What If" Scenarios

- **Scenario 2: Cross-database transactions with databases of different versioning configurations**
  - DB1 – read_committed_snapshot + allow_snapshot_isolation
  - DB2 – neither is enabled
  - Client application code:

```
USE [DB1];
GO
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
GO
BEGIN TRAN
SELECT * FROM [DB1].[schema].[table]; <no problem>
SELECT / UPDATE / ANYTHING FROM [DB2].[schema].[table];
Msg 3952, Level 16, State 1, Line 5
Snapshot isolation transaction failed accessing database
'Credit' because snapshot isolation is not allowed in this
database. Use ALTER DATABASE to allow snapshot isolation.
```


"What If" Scenarios

- **Scenario 3: Cross-database transactions with databases of different versioning configurations**
  - DB1 – read_committed_snapshot + allow_snapshot_isolation
  - DB2 – neither is enabled
  - Client application code:

```
USE [DB1];
GO
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
GO
BEGIN TRAN
SELECT * FROM [DB1].[schema].[table]; <no problem>
SELECT / UPDATE / ANYTHING FROM [DB2].[schema].[table];
```

  - NO PROBLEMS!
    - Reads in DB1 are read-committed using versions (no waits)
    - Reads in DB2 are read-committed using locking (possible [indefinite] waits)