

Module 6

Mixed Workloads and Effective Analysis

Using Snapshot Isolation

➔ Kimberly L. Tripp
➔ SQLskills.com



Overview

- ➔ **Availability, What About Locking?**
- ➔ **Mixed Workloads**
- ➔ **ACID Properties**
- ➔ **Understanding Isolation Levels**
- ➔ **Controlling Isolation Levels**
- ➔ **Allowing Read Committed Using Statement-level Snapshot**
- ➔ **Allowing Snapshot Isolation**
- ➔ **Upgrade impact**
- ➔ **Potential Conflicts**

©Kimberly L. Tripp
Designing for Performance

Availability, What About Locking?

- **ACID Transaction Design Requirements**
 - **Atomicity Consistency Isolation Durability**
- **Isolation Levels**
 - **Level 0 – Read Uncommitted**
 - **Level 1 – Read Committed**
 - **Level 2 – Repeatable Reads**
 - **Level 3 – Serializable**
- **Default Isolation Level in BOTH 2000/2005 is ANSI/ISO Level 1, Read Committed**
- **In implementation this default level uses locking**

©Kimberly L. Tripp
Designing for Performance

Mixed Workload Alternatives

- **Backup and Restore**
- **SQL IS**
- **Replication**
- **Analysis Services**
- **Database Snapshots**
- **Real-time Mixed Workloads – LIVE**
 - ⇒ **Snapshot Isolation**

©Kimberly L. Tripp
Designing for Performance

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

©Kimberly L. Tripp
Designing for Performance

Isolation Level 0 – 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
- In 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 (rolled back)

©Kimberly L. Tripp
Designing for Performance

Isolation Level 1 – 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

©Kimberly L. Tripp
Designing for Performance

Isolation Level 2 – 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

©Kimberly L. Tripp
Designing for Performance

Isolation Level 3 – Serializable Phenomenon: None

Key range
and index
reminder

- 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.

©Kimberly L. Tripp
Designing for Performance

Isolation in Implementation

- All versions of SQL Server – including SQL Server 2005 – 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 – resulting in blocking
- There are benefits of locking, if true "isolation" is desired

©Kimberly L. Tripp
Designing for Performance

Isolation in Implementation (cont'd)

- SQL Server 2005 combines an optional row versioning-based implementation with this locking-based implementation
- Two end results (and methods) for implementing row-level versioning
 - Read Committed using Statement-level Snapshot
 - Also known as:
Statement-level Read Consistency
 - Transaction-level Read Consistency
 - Snapshot Isolation

©Kimberly L. Tripp
Designing for Performance

Setting Isolation Behavior

- Default Behavior
- Read Committed using Statement-Level Snapshot

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

- Snapshot Isolation

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

- Both RCSI and Snapshot Isolation

```
ALTER DATABASE <database_name> SET RCSLS...  
ALTER DATABASE <database_name> SET Snapshot...
```

©Kimberly L. Tripp
Designing for Performance

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

©Kimberly L. Tripp
Designing for Performance

Tips to Minimize Blocking

- Write efficient transactions – keep them short and in one batch
- Do not allow interaction in the midst of the batch
- Use indexes to help SQL Server find – and lock – only the necessary data
- Consider estimates for long running queries and/or migrating data to a secondary analysis server
- Problems with locking becoming blocking are often when long running (and conflicting) transactions execute

©Kimberly L. Tripp
Designing for Performance

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 is not guaranteed to be accurate/committed
- Rows accessed in Q1 are *not* locked and therefore may change before Q2 and even be in-flight during Q2
- The data accessed for Q2 is not guaranteed to be accurate/committed

©Kimberly L. Tripp
Designing for Performance

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 is guaranteed to be **ONLY** committed data
- Rows accessed during Q1 are *not* locked and therefore may be read inconsistently even in Q1
- The data accessed for Q2 is guaranteed to be **ONLY** committed data

©Kimberly L. Tripp
Designing for Performance

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
- 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

©Kimberly L. Tripp
Designing for Performance

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 Q2 to change
- Uses locks to guarantee consistency
- Locked rows are not accessible for modifications to other transactions

©Kimberly L. Tripp
Designing for Performance

Locking Prevents Conflicts

At the expense of blocking

- No other transactions can invalidate the data set through modifications
- Is this always what you want or need?
 - Queuing applications typically want locks, if someone is modifying that row – we want to go to another not see last transactional state
 - Very volatile prices – don't want to give them the last price if it's currently being updated, so wait... (*but the update's fast*)
- What about long running transactions or cases where you don't need absolute current...

©Kimberly L. Tripp
Designing for Performance

Read Committed Snapshot Isolation

Database Changed to READ_COMMITTED_SNAPSHOT

- No phenomena are possible in the bounds of a single 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

©Kimberly L. Tripp
Designing for Performance

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
```

RCSI

- > 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

Uses
tempdb

©Kimberly L. Tripp
Designing for Performance

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 behavior is **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

©Kimberly L. Tripp
Designing for Performance

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
```

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

©Kimberly L. Tripp
Designing for Performance

Isolation Levels: In Summary

- **READ UNCOMMITTED (Level 0)**
 - “Dirty Reads” – An option **ONLY** for readers
 - Any data (even that which is in-flight/locked) can be viewed
- **READ COMMITTED (Level 1 – Default)**
 - Only committed changes are visible
 - Data in an intermediate state cannot be accessed
- **Read Committed w/statement-level Snapshot**
 - Statement-level read consistency
 - New non-blocking, non-locking (ex. SCH_S), version-based Level 1

©Kimberly L. Tripp
Designing for Performance

Isolation Levels: In Summary (cont'd)

- **REPEATABLE READS (Level 2)**
 - 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 (Level 3)**
 - All reads are consistent for the life of a transaction
 - Avoids phantoms – no new records
- **Snapshot Isolation – 2005**
 - Transaction-Level consistency using snapshot
 - New non-blocking, non-locking, version-based transactions

©Kimberly L. Tripp
Designing for Performance

Controlling Isolation Levels

Table-level changes

From Clause, per table (no spaces)

- Level 0 – READUNCOMMITTED, NOLOCK
- Level 1 – READCOMMITTED (locking)
- Level 1 – READCOMMITTED (versioning)
 - Only in 2005 and *only* if the database option to READ_COMMITTED_SNAPSHOT is on
 - Can be overridden with READCOMMITTEDLOCK
- Level 2 – REPEATABLE READ
- Level 3 – SERIALIZABLE, HOLDLOCK

```
FROM dbo.titles WITH(READUNCOMMITTED)
JOIN dbo.publishers WITH(SERIALIZABLE)
```

©Kimberly L. Tripp
Designing for Performance

Controlling Isolation Levels

Session-level changes (cont'd)

Session level settings impact entire session but can be overridden with table-level settings

- Level 0 – READ UNCOMMITTED
- Level 1 – READ COMMITTED
- Level 2 – REPEATABLE READ
- Level 3 – SERIALIZABLE

SET TRANSACTION ISOLATION LEVEL *opt*
READ UNCOMMITTED
READ COMMITTED
REPEATABLE READ
SERIALIZABLE
SNAPSHOT

Only in 2005 and *only* if the database option is on

©Kimberly L. Tripp
Designing for Performance

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...
- No changes to your queries or your applications – they automatically return with statement-level consistency
(NOTE: You may need to modify your applications if you depend on locking
– re: queues, readers to wait for change)
- Changes to blocking...
- However, if this is NOT your performance problem (meaning concurrency isn't your bottleneck) then you may hinder performance not improve

➤ Expect this change in behavior at a cost

©Kimberly L. Tripp
Designing for Performance

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
- **Expect this change in behavior at a *higher* cost**

©Kimberly L. Tripp
Designing for Performance

Potential Issues

- **Cost in row overhead – when enabled, 14 bytes added to row**
- **If snapshot, do you depend on locking?**
 - OK, most of you will say no but what about status queues...
 - Tip: Use READCOMMITTEDLOCK hint
- **If Snapshot Isolation, could you have conflicts?**
 - Be sure to have proper conflict detection and error handling, see whitepaper for details and example

©Kimberly L. Tripp
Designing for Performance

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 monitoring with DMVs**

©Kimberly L. Tripp
Designing for Performance

Review

- **Availability, What About Locking?**
- **Mixed Workloads**
- **ACID Properties**
- **Understanding Isolation Levels**
- **Controlling Isolation Levels**
- **Allowing Read Committed Using Statement-level Snapshot**
- **Allowing Snapshot Isolation**
- **Upgrade impact**
- **Potential Conflicts**

©Kimberly L. Tripp
Designing for Performance