Snapshot Isolation in SQL Server

09/15/2021

Snapshot isolation enhances concurrency for OLTP applications.

Understanding Snapshot Isolation and Row Versioning

Once snapshot isolation is enabled, updated row versions for each transaction must be maintained. Prior to SQL Server 2019, these versions were stored in **tempdb**. SQL Server 2019 introduces a new feature, Accelerated Database Recovery (ADR) which requires its own set of row versions. So, as of SQL Server 2019, if ADR is not enabled, row versions are kept in **tempdb** as always. If ADR is enabled, then all row versions, both related to snapshot isolation and ADR, are kept in ADR's Persistent Version Store (PVS), which is located in the user database in a filegroup which the user specifies. A unique transaction sequence number identifies each transaction, and these unique numbers are recorded for each row version. The transaction works with the most recent row versions having a sequence number before the sequence number of the transaction. Newer row versions created after the transaction has begun are ignored by the transaction.

The term "snapshot" reflects the fact that all queries in the transaction see the same version, or snapshot, of the database, based on the state of the database at the moment in time when the transaction begins. No locks are acquired on the underlying data rows or data pages in a snapshot transaction, which permits other transactions to execute without being blocked by a prior uncompleted transaction. Transactions that modify data do not block transactions that read data, and transactions that read data do not block transactions that write data, as they normally would under the default READ COMMITTED isolation level in SQL Server. This non-blocking behavior also significantly reduces the likelihood of deadlocks for complex transactions.

Snapshot isolation uses an optimistic concurrency model. If a snapshot transaction attempts to commit modifications to data that has changed since the transaction began, the transaction will roll back and an error will be raised. You can avoid this by using UPDLOCK hints for SELECT statements that access data to be modified. For more information, see Hints (Transact-SQL).

Snapshot isolation must be enabled by setting the ALLOW_SNAPSHOT_ISOLATION ON database option before it is used in transactions. This activates the mechanism for storing row versions in the temporary database (**tempdb**). You must enable snapshot isolation in each database that uses it with the Transact-SQL ALTER DATABASE statement. In this respect, snapshot isolation differs from the traditional isolation levels of READ COMMITTED, REPEATABLE READ, SERIALIZABLE, and READ UNCOMMITTED, which require no configuration. The following statements activate snapshot isolation and replace the default READ COMMITTED behavior with SNAPSHOT:

SQL

```sql
ALTER DATABASE MyDatabase
SET ALLOW_SNAPSHOT_ISOLATION ON

ALTER DATABASE MyDatabase
SET READ_COMMITTED_SNAPSHOT ON
```

Setting the READ_COMMITTED_SNAPSHOT ON option allows access to versioned rows under the default READ COMMITTED isolation level. If the READ_COMMITTED_SNAPSHOT option is set to OFF, you must explicitly set the Snapshot isolation level for each session in order to access versioned rows.

Managing Concurrency with Isolation Levels

The isolation level under which a Transact-SQL statement executes determines its locking and row versioning behavior. An isolation level has connection-wide scope, and once set for a connection with the SET TRANSACTION ISOLATION LEVEL statement, it remains in effect until the connection is closed or another isolation level is set. When a connection is closed and returned to the pool, the isolation level from the last SET TRANSACTION ISOLATION LEVEL statement is retained. Subsequent connections reusing a pooled connection use the isolation level that was in effect at the time the connection is pooled.

Individual queries issued within a connection can contain lock hints that modify the isolation for a single statement or transaction but do not affect the isolation level of the connection. Isolation levels or lock hints set in stored procedures or functions do not change the isolation level of the connection that calls them and are in effect only for the duration of the stored procedure or function call.

Four isolation levels defined in the SQL-92 standard were supported in early versions of SQL Server:

- READ UNCOMMITTED is the least restrictive isolation level because it ignores locks placed by other transactions. Transactions executing under READ UNCOMMITTED can read modified data values that have not yet been committed by other transactions; these are called "dirty" reads.

- READ COMMITTED is the default isolation level for SQL Server. It prevents dirty reads by specifying that statements cannot read data values that have been modified but not yet committed by other transactions. Other transactions can still modify, insert, or delete data between executions of individual statements within the current transaction, resulting in non-repeatable reads, or "phantom" data.

- REPEATABLE READ is a more restrictive isolation level than READ COMMITTED. It encompasses READ COMMITTED and additionally specifies that no other transactions can modify or delete data that has been read by the current transaction until the current transaction commits. Concurrency is lower than for READ COMMITTED because shared locks on read data are held for the duration of the transaction instead of being released at the end of each statement.

- SERIALIZABLE is the most restrictive isolation level, because it locks entire ranges of keys and holds the locks until the transaction is complete. It encompasses REPEATABLE READ and adds the restriction that other transactions cannot insert new rows into ranges that have been read by the transaction until the transaction is complete.

For more information, refer to the Transaction Locking and Row Versioning Guide.

Snapshot Isolation Level Extensions

SQL Server introduced extensions to the SQL-92 isolation levels with the introduction of the SNAPSHOT isolation level and an additional implementation of READ COMMITTED. The READ_COMMITTED_SNAPSHOT isolation level can transparently replace READ COMMITTED for all transactions.

- SNAPSHOT isolation specifies that data read within a transaction will never reflect changes made by other simultaneous transactions. The transaction uses the data row versions that exist when the transaction begins. No locks are placed on the data when it is read, so SNAPSHOT transactions do not block other transactions from writing data. Transactions that write data do not block snapshot transactions from reading data. You need to enable snapshot isolation by setting the ALLOW_SNAPSHOT_ISOLATION database option in order to use it.

- The READ_COMMITTED_SNAPSHOT database option determines the behavior of the default READ COMMITTED isolation level when snapshot isolation is enabled in a database. If you do not explicitly specify READ_COMMITTED_SNAPSHOT ON, READ COMMITTED is applied to all implicit transactions. This produces the same behavior as setting READ_COMMITTED_SNAPSHOT OFF (the default). When READ_COMMITTED_SNAPSHOT OFF is in effect, the Database Engine uses shared locks to enforce the default isolation level. If you set the READ_COMMITTED_SNAPSHOT database option to ON, the database engine uses row versioning and snapshot isolation as the default, instead of using locks to protect the data.

How Snapshot Isolation and Row Versioning Work

When the SNAPSHOT isolation level is enabled, each time a row is updated, the SQL Server Database Engine stores a copy of the original row in **tempdb**, and adds a transaction sequence number to the row. The following is the sequence of events that occurs:

- A new transaction is initiated, and it is assigned a transaction sequence number.
- The Database Engine reads a row within the transaction and retrieves the row version from **tempdb** whose sequence number is closest to, and lower than, the transaction sequence number.
- The Database Engine checks to see if the transaction sequence number is not in the list of transaction sequence numbers of the uncommitted transactions active when the snapshot transaction started.
- The transaction reads the version of the row from **tempdb** that was current as of the start of the transaction. It will not see new rows inserted after the transaction was started because those sequence number values will be higher than the value of the transaction sequence number.
- The current transaction will see rows that were deleted after the transaction began, because there will be a row version in **tempdb** with a lower sequence number value.

The net effect of snapshot isolation is that the transaction sees all of the data as it existed at the start of the transaction, without honoring or placing any locks on the underlying tables. This can result in performance improvements in situations where there is contention.

A snapshot transaction always uses optimistic concurrency control, withholding any locks that would prevent other transactions from updating rows. If a snapshot transaction attempts to commit an update to a row that was changed after the transaction began, the transaction is rolled back, and an error is raised.

Working with Snapshot Isolation in ADO.NET

Snapshot isolation is supported in ADO.NET by the SqlTransaction class. If a database has been enabled for snapshot isolation but is not configured for READ_COMMITTED_SNAPSHOT ON, you must initiate a SqlTransaction using the **IsolationLevel.Snapshot** enumeration value when calling the BeginTransaction method. This code fragment assumes that connection is an open SqlConnection object.

C#

```
SqlTransaction sqlTran =
  connection.BeginTransaction(IsolationLevel.Snapshot);
```

Example

The following example demonstrates how the different isolation levels behave by attempting to access locked data, and it is not intended to be used in production code.

The code connects to the **AdventureWorks** sample database in SQL Server and creates a table named **TestSnapshot** and inserts one row of data. The code uses the ALTER DATABASE Transact-SQL statement to turn on snapshot isolation for the database, but it does not set the READ_COMMITTED_SNAPSHOT option, leaving the default READ COMMITTED isolation-level behavior in effect. The code then performs the following actions:

- It begins, but does not complete, sqlTransaction1, which uses the SERIALIZABLE isolation level to start an update transaction. This has the effect of locking the table.
- It opens a second connection and initiates a second transaction using the SNAPSHOT isolation level to read the data in the **TestSnapshot** table. Because snapshot isolation is enabled, this transaction can read the data that existed before sqlTransaction1 started.
- It opens a third connection and initiates a transaction using the READ COMMITTED isolation level to attempt to read the data in the table. In this case, the code cannot read the data because it cannot read past the locks placed on the table in the first transaction and times out. The same result would occur if the REPEATABLE READ and SERIALIZABLE isolation levels were used because these isolation levels also cannot read past the locks placed in the first transaction.
- It opens a fourth connection and initiates a transaction using the READ UNCOMMITTED isolation level, which performs a dirty read of the uncommitted value in sqlTransaction1. This value may never actually exist in the database if the first transaction is not committed.

- It rolls back the first transaction and cleans up by deleting the **TestSnapshot** table and turning off snapshot isolation for the **AdventureWorks** database.

> ⓘ Note
>
> The following examples use the same connection string with connection pooling turned off. If a connection is pooled, resetting its isolation level does not reset the isolation level at the server. As a result, subsequent connections that use the same pooled inner connection start with their isolation levels set to that of the pooled connection. An alternative to turning off connection pooling is to set the isolation level explicitly for each connection.

C#

```
// Assumes GetConnectionString returns a valid connection string
// where pooling is turned off by setting Pooling=False;.
var connectionString = GetConnectionString();
using (SqlConnection connection1 = new(connectionString))
{
    // Drop the TestSnapshot table if it exists
    connection1.Open();
    SqlCommand command1 = connection1.CreateCommand();
    command1.CommandText = "IF EXISTS "
        + "(SELECT * FROM sys.tables WHERE name=N'TestSnapshot') "
        + "DROP TABLE TestSnapshot";
    try
    {
        command1.ExecuteNonQuery();
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
    // Enable Snapshot isolation
    command1.CommandText =
        "ALTER DATABASE AdventureWorks SET ALLOW_SNAPSHOT_ISOLATION ON";
    command1.ExecuteNonQuery();

    // Create a table named TestSnapshot and insert one row of data
    command1.CommandText =
        "CREATE TABLE TestSnapshot (ID int primary key, valueCol int)";
    command1.ExecuteNonQuery();
    command1.CommandText =
        "INSERT INTO TestSnapshot VALUES (1,1)";
    command1.ExecuteNonQuery();

    // Begin, but do not complete, a transaction to update the data
    // with the Serializable isolation level, which locks the table
    // pending the commit or rollback of the update. The original
    // value in valueCol was 1, the proposed new value is 22.
    SqlTransaction transaction1 =
        connection1.BeginTransaction(IsolationLevel.Serializable);
    command1.Transaction = transaction1;
    command1.CommandText =
        "UPDATE TestSnapshot SET valueCol=22 WHERE ID=1";
    command1.ExecuteNonQuery();

    // Open a second connection to AdventureWorks
    using (SqlConnection connection2 = new(connectionString))
    {
        connection2.Open();
        // Initiate a second transaction to read from TestSnapshot
        // using Snapshot isolation. This will read the original
        // value of 1 since transaction1 has not yet committed.
        SqlCommand command2 = connection2.CreateCommand();
        SqlTransaction transaction2 =
            connection2.BeginTransaction(IsolationLevel.Snapshot);
        command2.Transaction = transaction2;
        command2.CommandText =
            "SELECT ID, valueCol FROM TestSnapshot";
        SqlDataReader reader2 = command2.ExecuteReader();
        while (reader2.Read())
        {
            Console.WriteLine("Expected 1,1 Actual "
                + reader2.GetValue(0)
                + "," + reader2.GetValue(1));
        }
        transaction2.Commit();
    }

    // Open a third connection to AdventureWorks and
    // initiate a third transaction to read from TestSnapshot
    // using ReadCommitted isolation level. This transaction
    // will not be able to view the data because of
    // the locks placed on the table in transaction1
    // and will time out after 4 seconds.
    // You would see the same behavior with the
    // RepeatableRead or Serializable isolation levels.
    using (SqlConnection connection3 = new(connectionString))
    {
        connection3.Open();
        SqlCommand command3 = connection3.CreateCommand();
        SqlTransaction transaction3 =
            connection3.BeginTransaction(IsolationLevel.ReadCommitted);
        command3.Transaction = transaction3;
        command3.CommandText =
            "SELECT ID, valueCol FROM TestSnapshot";
        command3.CommandTimeout = 4;
        try
        {
            SqlDataReader sqldatareader3 = command3.ExecuteReader();
            while (sqldatareader3.Read())
            {
                Console.WriteLine("You should never hit this.");
            }
            transaction3.Commit();
        }
        catch (Exception ex)
        {
            Console.WriteLine("Expected timeout expired exception: "
                + ex.Message);
            transaction3.Rollback();
        }
    }

    // Open a fourth connection to AdventureWorks and
    // initiate a fourth transaction to read from TestSnapshot
    // using the ReadUncommitted isolation level. ReadUncommitted
    // will not hit the table lock, and will allow a dirty read
    // of the proposed new value 22 for valueCol. If the first
    // transaction rolls back, this value will never actually have
    // existed in the database.
    using (SqlConnection connection4 = new(connectionString))
    {
        connection4.Open();
        SqlCommand command4 = connection4.CreateCommand();
        SqlTransaction transaction4 =
            connection4.BeginTransaction(IsolationLevel.ReadUncommitted);
        command4.Transaction = transaction4;
        command4.CommandText =
            "SELECT ID, valueCol FROM TestSnapshot";
        SqlDataReader reader4 = command4.ExecuteReader();
        while (reader4.Read())
        {
            Console.WriteLine("Expected 1,22 Actual "
                + reader4.GetValue(0)
                + "," + reader4.GetValue(1));
        }

        transaction4.Commit();
    }

    // Roll back the first transaction
    transaction1.Rollback();
}

// CLEANUP
// Delete the TestSnapshot table and set
// ALLOW_SNAPSHOT_ISOLATION OFF
```

```
using (SqlConnection connection5 = new(connectionString))
{
    connection5.Open();
    SqlCommand command5 = connection5.CreateCommand();
    command5.CommandText = "DROP TABLE TestSnapshot";
    SqlCommand command6 = connection5.CreateCommand();
    command6.CommandText =
        "ALTER DATABASE AdventureWorks SET ALLOW_SNAPSHOT_ISOLATION OFF";
    try
    {
        command5.ExecuteNonQuery();
        command6.ExecuteNonQuery();
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}
Console.WriteLine("Done!");
```

Example

The following example demonstrates the behavior of snapshot isolation when data is being modified. The code performs the following actions:

- Connects to the **AdventureWorks** sample database and enables SNAPSHOT isolation.
- Creates a table named **TestSnapshotUpdate** and inserts three rows of sample data.
- Begins, but does not complete, sqlTransaction1 using SNAPSHOT isolation. Three rows of data are selected in the transaction.
- Creates a second **SqlConnection** to **AdventureWorks** and creates a second transaction using the READ COMMITTED isolation level that updates a value in one of the rows selected in sqlTransaction1.
- Commits sqlTransaction2.
- Returns to sqlTransaction1 and attempts to update the same row that sqlTransaction1 already committed. Error 3960 is raised, and sqlTransaction1 is rolled back automatically. The **SqlException.Number** and **SqlException.Message** are displayed in the Console window.
- Executes clean-up code to turn off snapshot isolation in **AdventureWorks** and delete the **TestSnapshotUpdate** table.

C#

```
// Assumes GetConnectionString returns a valid connection string
// where pooling is turned off by setting Pooling=False;.
var connectionString = GetConnectionString();
using (SqlConnection connection1 = new(connectionString))
{
    connection1.Open();
    SqlCommand command1 = connection1.CreateCommand();

    // Enable Snapshot isolation in AdventureWorks
    command1.CommandText =
        "ALTER DATABASE AdventureWorks SET ALLOW_SNAPSHOT_ISOLATION ON";
    try
    {
        command1.ExecuteNonQuery();
        Console.WriteLine(
            "Snapshot Isolation turned on in AdventureWorks.");
    }
    catch (Exception ex)
    {
        Console.WriteLine($"ALLOW_SNAPSHOT_ISOLATION ON failed: {ex.Message}");
    }
    // Create a table
    command1.CommandText =
        "IF EXISTS "
        + "(SELECT * FROM sys.tables "
        + "WHERE name=N'TestSnapshotUpdate')"
        + " DROP TABLE TestSnapshotUpdate";
    command1.ExecuteNonQuery();
    command1.CommandText =
        "CREATE TABLE TestSnapshotUpdate "
        + "(ID int primary key, CharCol nvarchar(100));";
    try
    {
        command1.ExecuteNonQuery();
        Console.WriteLine("TestSnapshotUpdate table created.");
    }
    catch (Exception ex)
    {
        Console.WriteLine($"CREATE TABLE failed: {ex.Message}");
    }
    // Insert some data
    command1.CommandText =
        "INSERT INTO TestSnapshotUpdate VALUES (1,N'abcdefg');"
        + "INSERT INTO TestSnapshotUpdate VALUES (2,N'hijklmn');"
        + "INSERT INTO TestSnapshotUpdate VALUES (3,N'opqrstuv');";
    try
    {
        command1.ExecuteNonQuery();
        Console.WriteLine("Data inserted TestSnapshotUpdate table.");
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }

        // Begin, but do not complete, a transaction
        // using the Snapshot isolation level.
        SqlTransaction transaction1 = default!;
        try
        {
            transaction1 = connection1.BeginTransaction(IsolationLevel.Snapshot);
            command1.CommandText =
                "SELECT * FROM TestSnapshotUpdate WHERE ID BETWEEN 1 AND 3";
            command1.Transaction = transaction1;
            command1.ExecuteNonQuery();
            Console.WriteLine("Snapshot transaction1 started.");

            // Open a second Connection/Transaction to update data
            // using ReadCommitted. This transaction should succeed.
            using (SqlConnection connection2 = new(connectionString))
            {
                connection2.Open();
                SqlCommand command2 = connection2.CreateCommand();
                command2.CommandText = "UPDATE TestSnapshotUpdate SET CharCol="
                    + "N'New value from Connection2' WHERE ID=1";
                SqlTransaction transaction2 =
                    connection2.BeginTransaction(IsolationLevel.ReadCommitted);
                command2.Transaction = transaction2;
                try
                {
                    command2.ExecuteNonQuery();
                    transaction2.Commit();
                    Console.WriteLine(
                        "transaction2 has modified data and committed.");
                }
                catch (SqlException ex)
                {
                    Console.WriteLine(ex.Message);
                    transaction2.Rollback();
                }
                finally
                {
                    transaction2.Dispose();
                }
            }

            // Now try to update a row in Connection1/Transaction1.
            // This transaction should fail because Transaction2
            // succeeded in modifying the data.
            command1.CommandText =
                "UPDATE TestSnapshotUpdate SET CharCol="
                + "N'New value from Connection1' WHERE ID=1";
            command1.Transaction = transaction1;
            command1.ExecuteNonQuery();
            transaction1.Commit();
            Console.WriteLine("You should never see this.");
        }
        catch (SqlException ex)
        {
            Console.WriteLine("Expected failure for transaction1:");
            Console.WriteLine($"  {ex.Number}: {ex.Message}");
        }
        finally
        {
            transaction1.Dispose();
        }
    }

    // CLEANUP:
    // Turn off Snapshot isolation and delete the table
    using (SqlConnection connection3 = new(connectionString))
    {
        connection3.Open();
        SqlCommand command3 = connection3.CreateCommand();
        command3.CommandText =
            "ALTER DATABASE AdventureWorks SET ALLOW_SNAPSHOT_ISOLATION OFF";
        try
        {
            command3.ExecuteNonQuery();
            Console.WriteLine(
                "CLEANUP: Snapshot isolation turned off in AdventureWorks.");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"CLEANUP FAILED: {ex.Message}");
        }
        command3.CommandText = "DROP TABLE TestSnapshotUpdate";
        try
        {
            command3.ExecuteNonQuery();
            Console.WriteLine("CLEANUP: TestSnapshotUpdate table deleted.");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"CLEANUP FAILED: {ex.Message}");
        }
    }
```

Using Lock Hints with Snapshot Isolation

In the previous example, the first transaction selects data, and a second transaction updates the data before the first transaction is able to complete, causing an update conflict when the first transaction tries to update the same row. You can reduce the chance of update conflicts in long-running snapshot transactions by supplying lock hints at the beginning of the transaction. The following SELECT statement uses the UPDLOCK hint to lock the selected rows:

SQL

```sql
SELECT * FROM TestSnapshotUpdate WITH (UPDLOCK)
  WHERE PriKey BETWEEN 1 AND 3
```

Using the UPDLOCK lock hint blocks any rows attempting to update the rows before the first transaction completes. This guarantees that the selected rows have no conflicts when they are updated later in the transaction. For more information, see Hints (Transact-SQL).

If your application has many conflicts, snapshot isolation may not be the best choice. Hints should only be used when really needed. Your application should not be designed so that it constantly relies on lock hints for its operation.

See also

- SQL Server and ADO.NET
- ADO.NET Overview
- Transaction Locking and Row Versioning Guide