# Transaction Locking and Row Versioning Guide

In any database, mismanagement of transactions often leads to contention and performance problems in systems that have many users.

## Transaction Basics

A transaction is a sequence of operations performed as a single logical unit of work. A logical unit of work must exhibit four properties, called the atomicity, consistency, isolation, and durability (ACID) properties, to qualify as a transaction.

### Atomicity

A transaction must be an atomic unit of work; either all of its data modifications are performed or none of them are performed.

### Consistency

When completed, a transaction must leave all data in a consistent state. In a relational database, all rules must be applied to the transaction's modifications to maintain all data integrity. All internal data structures, such as B-tree indexes or doubly linked lists, must be correct at the end of the transaction.

### Isolation

Modifications made by concurrent transactions must be isolated from the modifications made by any other concurrent transactions. A transaction either recognizes data in the state it was in before another concurrent transaction modified it, or it recognizes the data after the second transaction has completed, but it does not recognize an intermediate state.

### Durability

After a fully durable transaction has completed, its effects are permanently in place in the system. The modifications persist even in the event of a system failure.

* Locking facilities that preserve transaction isolation.
* Logging facilities ensure transaction durability.

## Controlling Transactions

### Explicit Transactions

An explicit transaction is one in which you explicitly define both the start and end of the transaction through an API function or by issuing the Transact-SQL BEGIN TRANSACTION, COMMIT TRANSACTION, COMMIT WORK, ROLLBACK TRANSACTION, or ROLLBACK WORK Transact-SQL statements. When the transaction ends, the connection returns to the transaction mode it was in before the explicit transaction was started, either implicit or autocommit mode.

### AutoCmmit Transactions

### Autocommit mode is the default transaction management mode of the SQL Server Database Engine. Every Transact-SQL statement is committed or rolled back when it completes. If a statement completes successfully, it is committed; if it encounters any error, it is rolled back.

### Implicit Transactions

When a connection is operating in implicit transaction mode, the instance of the SQL Server Database Engine automatically starts a new transaction after the current transaction is committed or rolled back. You do nothing to delineate the start of a transaction; you only commit or roll back each transaction. Implicit transaction mode generates a continuous chain of transactions. Set implicit transaction mode on through either an API function or the Transact-SQL SET IMPLICIT\_TRANSACTIONS ON statement.

### Batch-Scoped Transactions

Applicable only to multiple active result sets (MARS), a Transact-SQL explicit or implicit transaction that starts under a MARS session becomes a batch-scoped transaction. A batch-scoped transaction that is not committed or rolled back when a batch completes is automatically rolled back by SQL Server.

### Distributed Transactions

Distributed transactions span two or more servers known as resource managers. The management of the transaction must be coordinated between the resource managers by a server component called a transaction manager. Each instance of the SQL Server Database Engine can operate as a resource manager in distributed transactions coordinated by transaction managers, such as Microsoft Distributed Transaction Coordinator (MS DTC), or other transaction managers that support the Open Group XA specification for distributed transaction processing

A transaction within a single instance of the SQL Server Database Engine that spans two or more databases is actually a distributed transaction.

At the application, a distributed transaction is managed much the same as a local transaction. At the end of the transaction, the application requests the transaction to be either committed or rolled back. A distributed commit must be managed differently by the transaction manager to minimize the risk that a network failure may result in some resource managers successfully committing while others roll back the transaction. This is achieved by managing the commit process in two phases (the prepare phase and the commit phase), which is known as a two-phase commit (2PC).

#### Prepare Phase

When the transaction manager receives a commit request, it sends a prepare command to all of the resource managers involved in the transaction. Each resource manager then does everything required to make the transaction durable, and all buffers holding log images for the transaction are flushed to disk. As each resource manager completes the prepare phase, it returns success or failure of the prepare to the transaction manager.

#### Commit Phase

If the transaction manager receives successful prepares from all of the resource managers, it sends commit commands to each resource manager. The resource managers can then complete the commit. If all of the resource managers report a successful commit, the transaction manager then sends a success notification to the application. If any resource manager reported a failure to prepare, the transaction manager sends a rollback command to each resource manager and indicates the failure of the commit to the application.

### Ending Transactions

You can end transactions with either a COMMIT or ROLLBACK statement, or through a corresponding API function.

#### COMMIT

If a transaction is successful, commit it. A COMMIT statement guarantees all of the transaction's modifications are made a permanent part of the database. A COMMIT also frees resources, such as locks, used by the transaction.

#### ROLLBACK

If an error occurs in a transaction, or if the user decides to cancel the transaction, then roll the transaction back. A ROLLBACK statement backs out all modifications made in the transaction by returning the data to the state it was in at the start of the transaction. A ROLLBACK also frees resources held by the transaction.

### Errors During Transaction Processing

If an error prevents the successful completion of a transaction, SQL Server automatically rolls back the transaction and frees all resources held by the transaction. All outstanding transactions are rolled back in case of all type networks, application etc. failures.

If a run-time statement error (such as a constraint violation) occurs in a batch, the default behavior in the SQL Server Database Engine is to roll back only the statement that generated the error. You can change this behavior using the SET XACT\_ABORT statement. After SET XACT\_ABORT ON is executed, any run-time statement error causes an automatic rollback of the current transaction. Compile errors, such as syntax errors, are not affected by SET XACT\_ABORT. When errors occur, corrective action (COMMIT or ROLLBACK) should be included in application code. One effective tool for handling errors, including those in transactions, is the Transact-SQL TRY…CATCH construct.

#### Compile and Run-time Errors in Autocommit mode

In autocommit mode, it sometimes appears as if an instance of the SQL Server Database Engine has rolled back an entire batch instead of just one SQL statement. This happens if the error encountered is a compile error, not a run-time error. A compile error prevents the SQL Server Database Engine from building an execution plan, so nothing in the batch is executed. Although it appears that all of the statements before the one generating the error were rolled back, the error prevented anything in the batch from being executed.

## Locking and Row Versioning Basics

The SQL Server Database Engine uses the following mechanisms to ensure the integrity of transactions and maintain the consistency of databases when multiple users are accessing data at the same time:

* **Locking**

Each transaction requests locks of different types on the resources, such as rows, pages, or tables, on which the transaction is dependent. The locks block other transactions from modifying the resources in a way that would cause problems for the transaction requesting the lock. Each transaction frees its locks when it no longer has a dependency on the locked resources.

* **Row versioning**

When a row versioning-based isolation level is enabled, the SQL Server Database Engine maintains versions of each row that is modified. Applications can specify that a transaction use the row versions to view data as it existed at the start of the transaction or query instead of protecting all reads with locks. By using row versioning, the chance that a read operation will block other transactions is greatly reduced.

Locking and row versioning prevent users from reading uncommitted data and prevent multiple users from attempting to change the same data at the same time. Without locking or row versioning, queries executed against that data could produce unexpected results by returning data that has not yet been committed in the database.

### Managing Concurrent Data Access

Users who access a resource at the same time are said to be accessing the resource concurrently. Concurrent data access requires mechanisms to prevent adverse effects when multiple users try to modify resources that other users are actively using.

#### Concurrency Effects

If a data storage system has no concurrency control, users could see the following side effects:

* **Lost updates**

Lost updates occur when two or more transactions select the same row and then update the row based on the value originally selected. Each transaction is unaware of the other transactions. The last update overwrites updates made by the other transactions, which results in lost data.

* **Dirty read (Uncommitted dependency)**

Uncommitted dependency occurs when a second transaction selects a row that is being updated by another transaction. The second transaction is reading data that has not been committed yet and may be changed by the transaction updating the row. All isolation levels except for read uncommitted protect against dirty reads.

* **Nonrepeatable read (Inconsistent analysis)**

If a specific set of data is accessed more than once in the same transaction (such as when two different queries against the same table use the same WHERE clause) and the rows accessed between these accesses are updated or deleted by another transaction, a non-repeatable read has taken place. That is, if two queries against the same table with the same WHERE clause are executed in the same transaction, they return different results. The repeatable read, serializable, and snapshot isolation levels protect a transaction from non-repeatable reads.

* **Phantom reads**

A phantom read is a situation that occurs when two identical queries are executed and the collection of rows returned by the second query is different. The example below shows how this may occur. Assume the two transactions below are executing at the same time. The two SELECTstatements in the first transaction may return different results because the INSERT statement in the second transaction changes the data used by both.

### Types of Concurrency

When many people attempt to modify data in a database at the same time, a system of controls must be implemented so that modifications made by one person do not adversely affect those of another person. This is called concurrency control.

Concurrency control theory has two classifications for the methods of instituting concurrency control:

#### **Pessimistic Concurrency control**

A system of locks prevents users from modifying data in a way that affects other users. After a user performs an action that causes a lock to be applied, other users cannot perform actions that would conflict with the lock until the owner releases it. This is called pessimistic control because it is mainly used in environments where there is high contention for data, where the cost of protecting data with locks is less than the cost of rolling back transactions if concurrency conflicts occur.

* 1. System of locks 2.Prevent data modifications

1. No lock conflicts allowed 4.High contention environments

#### **Optimistic Concurrency control**

In optimistic concurrency control, users do not lock data when they read it. When a user updates data, the system checks to see if another user changed the data after it was read. If another user updated the data, an error is raised. Typically, the user receiving the error rolls back the transaction and starts over. This is called optimistic because it is mainly used in environments where there is low contention for data, and where the cost of occasionally rolling back a transaction is lower than the cost of locking data when read.

* Data is not locked when read
* System checks for changes
* Errors are raised
* Rolled Back
* Low Contention Environments

### Isolation Levels in the SQL Server Database Engine

Transactions specify an isolation level that defines the degree to which one transaction must be isolated from resource or data modifications made by other transactions.

Transaction isolation levels control:

* Whether locks are taken when data is read, and what type of locks are requested.
* How long the read locks are held.
* Whether a read operation referencing rows modified by another transaction:
  + Blocks until the exclusive lock on the row is freed.
  + Retrieves the committed version of the row that existed at the time the statement or transaction started.
  + Reads the uncommitted data modification.

A lower isolation level increases the ability of many users to access data at the same time, but increases the number of concurrency effects (such as dirty reads or lost updates) users might encounter. Conversely, a higher isolation level reduces the types of concurrency effects that users may encounter, but requires more system resources and increases the chances that one transaction will block another.

**SQL Server Database Engine Isolation Levels**

Isolation levels:

* READ UNCOMMITTED
* READ COMMITED
* REPEATABLE READ
* SERIALIZATION
* SNAPSHOT (Row Versioning Isolation Level)
* Read Committed Snapshot (Row Versioning Isolation Level)

#### READ UNCOMMITTED

Specifies that statements can read rows that have been modified by other transactions but not yet committed. It leads to dirty reads

#### READ COMMITED

Specifies that statements cannot read data that has been modified but not committed by other transactions. This prevents dirty reads. Data can be changed by other transactions between individual statements within the current transaction, resulting in nonrepeatable reads or phantom data. This option is the SQL Server default.

#### REPEATABLE READ

* Statements cannot read data that has been modified but not yet committed by other transactions.
* No other transactions can modify data that has been read by the current transaction until the current transaction completes.

Shared locks are placed on all data read by each statement in the transaction and are held until the transaction completes. This prevents other transactions from modifying any rows that have been read by the current transaction. Other transactions can insert new rows that match the search conditions of statements issued by the current transaction. If the current transaction then retries the statement, it will retrieve the new rows, which results in phantom reads.

#### SERIALIZABLE

* Statements cannot read data that has been modified but not yet committed by other transactions.
* No other transactions can modify data that has been read by the current transaction until the current transaction completes.
* Other transactions cannot insert new rows with key values that would fall in the range of keys read by any statements in the current transaction until the current transaction completes.

Range locks are placed in the range of key values that match the search conditions of each statement executed in a transaction. This blocks other transactions from updating or inserting any rows that would qualify for any of the statements executed by the current transaction. This means that if any of the statements in a transaction are executed a second time, they will read the same set of rows. The range locks are held until the transaction completes. This is the most restrictive of the isolation levels because it locks entire ranges of keys and holds the locks until the transaction completes. Because concurrency is lower, use this option only when necessary. This option has the same effect as setting HOLDLOCK on all tables in all SELECT statements in a transaction.

#### SNAPSHOT

The snapshot isolation level uses row versioning to provide transaction-level read consistency. Read operations acquire no page or row locks; only SCH-S table locks are acquired.

**Note:** SQL Server does not support versioning of metadata. For this reason, there are restrictions on what DDL operations can be performed in an explicit transaction that is running under snapshot isolation. The DDL statements are not permitted under snapshot isolation after a BEGIN TRANSACTION statement:

* The transaction can only recognize data modifications that were committed before the start of the transaction.
* Data modifications made by other transactions after the start of the current transaction are not visible to statements executing in the current transaction.

#### Read Committed Snapshot

When the READ\_COMMITTED\_SNAPSHOT database option is set ON, read committed isolation uses row versioning to provide statement-level read consistency. Read operations require only SCH-S table level locks and no page or row locks. Locks are not used to protect the data from updates by other transactions. A user-defined function can return data that was committed after the time the statement containing the UDF began.

| **Isolation level** | **Dirty read** | **Nonrepeatable read** | **Phantom** |
| --- | --- | --- | --- |
| **Read uncommitted** | Yes | Yes | Yes |
| **Read committed** | No | Yes | Yes |
| **Repeatable read** | No | No | Yes |
| **Snapshot** | No | No | No |
| **Serializable** | No | No | No |