## Locking in the SQL Server Database Engine

Locking is a mechanism used by the SQL Server Database Engine to synchronize access by multiple users to the same piece of data at the same time. Before a transaction acquires a dependency on the current state of a piece of data, such as by reading or modifying the data, it must protect itself from the effects of another transaction modifying the same data. The transaction does this by requesting a lock on the piece of data. Locks have different modes, such as shared or exclusive. The lock mode defines the level of dependency the transaction has on the data.

When a transaction modifies a piece of data, it holds the lock protecting the modification until the end of the transaction. How long a transaction holds the locks acquired to protect read operations depends on the transaction isolation level setting. All locks held by a transaction are released when the transaction completes (either commits or rolls back).

Applications do not typically request locks directly. Locks are managed internally by a part of the SQL Server Database Engine called the lock manager.

### Lock Granularity and Hierarchies

The SQL Server Database Engine has multigranular locking that allows different types of resources to be locked by a transaction. To minimize the cost of locking, the SQL Server Database Engine locks resources automatically at a level appropriate to the task. Locking at a smaller granularity, such as rows, increases concurrency but has a higher overhead because more locks must be held if many rows are locked. Locking at a larger granularity, such as tables, are expensive in terms of concurrency because locking an entire table restricts access to any part of the table by other transactions. However, it has a lower overhead because fewer locks are being maintained.

This group of locks at multiple levels of granularity is called a lock hierarchy.

| **Resource** | **Description** |
| --- | --- |
| **RID** | A row identifier used to lock a single row within a heap. |
| **KEY** | A row lock within an index used to protect key ranges in serializable transactions. |
| **PAGE** | An 8-kilobyte (KB) page in a database, such as data or index pages. |
| **EXTENT** | A contiguous group of eight pages, such as data or index pages. |
| **HoBT** | A heap or B-tree. A lock protecting a B-tree (index) or the heap data pages in a table that does not have a clustered index. |
| **TABLE** | The entire table, including all data and indexes. |
| **FILE** | A database file. |
| **APPLICATION** | An application-specified resource. |
| **METADATA** | Metadata locks. |
| **ALLOCATION\_UNIT** | An allocation unit. |
| **DATABASE** | The entire database. |

### Lock Modes

The SQL Server Database Engine locks resources using different lock modes that determine how the resources can be accessed by concurrent transactions.

| **Lock mode** | **Description** |
| --- | --- |
| **Shared (S)** | Used for read operations that do not change or update data, such as a SELECT statement. |
| **Update (U)** | Used on resources that can be updated. Prevents a common form of deadlock that occurs when multiple sessions are reading, locking, and potentially updating resources later. |
| **Exclusive (X)** | Used for data-modification operations, such as INSERT, UPDATE, DELETE Ensures that multiple updates cannot be made to the same resource at the same time. |
| **Intent** | Used to establish a lock hierarchy. The types of intent locks are: intent shared (IS), intent exclusive (IX), and shared with intent exclusive (SIX). |
| **Schema** | Used when an operation dependent on the schema of a table is executing. The types of schema locks are: schema modification (Sch-M) and schema stability (Sch-S). |
| **Bulk Update (BU)** | Used when bulk copying data into a table and the TABLOCK hint is specified. |
| **Key-range** | Protects the range of rows read by a query when using the serializable transaction isolation level. Ensures that other transactions cannot insert rows that would qualify for the queries of the serializable transaction if the queries were run again. |

###### **Shared Locks**

Shared (S) locks allow concurrent transactions to read (SELECT) a resource under pessimistic concurrency control. No other transactions can modify the data while shared (S) locks exist on the resource. Shared (S) locks on a resource are released as soon as the read operation completes, unless the transaction isolation level is set to repeatable read or higher, or a locking hint is used to retain the shared (S) locks for the duration of the transaction.

###### **Update Locks**

Update (U) locks prevent a common form of deadlock. In a repeatable read or serializable transaction, the transaction reads data, acquiring a shared (S) lock on the resource (page or row), and then modifies the data, which requires lock conversion to an exclusive (X) lock. If two transactions acquire shared-mode locks on a resource and then attempt to update data concurrently, one transaction attempts the lock conversion to an exclusive (X) lock. The shared-mode-to-exclusive lock conversion must wait because the exclusive lock for one transaction is not compatible with the shared-mode lock of the other transaction; a lock wait occurs. The second transaction attempts to acquire an exclusive (X) lock for its update. Because both transactions are converting to exclusive (X) locks, and they are each waiting for the other transaction to release its shared-mode lock, a deadlock occurs.

To avoid this potential deadlock problem, update (U) locks are used. Only one transaction can obtain an update (U) lock to a resource at a time. If a transaction modifies a resource, the update (U) lock is converted to an exclusive (X) lock.

###### **Exclusive Locks**

Exclusive (X) locks prevent access to a resource by concurrent transactions. With an exclusive (X) lock, no other transactions can modify data; read operations can take place only with the use of the NOLOCK hint or read uncommitted isolation level.

###### **Intent Locks**

The SQL Server Database Engine uses intent locks to protect placing a shared (S) lock or exclusive (X) lock on a resource lower in the lock hierarchy. Intent locks are named intent locks because they are acquired before a lock at the lower level, and therefore signal intent to place locks at a lower level.

Intent locks serve two purposes:

* To prevent other transactions from modifying the higher-level resource in a way that would invalidate the lock at the lower level.
* To improve the efficiency of the SQL Server Database Engine in detecting lock conflicts at the higher level of granularity.

For example, a shared intent lock is requested at the table level before shared (S) locks are requested on pages or rows within that table. Setting an intent lock at the table level prevents another transaction from subsequently acquiring an exclusive (X) lock on the table containing that page. Intent locks improve performance because the SQL Server Database Engine examines intent locks only at the table level to determine if a transaction can safely acquire a lock on that table. This removes the requirement to examine every row or page lock on the table to determine if a transaction can lock the entire table.

Intent locks include intent shared (IS), intent exclusive (IX), and shared with intent exclusive (SIX).

| **Lock mode** | **Description** |
| --- | --- |
| **Intent shared (IS)** | Protects requested or acquired shared locks on some (but not all) resources lower in the hierarchy. |
| **Intent exclusive (IX)** | Protects requested or acquired exclusive locks on some (but not all) resources lower in the hierarchy. IX is a superset of IS, and it also protects requesting shared locks on lower level resources. |
| **Shared with intent exclusive (SIX)** | Protects requested or acquired shared locks on all resources lower in the hierarchy and intent exclusive locks on some (but not all) of the lower level resources. Concurrent IS locks at the top-level resource are allowed. For example, acquiring a SIX lock on a table also acquires intent exclusive locks on the pages being modified and exclusive locks on the modified rows. There can be only one SIX lock per resource at one time, preventing updates to the resource made by other transactions, although other transactions can read resources lower in the hierarchy by obtaining IS locks at the table level. |
| **Intent update (IU)** | Protects requested or acquired update locks on all resources lower in the hierarchy. IU locks are used only on page resources. IU locks are converted to IX locks if an update operation takes place. |
| **Shared intent update (SIU)** | A combination of S and IU locks, as a result of acquiring these locks separately and simultaneously holding both locks. For example, a transaction executes a query with the PAGLOCK hint and then executes an update operation. The query with the PAGLOCK hint acquires the S lock, and the update operation acquires the IU lock. |
| **Update intent exclusive (UIX)** | A combination of U and IX locks, as a result of acquiring these locks separately and simultaneously holding both locks. |

###### **Schema Locks**

The SQL Server Database Engine uses schema modification (Sch-M) locks during a table data definition language (DDL) operation, such as adding a column or dropping a table. During the time that it is held, the Sch-M lock prevents concurrent access to the table. This means the Sch-M lock blocks all outside operations until the lock is released.

Some data manipulation language (DML) operations, such as table truncation, use Sch-M locks to prevent access to affected tables by concurrent operations.

The SQL Server Database Engine uses schema stability (Sch-S) locks when compiling and executing queries. Sch-S locks do not block any transactional locks, including exclusive (X) locks. Therefore, other transactions, including those with X locks on a table, continue to run while a query is being compiled.

###### **Bulk Update Locks**

Bulk update (BU) locks allow multiple threads to bulk load data concurrently into the same table while preventing other processes that are not bulk loading data from accessing the table. The SQL Server Database Engine uses bulk update (BU) locks when both of the following conditions are true.

* You use the Transact-SQL BULK INSERT statement, or the OPENROWSET(BULK) function, or you use one of the Bulk Insert API commands such as .NET SqlBulkCopy, OLEDB Fast Load APIs, or the ODBC Bulk Copy APIs to bulk copy data into a table.
* The **TABLOCK** hint is specified or the **table lock on bulk load** table option is set using **sp\_tableoption**.

Tip

Unlike the BULK INSERT statement, which holds a less restrictive Bulk Update lock, INSERT INTO…SELECT with the TABLOCK hint holds an exclusive (X) lock on the table. This means that you cannot insert rows using parallel insert operations.

###### **Key-Range Locks**

Key-range locks protect a range of rows implicitly included in a record set being read by a Transact-SQL statement while using the serializable transaction isolation level. Key-range locking prevents phantom reads. By protecting the ranges of keys between rows, it also prevents phantom insertions or deletions into a record set accessed by a transaction.

##### **Lock Compatibility**

Lock compatibility controls whether multiple transactions can acquire locks on the same resource at the same time. If a resource is already locked by another transaction, a new lock request can be granted only if the mode of the requested lock is compatible with the mode of the existing lock. If the mode of the requested lock is not compatible with the existing lock, the transaction requesting the new lock waits for the existing lock to be released or for the lock timeout interval to expire. For example, no lock modes are compatible with exclusive locks. While an exclusive (X) lock is held, no other transaction can acquire a lock of any kind (shared, update, or exclusive) on that resource until the exclusive (X) lock is released. Alternatively, if a shared (S) lock has been applied to a resource, other transactions can also acquire a shared lock or an update (U) lock on that item even if the first transaction has not completed. However, other transactions cannot acquire an exclusive lock until the shared lock has been released.

The following table shows the compatibility of the most commonly encountered lock modes.

|  | **Existing granted mode** |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Requested mode** | **IS** | **S** | **U** | **IX** | **SIX** | **X** |
| **Intent shared (IS)** | Yes | Yes | Yes | Yes | Yes | No |
| **Shared (S)** | Yes | Yes | Yes | No | No | No |
| **Update (U)** | Yes | Yes | No | No | No | No |
| **Intent exclusive (IX)** | Yes | No | No | Yes | No | No |
| **Shared with intent exclusive (SIX)** | Yes | No | No | No | No | No |
| **Exclusive (X)** | No | No | No | No | No | No |

Note

An intent exclusive (IX) lock is compatible with an IX lock mode because IX means the intention is to update only some of the rows rather than all of them. Other transactions that attempt to read or update some of the rows are also permitted as long as they are not the same rows being updated by other transactions. Further, if two transactions attempt to update the same row, both transactions will be granted an IX lock at table and page level. However, one transaction will be granted an X lock at row level. The other transaction must wait until the row-level lock is removed.

Use the following table to determine the compatibility of all the lock modes available in SQL Server.

### Dynamic Locking

Using low-level locks, such as row locks, increases concurrency by decreasing the probability that two transactions will request locks on the same piece of data at the same time. Using low-level locks also increases the number of locks and the resources needed to manage them. Using high-level table or page locks lowers overhead, but at the expense of lowering concurrency.

The SQL Server Database Engine uses a dynamic locking strategy to determine the most cost-effective locks. The SQL Server Database Engine automatically determines what locks are most appropriate when the query is executed, based on the characteristics of the schema and query. For example, to reduce the overhead of locking, the optimizer may choose page-level locks in an index when performing an index scan.

Dynamic locking has the following advantages:

* Simplified database administration. Database administrators do not have to adjust lock escalation thresholds.
* Increased performance. The SQL Server Database Engine minimizes system overhead by using locks appropriate to the task.
* Application developers can concentrate on development. The SQL Server Database Engine adjusts locking automatically.

### Deadlocking

A deadlock occurs when two or more tasks permanently block each other by each task having a lock on a resource which the other tasks are trying to lock. For example:

Transaction A acquires a share lock on row 1. Transaction B acquires a share lock on row 2. Transaction A now requests an exclusive lock on row 2, and is blocked until transaction B finishes and releases the share lock it has on row 2. Transaction B now requests an exclusive lock on row 1, and is blocked until transaction A finishes and releases the share lock it has on row 1.

Transaction A cannot complete until transaction B completes, but transaction B is blocked by transaction A. This condition is also called a cyclic dependency: Transaction A has a dependency on transaction B, and transaction B closes the circle by having a dependency on transaction A.

Both transactions in a deadlock will wait forever unless the deadlock is broken by an external process. The SQL Server Database Engine deadlock monitor periodically checks for tasks that are in a deadlock. If the monitor detects a cyclic dependency, it chooses one of the tasks as a victim and terminates its transaction with an error. This allows the other task to complete its transaction. The application with the transaction that terminated with an error can retry the transaction, which usually completes after the other deadlocked transaction has finished.

Deadlocking is often confused with normal blocking. When a transaction requests a lock on a resource locked by another transaction, the requesting transaction waits until the lock is released. **By default, SQL Server transactions do not time out, unless LOCK\_TIMEOUT is set.** The requesting transaction is blocked, not deadlocked, because the requesting transaction has not done anything to block the transaction owning the lock. Eventually, the owning transaction will complete and release the lock, and then the requesting transaction will be granted the lock and proceed.

Deadlocks are sometimes called a deadly embrace.

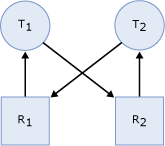
Deadlock is a condition that can occur on any system with multiple threads, not just on a relational database management system, and can occur for resources other than locks on database objects. For example, a thread in a multithreaded operating system might acquire one or more resources, such as blocks of memory. If the resource being acquired is currently owned by another thread, the first thread may have to wait for the owning thread to release the target resource. The waiting thread is said to have a dependency on the owning thread for that particular resource. In an instance of the SQL Server Database Engine, sessions can deadlock when acquiring nondatabase resources, such as memory or threads.

Deadlocks can also occur when a table is partitioned and the LOCK\_ESCALATION setting of ALTER TABLE is set to AUTO. When LOCK\_ESCALATIONis set to AUTO, concurrency increases by allowing the SQL Server Database Engine to lock table partitions at the HoBT level instead of at the table level. However, when separate transactions hold partition locks in a table and want a lock somewhere on the other transactions partition, this causes a deadlock. This type of deadlock can be avoided by setting LOCK\_ESCALATION to TABLE; although this setting will reduce concurrency by forcing large updates to a partition to wait for a table lock.

###### **Detecting and Ending Deadlocks**

A deadlock occurs when two or more tasks permanently block each other by each task having a lock on a resource which the other tasks are trying to lock.

* Task T1 has a lock on resource R1 (indicated by the arrow from R1 to T1) and has requested a lock on resource R2 (indicated by the arrow from T1 to R2).
* Task T2 has a lock on resource R2 (indicated by the arrow from R2 to T2) and has requested a lock on resource R1 (indicated by the arrow from T2 to R1).
* Because neither task can continue until a resource is available and neither resource can be released until a task continues, a deadlock state exists.



**Resources that can Deadlock**

Each user session might have one or more tasks running on its behalf where each task might acquire or wait to acquire a variety of resources. The following types of resources can cause blocking that could result in a deadlock.

* **Locks**. Waiting to acquire locks on resources, such as objects, pages, rows, metadata, and applications can cause deadlock. For example, transaction T1 has a shared (S) lock on row r1 and is waiting to get an exclusive (X) lock on r2. Transaction T2 has a shared (S) lock on r2 and is waiting to get an exclusive (X) lock on row r1. This results in a lock cycle in which T1 and T2 wait for each other to release the locked resources.
* **Worker threads**. A queued task waiting for an available worker thread can cause deadlock. If the queued task owns resources that are blocking all worker threads, a deadlock will result. For example, session S1 starts a transaction and acquires a shared (S) lock on row r1 and then goes to sleep. Active sessions running on all available worker threads are trying to acquire exclusive (X) locks on row r1. Because session S1 cannot acquire a worker thread, it cannot commit the transaction and release the lock on row r1. This results in a deadlock.
* **Memory**. When concurrent requests are waiting for memory grants that cannot be satisfied with the available memory, a deadlock can occur. For example, two concurrent queries, Q1 and Q2, execute as user-defined functions that acquire 10MB and 20MB of memory respectively. If each query needs 30MB and the total available memory is 20MB, then Q1 and Q2 must wait for each other to release memory, and this results in a deadlock.
* **Parallel query execution-related resources** Coordinator, producer, or consumer threads associated with an exchange port may block each other causing a deadlock usually when including at least one other process that is not a part of the parallel query. Also, when a parallel query starts execution, SQL Server determines the degree of parallelism, or the number of worker threads, based upon the current workload. If the system workload unexpectedly changes, for example, where new queries start running on the server or the system runs out of worker threads, then a deadlock could occur.
* **Multiple Active Result Sets (MARS) resources**. These resources are used to control interleaving of multiple active requests under MARS. For more information, see [Using Multiple Active Result Sets (MARS)](https://docs.microsoft.com/en-us/sql/relational-databases/native-client/features/using-multiple-active-result-sets-mars?view=sql-server-2017).
  + **User resource**. When a thread is waiting for a resource that is potentially controlled by a user application, the resource is considered an external or user resource and is treated like a lock.
  + **Session mutex**. The tasks running in one session are interleaved, meaning that only one task can run under the session at a given time. Before the task can run, it must have exclusive access to the session mutex.
  + **Transaction mutex**. All tasks running in one transaction are interleaved, meaning that only one task can run under the transaction at a given time. Before the task can run, it must have exclusive access to the transaction mutex.

**Deadlock Detection**

All of the resources listed in the section above participate in the SQL Server Database Engine deadlock detection scheme. Deadlock detection is performed by a lock monitor thread that periodically initiates a search through all of the tasks in an instance of the SQL Server Database Engine. The following points describe the search process:

* The default interval is 5 seconds.
* If the lock monitor thread finds deadlocks, the deadlock detection interval will drop from 5 seconds to as low as 100 milliseconds depending on the frequency of deadlocks.
* If the lock monitor thread stops finding deadlocks, the SQL Server Database Engine increases the intervals between searches to 5 seconds.

The SQL Server Database Engine typically performs periodic deadlock detection only. Because the number of deadlocks encountered in the system is usually small, periodic deadlock detection helps to reduce the overhead of deadlock detection in the system.

When the lock monitor initiates deadlock search for a particular thread, it identifies the resource on which the thread is waiting. The lock monitor then finds the owner(s) for that particular resource and recursively continues the deadlock search for those threads until it finds a cycle. A cycle identified in this manner forms a deadlock.

After a deadlock is detected, the SQL Server Database Engine ends a deadlock by choosing one of the threads as a deadlock victim. The SQL Server Database Engine terminates the current batch being executed for the thread, rolls back the transaction of the deadlock victim, and returns a 1205 error to the application. Rolling back the transaction for the deadlock victim releases all locks held by the transaction. This allows the transactions of the other threads to become unblocked and continue. The 1205 deadlock victim error records information about the threads and resources involved in a deadlock in the error log.

By default, the SQL Server Database Engine chooses as the deadlock victim the session running the transaction that is least expensive to roll back. Alternatively, a user can specify the priority of sessions in a deadlock situation using the SET DEADLOCK\_PRIORITY statement. DEADLOCK\_PRIORITY can be set to LOW, NORMAL, or HIGH, or alternatively can be set to any integer value in the range (-10 to 10). The deadlock priority defaults to NORMAL. If two sessions have different deadlock priorities, the session with the lower priority is chosen as the deadlock victim. If both sessions have the same deadlock priority, the session with the transaction that is least expensive to roll back is chosen. If sessions involved in the deadlock cycle have the same deadlock priority and the same cost, a victim is chosen randomly.

**Deadlock Information Tools**

To view deadlock information, the SQL Server Database Engine provides monitoring tools in the form of the the system\_health xEvent session, two trace flags, and the deadlock graph event in SQL Profiler.

**Deadlock in system\_health session**

* **victim-list**. The deadlock victim process identifier.
* **process-list**. Information on all the processes involved in the deadlock.
* **resource-list**. Information about the resources involved in the deadlock.

Opening the system\_health session file or ring buffer, if the xml\_deadlock\_report xEvent is recorded, Management Studio presents a graphical depiction of the tasks and resources involved in a deadlock:

SELECT xdr.value('@timestamp', 'datetime') AS [Date],

xdr.query('.') AS [Event\_Data]

FROM (SELECT CAST([target\_data] AS XML) AS Target\_Data

FROM sys.dm\_xe\_session\_targets AS xt

INNER JOIN sys.dm\_xe\_sessions AS xs ON xs.address = xt.event\_session\_address

WHERE xs.name = N'system\_health'

AND xt.target\_name = N'ring\_buffer'

) AS XML\_Data

CROSS APPLY Target\_Data.nodes('RingBufferTarget/event[@name="xml\_deadlock\_report"]') AS XEventData(xdr)

ORDER BY [Date] DESC

###### **Minimizing Deadlocks**

Although deadlocks cannot be completely avoided, following certain coding conventions can minimize the chance of generating a deadlock. To help minimize deadlocks

* Access objects in the same order.
* Avoid user interaction in transactions.
* Keep transactions short and in one batch.
* Use a lower isolation level.
* Use a row versioning-based isolation level.
  + Set READ\_COMMITTED\_SNAPSHOT database option ON to enable read-committed transactions to use row versioning.
  + Use snapshot isolation.
* Use bound connections

**Use a lower Isolation Level**

Determine whether a transaction can run at a lower isolation level. Implementing read committed allows a transaction to read data previously read (not modified) by another transaction without waiting for the first transaction to complete. Using a lower isolation level, such as read committed, holds shared locks for a shorter duration than a higher isolation level, such as serializable. This reduces locking contention.

**Use a Row Versioning-based Isolation Level**

When the READ\_COMMITTED\_SNAPSHOT database option is set ON, a transaction running under read committed isolation level uses row versioning rather than shared locks during read operations.

Snapshot isolation also uses row versioning, which does not use shared locks during read operations. Before a transaction can run under snapshot isolation, the ALLOW\_SNAPSHOT\_ISOLATION database option must be set ON. Implement these isolation levels to minimize deadlocks that can occur between read and write operations.

**Use bound connections**

Using bound connections, two or more connections opened by the same application can cooperate with each other. Any locks acquired by the secondary connections are held as if they were acquired by the primary connection, and vice versa. Therefore they do not block each other.

##### **Lock Partitioning**

For large computer systems, locks on frequently referenced objects can become a performance bottleneck as acquiring and releasing locks place contention on internal locking resources. Lock partitioning enhances locking performance by splitting a single lock resource into multiple lock resources. This feature is only available for systems with 16 or more CPUs, and is automatically enabled and cannot be disabled. [sys.dm\_tran\_locks (Transact-SQL)](https://docs.microsoft.com/en-us/sql/relational-databases/system-dynamic-management-views/sys-dm-tran-locks-transact-sql?view=sql-server-2017).

###### **Understanding Lock Partitioning**

Locking tasks access several shared resources, two of which are optimized by lock partitioning:

* **Spinlock**. This controls access to a lock resource, such as a row or a table.

Without lock partitioning, one spinlock manages all lock requests for a single lock resource. On systems that experience a large volume of activity, contention can occur as lock requests wait for the spinlock to become available. Under this situation, acquiring locks can become a bottleneck and can negatively impact performance.

To reduce contention on a single lock resource, lock partitioning splits a single lock resource into multiple lock resources to distribute the load across multiple spinlocks.

* **Memory**. This is used to store the lock resource structures.

Once the spinlock is acquired, lock structures are stored in memory and then accessed and possibly modified. Distributing lock access across multiple resources helps to eliminate the need to transfer memory blocks between CPUs, which will help to improve performance.

#### Row Versioning-based Isolation Levels in the SQL Server Database Engine

Row versioning is a general framework in SQL Server that invokes a copy-on-write mechanism when a row is modified or deleted. This requires that while the transaction is running, the old version of the row must be available for transactions. Row versioning is used to do the following:

* Build the **inserted** and **deleted** tables in triggers. Any rows modified by the trigger are versioned. This includes the rows modified by the statement that launched the trigger, as well as any data modifications made by the trigger.
* Support Multiple Active Result Sets (MARS). If a MARS session issues a data modification statement (such as INSERT, UPDATE, or DELETE) at a time there is an active result set, the rows affected by the modification statement are versioned.
* Support index operations that specify the ONLINE option.
* Support row versioning-based transaction isolation levels:
  + A new implementation of read committed isolation level that uses row versioning to provide statement-level read consistency.
  + A new isolation level, snapshot, to provide transaction-level read consistency.

The tempdb database must have enough space for the version store. When tempdb is full, update operations will stop generating versions and continue to succeed, but read operations might fail because a particular row version that is needed no longer exists. This affects operations like triggers, MARS, and online indexing.

Using row versioning for read-committed and snapshot transactions is a two-step process:

1. Set either or both the READ\_COMMITTED\_SNAPSHOT and ALLOW\_SNAPSHOT\_ISOLATION database options ON.
2. Set the appropriate transaction isolation level in an application:
   * When the READ\_COMMITTED\_SNAPSHOT database option is ON, transactions setting the read committed isolation level use row versioning.
   * When the ALLOW\_SNAPSHOT\_ISOLATION database option is ON, transactions can set the snapshot isolation level.

When either READ\_COMMITTED\_SNAPSHOT or ALLOW\_SNAPSHOT\_ISOLATION database option is set ON, the SQL Server Database Engine assigns a transaction sequence number (XSN) to each transaction that manipulates data using row versioning. Transactions start at the time a BEGIN TRANSACTION statement is executed. However, the transaction sequence number starts with the first read or write operation after the BEGIN TRANSACTION statement. The transaction sequence number is incremented by one each time it is assigned.

When either the READ\_COMMITTED\_SNAPSHOT or ALLOW\_SNAPSHOT\_ISOLATION database options are ON, logical copies (versions) are maintained for all data modifications performed in the database. Every time a row is modified by a specific transaction, the instance of the SQL Server Database Engine stores a version of the previously committed image of the row in tempdb. Each version is marked with the transaction sequence number of the transaction that made the change. The versions of modified rows are chained using a link list. The newest row value is always stored in the current database and chained to the versioned rows stored in tempdb.

### Using Row Versioning-based Isolation Levels

The row versioning framework is always enabled in SQL Server, and is used by multiple features. Besides providing row versioning-based isolation levels, it is used to support modifications made in triggers and multiple active result sets (MARS) sessions, and to support data reads for ONLINE index operations.

Row versioning-based isolation levels are enabled at the database level.

Read-committed that uses row versioning by setting the **READ\_COMMITTED\_SNAPSHOT** database option to **ON**

ALTER DATABASE AdventureWorks2016 SET READ\_COMMITTED\_SNAPSHOT ON;

When the database is enabled for READ\_COMMITTED\_SNAPSHOT, all queries running under the read committed isolation level use row versioning, which means that read operations do not block update operations.

Snapshot isolation by setting the ALLOW\_SNAPSHOT\_ISOLATION database option to ON

ALTER DATABASE AdventureWorks2016 SET ALLOW\_SNAPSHOT\_ISOLATION ON;

A transaction running under snapshot isolation can access tables in the database that have been enabled for snapshot. To access tables that have not been enabled for snapshot, the isolation level must be changed.

###### **Limitations of Transactions Using Row Versioning-based Isolation Levels**

Consider the following **limitations** when working with row versioning-based isolation levels:

* READ\_COMMITTED\_SNAPSHOT cannot be enabled in tempdb, msdb, or master.
* Global temp tables are stored in tempdb. When accessing global temp tables inside a snapshot transaction, one of the following must happen:
  + Set the ALLOW\_SNAPSHOT\_ISOLATION database option ON in tempdb.
  + Use an isolation hint to change the isolation level for the statement.
* Snapshot transactions fail when:
  + A database is made read-only after the snapshot transaction starts, but before the snapshot transaction accesses the database.
  + If accessing objects from multiple databases, a database state was changed in such a way that database recovery occurred after a snapshot transaction starts, but before the snapshot transaction accesses the database. For example: the database was set to OFFLINE and then to ONLINE, database autoclose and open, or database detach and attach.
* Distributed transactions, including queries in distributed partitioned databases, are not supported under snapshot isolation.
* SQL Server does not keep multiple versions of system metadata. Data definition language (DDL) statements on tables and other database objects (indexes, views, data types, stored procedures, and common language runtime functions) change metadata. If a DDL statement modifies an object, any concurrent reference to the object under snapshot isolation causes the snapshot transaction to fail. Read-committed transactions do not have this limitation when the READ\_COMMITTED\_SNAPSHOT database option is ON.

### Lock Escalation:

Lock escalation is the process of converting many fine-grain locks into fewer coarse-grain locks, reducing system overhead while increasing the probability of concurrency contention.

When the Database Engine checks for possible escalations at every 1250 newly acquired locks, a lock escalation will occur if and only if a Transact-SQL statement has acquired at least 5000 locks on a single reference of a table. Lock escalation is triggered when a Transact-SQL statement acquires at least 5,000 locks on a single reference of a table. For example, lock escalation is not triggered if a statement acquires 3,000 locks in one index and 3,000 locks in another index of the same table. Similarly, lock escalation is not triggered if a statement has a self-join on a table, and each reference to the table only acquires 3,000 locks in the table.

Lock escalation only occurs for tables that have been accessed at the time the escalation is triggered. Assume that a single SELECT statement is a join that accesses three tables in this sequence: **TableA**, **TableB**, and **TableC**. The statement acquires 3,000 row locks in the clustered index for **TableA** and at least 5,000 row locks in the clustered index for **TableB**, but has not yet accessed **TableC**. When the Database Engine detects that the statement has acquired at least 5,000 row locks in **TableB**, it attempts to escalate all locks held by the current transaction on **TableB**. It also attempts to escalate all locks held by the current transaction on **TableA**, but since the number of locks on **TableA** is < 5000, the escalation will not succeed. No lock escalation is attempted for **TableC** because it had not yet been accessed when the escalation occurred.

**Escalation Threshold for an Instance of the Database Engine**

Whenever the number of locks is greater than the memory threshold for lock escalation, the Database Engine triggers lock escalation. The memory threshold depends on the setting of the **locks** configuration option:

* If the **locks** option is set to its default setting of 0, then the lock escalation threshold is reached when the memory used by lock objects is 24 percent of the memory used by the Database Engine, excluding AWE memory. The data structure used to represent a lock is approximately 100 bytes long. This threshold is dynamic because the Database Engine dynamically acquires and frees memory to adjust for varying workloads.
* If the **locks** option is a value other than 0, then the lock escalation threshold is 40 percent (or less if there is a memory pressure) of the value of the locks option.

### Advanced Transaction Information

#### Nesting Transactions

Explicit transactions can be nested. This is primarily intended to support transactions in stored procedures that can be called either from a process already in a transaction or from processes that have no active transaction.

The transaction is either committed or rolled back based on the action taken at the end of the outermost transaction. If the outer transaction is committed, the inner nested transactions are also committed. If the outer transaction is rolled back, then all inner transactions are also rolled back, regardless of whether or not the inner transactions were individually committed.

#### Coding Guidelines

**These are guidelines for coding efficient transactions:**

* Do not require input from users during a transaction

Get all required input from users before a transaction is started. If additional user input is required during a transaction, roll back the current transaction and restart the transaction after the user input is supplied.

* Do not open a transaction while browsing through data, if at all possible.

Transactions should not be started until all preliminary data analysis has been completed.

* Keep the transaction as short as possible.
* To reduce blocking, consider using a row versioning-based isolation level for read-only queries.
* Make intelligent use of lower transaction isolation levels.
* Make intelligent use of lower cursor concurrency options, such as optimistic concurrency options.In a system with a low probability of concurrent updates, the overhead of dealing with an occasional "somebody else changed your data after you read it" error can be much lower than the overhead of always locking rows as they are read.
* Access the least amount of data possible while in a transaction.
* This lessens the number of locked rows, thereby reducing contention between transactions.

### Managing long-running transactions

A long-running transaction is an active transaction that has not been committed or roll backed the transaction in a timely manner. For example, if the beginning and end of a transaction is controlled by the user, a typical cause of a long-running transaction is a user starting a transaction and then leaving while the transaction waits for a response from the user.

A long running transaction can cause serious problems for a database, as follows:

* If a server instance is shut down after an active transaction has performed many uncommitted modifications, the recovery phase of the subsequent restart can take much longer than the time specified by the **recovery interval** server configuration option or by the ALTER DATABASE … SET TARGET\_RECOVERY\_TIMEoption. These options control the frequency of active and indirect checkpoints, respectively.
* More importantly, although a waiting transaction might generate very little log, it holds up log truncation indefinitely, causing the transaction log to grow and possibly fill up. If the transaction log fills up, the database cannot perform any more updates.

#### Discovering long-running transactions

To look for long-running transactions, use one of the following:

* **sys.dm\_tran\_database\_transactions**

This dynamic management view returns information about transactions at the database level. For a long-running transaction, columns of particular interest include the time of the first log record (**database\_transaction\_begin\_time**), the current state of the transaction (**database\_transaction\_state**), and the log sequence number (LSN) of the begin record in the transaction log (**database\_transaction\_begin\_lsn**).

* DBCC OPENTRAN

This statement lets you identify the user ID of the owner of the transaction, so you can potentially track down the source of the transaction for a more orderly termination (committing it rather than rolling it back).

#### Stopping a Transaction

You may have to use the KILL statement. Use this statement very carefully, however, especially when critical processes are running.