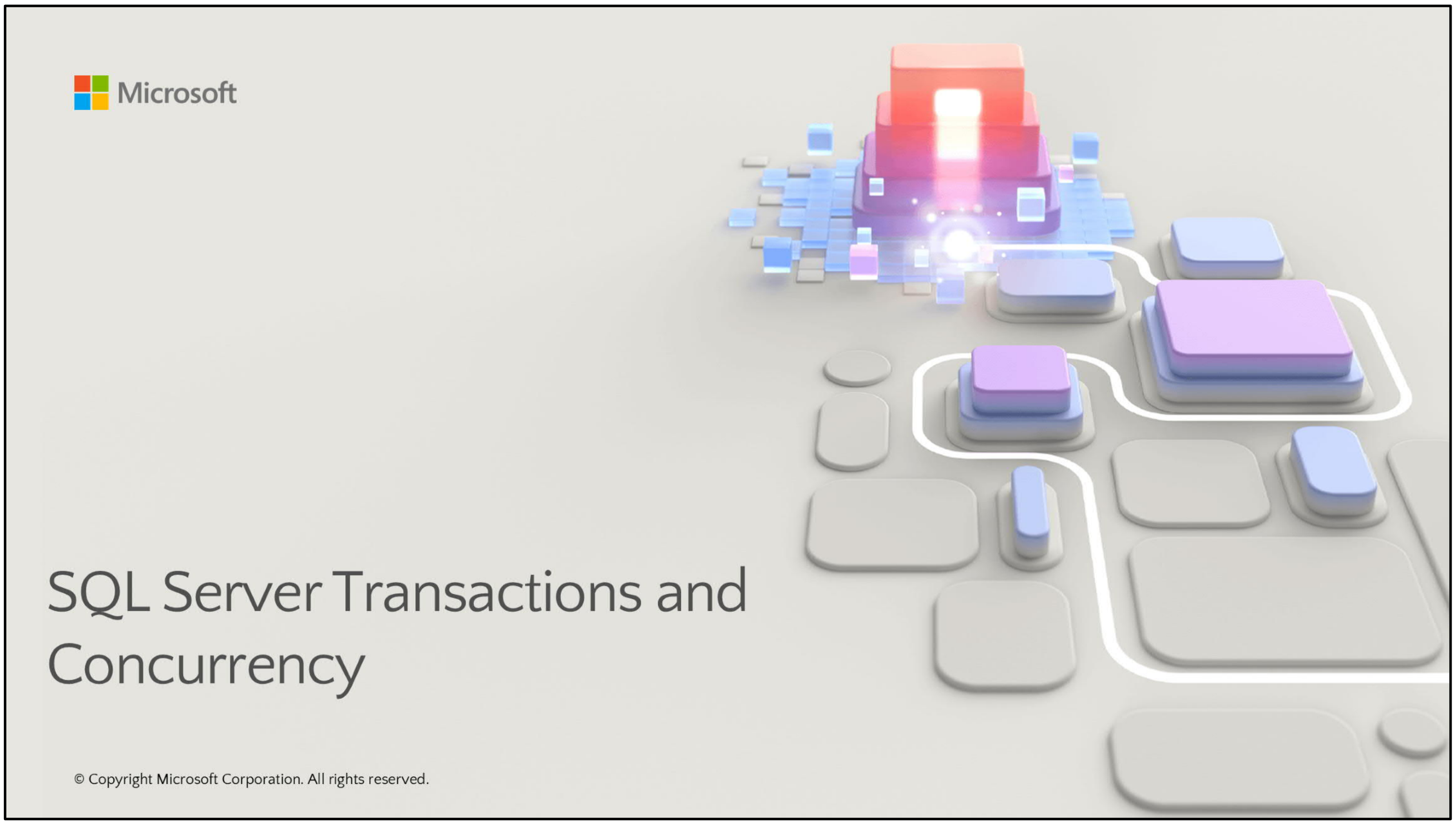


Microsoft

SQL Server Transactions and Concurrency

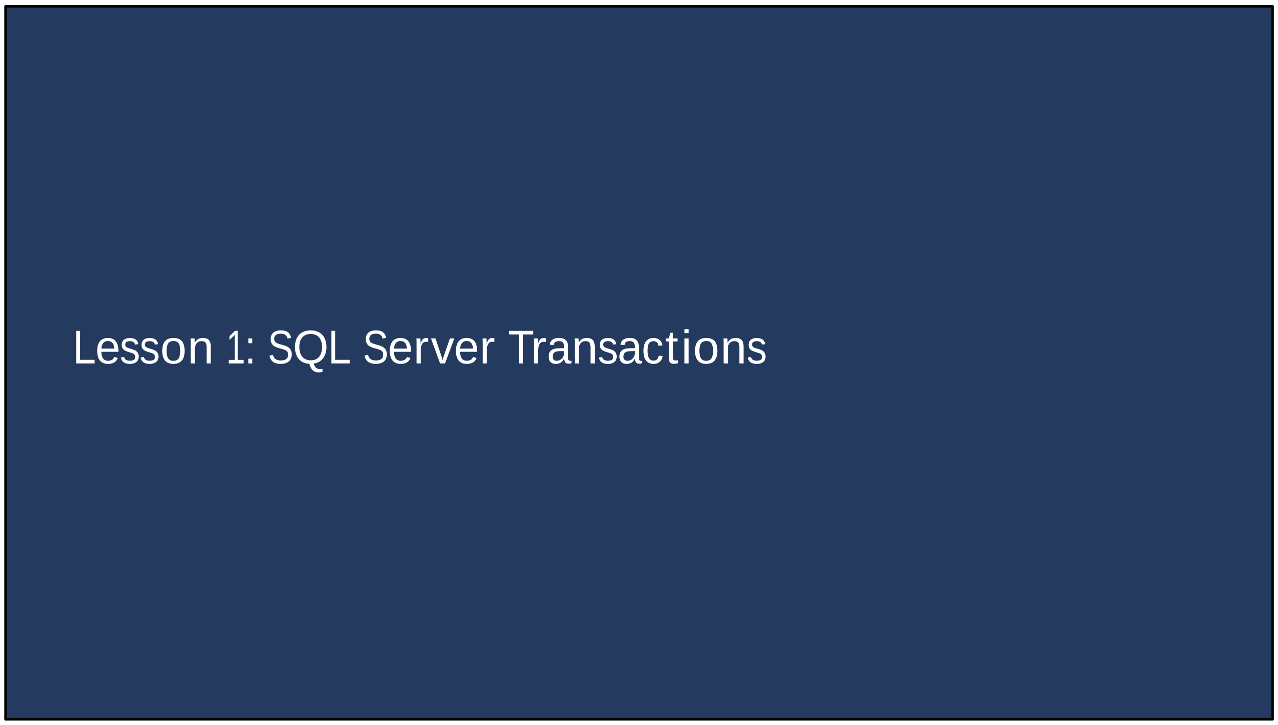
Learning Units covered in this Module

- Lesson 1: SQL Server Transactions
- Lesson 2: SQL Server Isolation Levels
- Lesson 3: SQL Server Locking
- Lesson 4: Troubleshooting Concurrency Performance

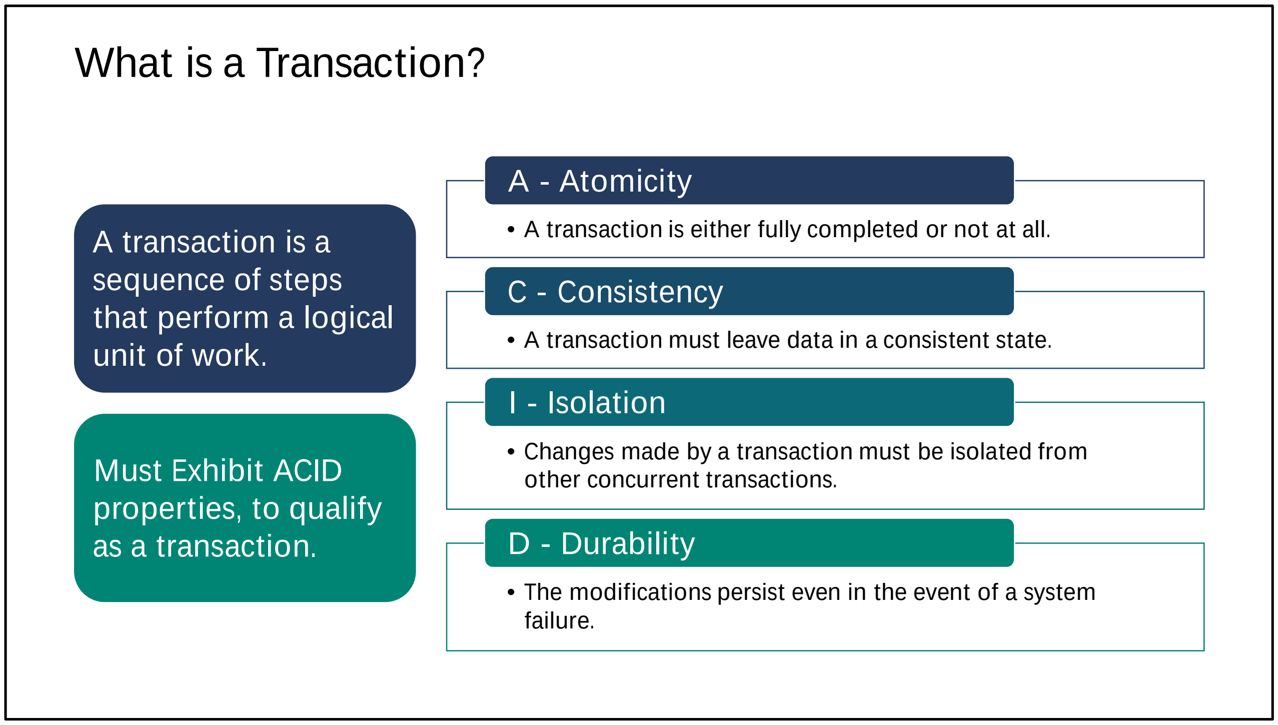
Objectives

After completing this learning, you will be able to:

- Understand what is a transaction and its types.
- Recommendations for using transactions.
- Monitor current transactions in SQL Server.
- Describe the concurrency and concurrency types



Lesson 1: SQL Server Transactions



What is a Transaction?

A transaction is a sequence of steps that perform a logical unit of work.

Must Exhibit ACID properties, to qualify as a transaction.

A - Atomicity
- A transaction is either fully completed or not at all.

C - Consistency
- A transaction must leave data in a consistent state.

I - Isolation
- Changes made by a transaction must be isolated from other concurrent transactions.

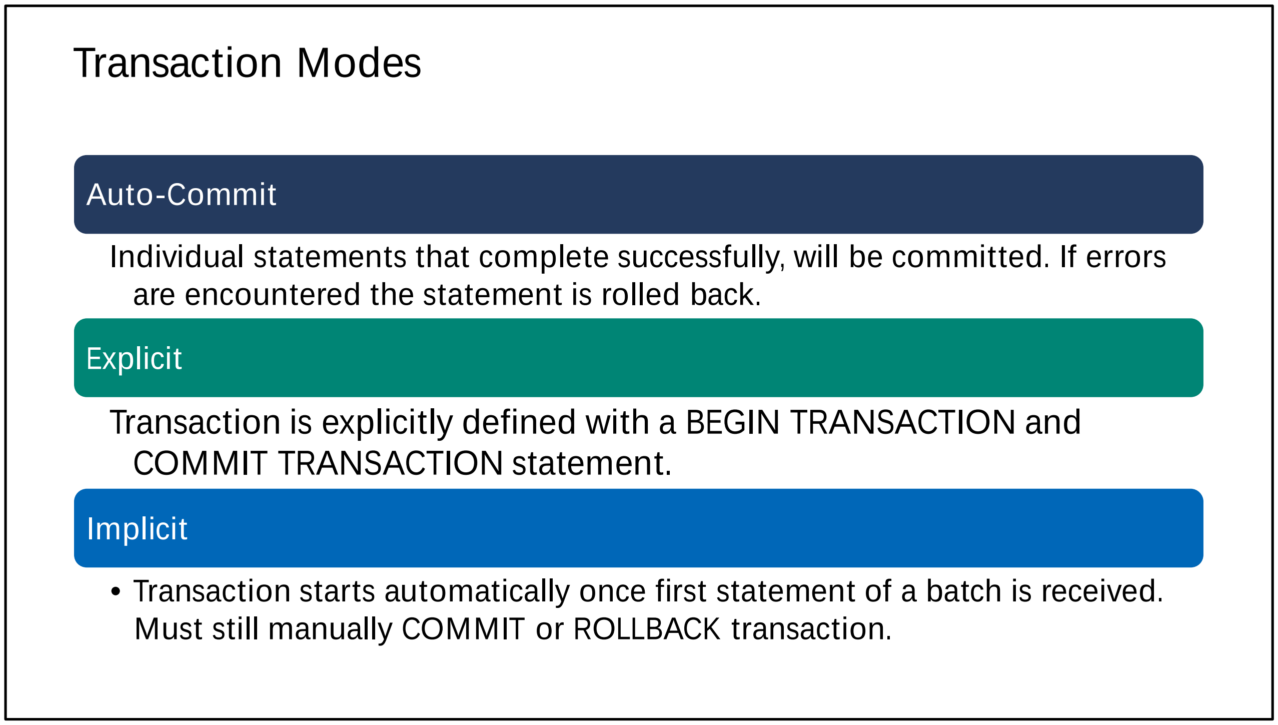
D - Durability
- The modifications persist even in the event of a system failure.

A transaction is a sequence of steps that perform a logical unit of work. They must exhibit four properties that are collectively known as ACID. Transactions ensure that multiple data modifications are processed as a unit. For example, a banking transaction might credit one account and debit another. Both steps must be completed together or not at all. SQL Server supports transaction processing to manage multiple transactions.

- Atomicity
  - Atomicity means that either all the steps in the transaction must succeed or none of them must be performed. For example, when money is transferred between two bank accounts, both the debit and credit actions must be performed or none should be performed.
- Consistency
  - Consistency ensures that when the transaction is complete, data must be in a consistent state. A consistent state is one where the data conforms to the business rules related to the data. Inconsistent data violates one or more business rules. In SQL Server, the database has to be in a consistent state after each statement within a transaction and not only at the end of the transaction.
- Isolation
  - Isolation defines that changes made by a transaction must be isolated from other concurrent transactions. 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 doesn't recognize an intermediate state.

- Durability
  - Durability defines that when the transaction is complete, the changes must be made permanent in the database and must survive even system failures.



Transaction Modes

Auto-Commit

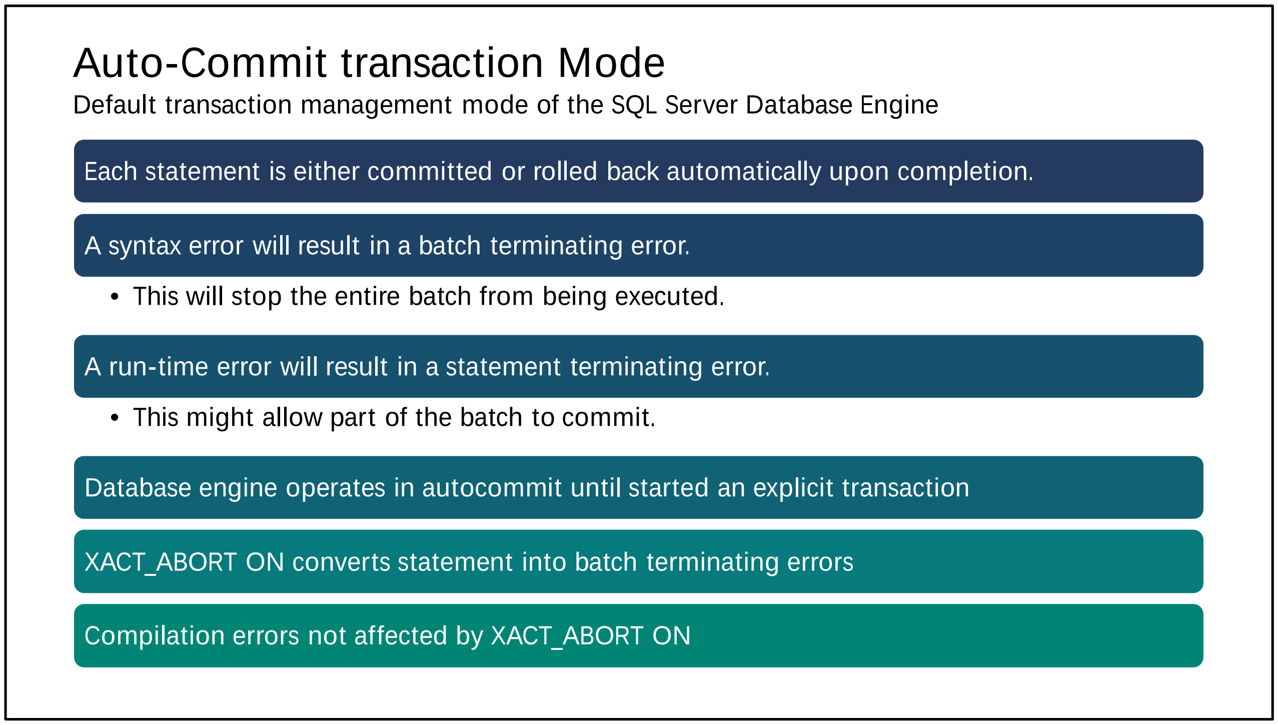
Individual statements that complete successfully, will be committed. If errors are encountered the statement is rolled back.

Explicit

Transaction is explicitly defined with a BEGIN TRANSACTION and COMMIT TRANSACTION statement.

Implicit

- Transaction starts automatically once first statement of a batch is received. Must still manually COMMIT or ROLLBACK transaction.



Auto-Commit transaction Mode

Default transaction management mode of the SQL Server Database Engine

Each statement is either committed or rolled back automatically upon completion.

A syntax error will result in a batch terminating error.

- This will stop the entire batch from being executed.

A run-time error will result in a statement terminating error.

- This might allow part of the batch to commit.

Database engine operates in autocommit until started an explicit transaction

XACT_ABORT ON converts statement into batch terminating errors

Compilation errors not affected by XACT_ABORT ON

Key Points:

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

The Autocommit mode:

- A connection to an instance of the database engine operates in the Autocommit mode whenever this default mode has not been overridden by either explicit or implicit transactions. The Autocommit mode is also the default mode for.NET Framework Data Provider
- A connection to an instance of the Database Engine operates in the Autocommit mode until a BEGIN TRANSACTION statement starts an explicit transaction, or implicit transaction mode is set on.
- When the explicit transaction is committed or rolled back, or when implicit transaction mode is turned off, the connection returns to the Autocommit mode.
  - **When SET XACT_ABORT is ON**, if a Transact-SQL statement raises a run-time error, **the entire transaction is terminated and rolled back.**
  - **When SET XACT_ABORT is OFF**, in some cases **only the Transact-SQL statement that raised the error is rolled back and the transaction continues processing.**

  Depending upon the severity of the error, the entire transaction may be rolled back even when SET XACT_ABORT is OFF. OFF is the default setting in a T-SQL statement, while ON is the default setting in a trigger.

- **Compile errors, such as syntax errors, are not affected** by SET XACT_ABORT.

Compile Errors

- Compile errors (such as syntax errors) are not affected by SET XACT_ABORT. When working in the Autocommit mode, compile time errors can cause more than one T-SQL statement to fail. In this mode, a batch of statements is compiled as a unit and if a compile error is found, nothing in the batch is compiled or executed.

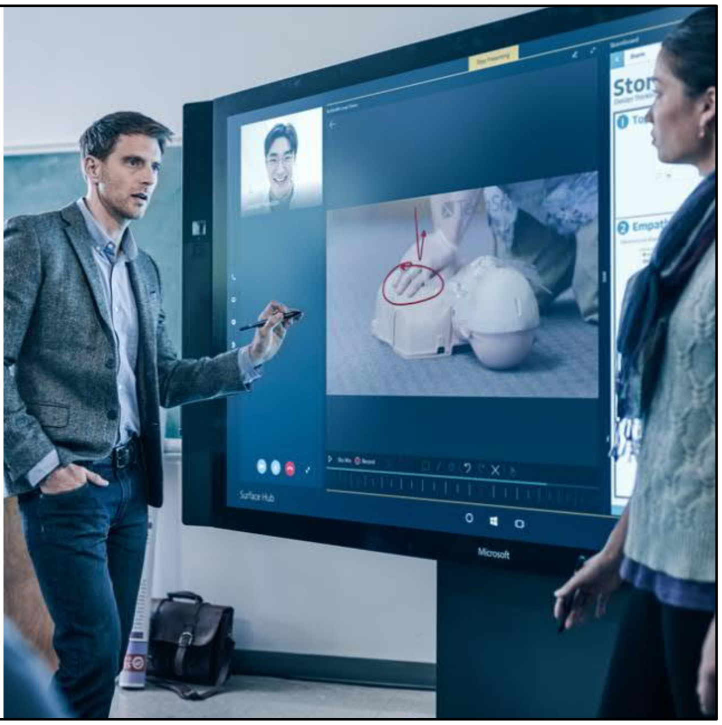
Example :

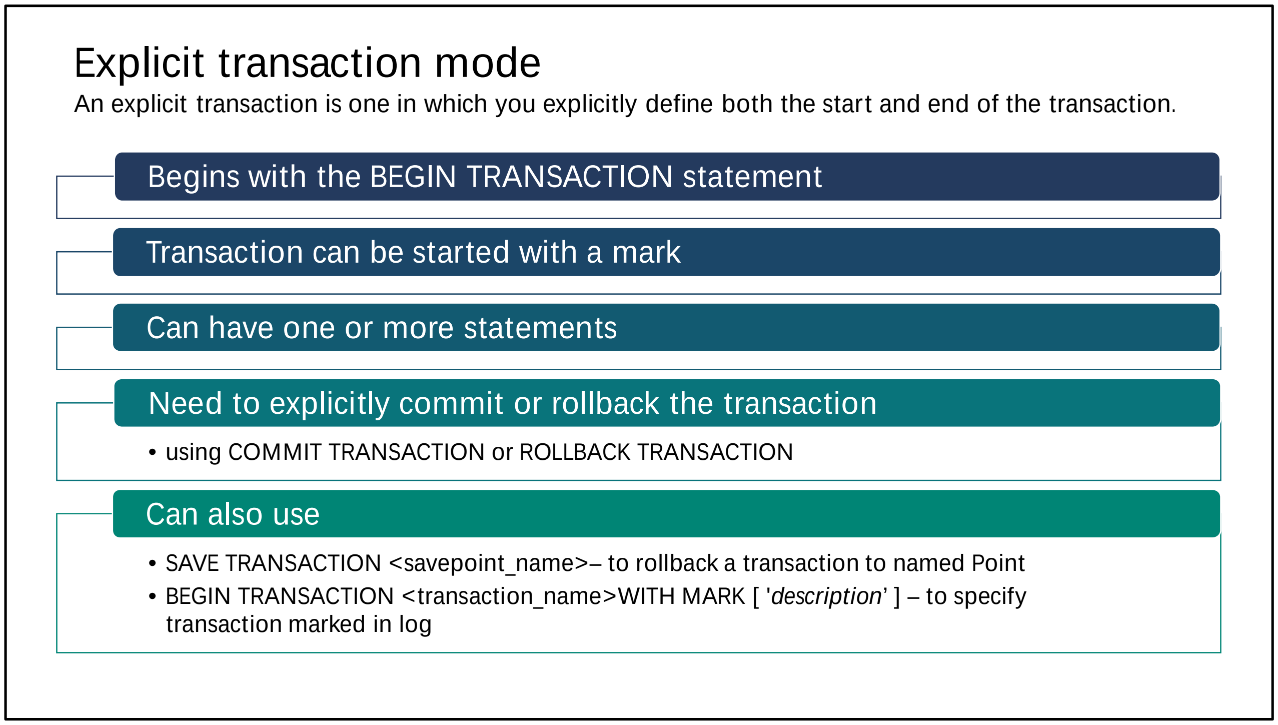
```
CREATE TABLE NewTable (Id INT PRIMARY KEY, Info CHAR(3));
GO
INSERT INTO NewTable VALUES (1, 'aaa');
INSERT INTO NewTable VALUES (2, 'bbb');
INSERT INTO NewTable VALUSE (3, 'ccc');  -- Syntax error.
GO
SELECT * FROM NewTable;  -- Returns no rows.
GO
```

References :https://docs.microsoft.com/en-us/sql/odbc/reference/develop-app/auto-commit-mode?view=sql-server-ver15

Demonstration

Compare Batch Terminating vs Statement Terminating Errors.





Explicit transaction mode

An explicit transaction is one in which you explicitly define both the start and end of the transaction.

- Begins with the BEGIN TRANSACTION statement
- Transaction can be started with a mark
- Can have one or more statements
- Need to explicitly commit or rollback the transaction
  - using COMMIT TRANSACTION or ROLLBACK TRANSACTION
- Can also use
  - SAVE TRANSACTION <savepoint_name>– to rollback a transaction to named Point
  - BEGIN TRANSACTION <transaction_name>WITH MARK ['*description*'] – to specify transaction marked in log

An explicit transaction is one in which you explicitly define both the start and end of the transaction. You can use explicit transactions to define your own units of business logic. For example, in a bank transfer function, you might enclose the withdrawal of funds from one account and the deposit of those funds in another account within one logical unit of work. DB-Library applications and Transact-SQL scripts use the BEGIN TRANSACTION, COMMIT TRANSACTION, COMMIT WORK, ROLLBACK TRANSACTION, or ROLLBACK WORK Transact-SQL statements to define explicit transactions.

Starting a Transaction:

- You start a transaction by using the BEGIN TRANSACTION statement. You can specify a name for the transaction and use the WITH MARK option to specify a description for the transaction to be marked in the transaction log. This transaction log mark can be used by database administrators when restoring a database to indicate the point to which you want to restore. The BEGIN TRANSACTION statement is often abbreviated to BEGIN TRAN.
- XACT_ABORT has an effect on explicit transaction as well as on implicit transactions.
- By default, only the statement in error would be rolled back and the batch would continue to run and commit the other statements in the transaction.
- Therefore error handling must be implemented (will be discussed in a later module) or XACT_ABORT can be turned ON to abort the batch and rollback the whole transaction in case of an error.

Committing a Transaction:

- You can commit the work contained in a transaction by issuing the COMMIT

TRANSACTION statement. Use this to end a transaction if no errors have occurred and you want the contents of the transaction to be committed to the database.

- The COMMIT TRANSACTION statement is often abbreviated to “COMMIT”. To assist users from other database platforms that are migrating to SQL Server, the statement COMMIT WORK can also be used.

Rolling Back a Transaction:

- You can cancel the work contained in a transaction by issuing the ROLLBACK TRANSACTION statement.Use this to end a transaction if errors have occurred and you want the changes made by the transaction to be undone and the database to return to the state it was before the transaction began.
- A rollback on an inner transaction also rolls back any outer transactions and a rollback on the outermost transaction will undo any inner committed transaction.

Saving a Transaction:

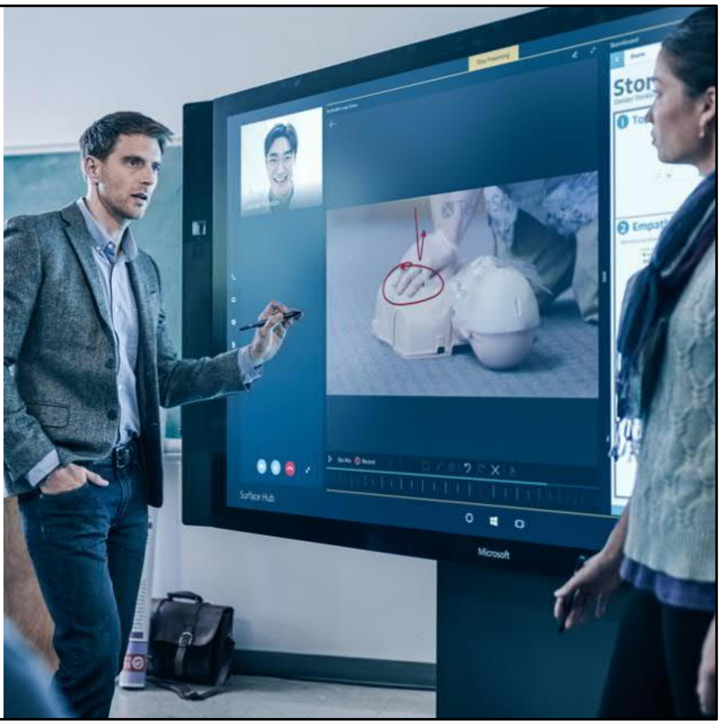
- By using savepoints, you can roll back a transaction to a named point in the transaction, instead of the beginning of the transaction. Inside the body of a transaction, create a savepoint by issuing the SAVE TRANSACTION statement and specifying the name of the savepoint. You can then use the ROLLBACK TRANSACTION statement and specify the savepoint name to roll the changes back to that point.
- Use savepoints when an error is unlikely to occur and the cost of checking the data before the error occurs is much higher than testing for the error after the data modifications have been submitted.

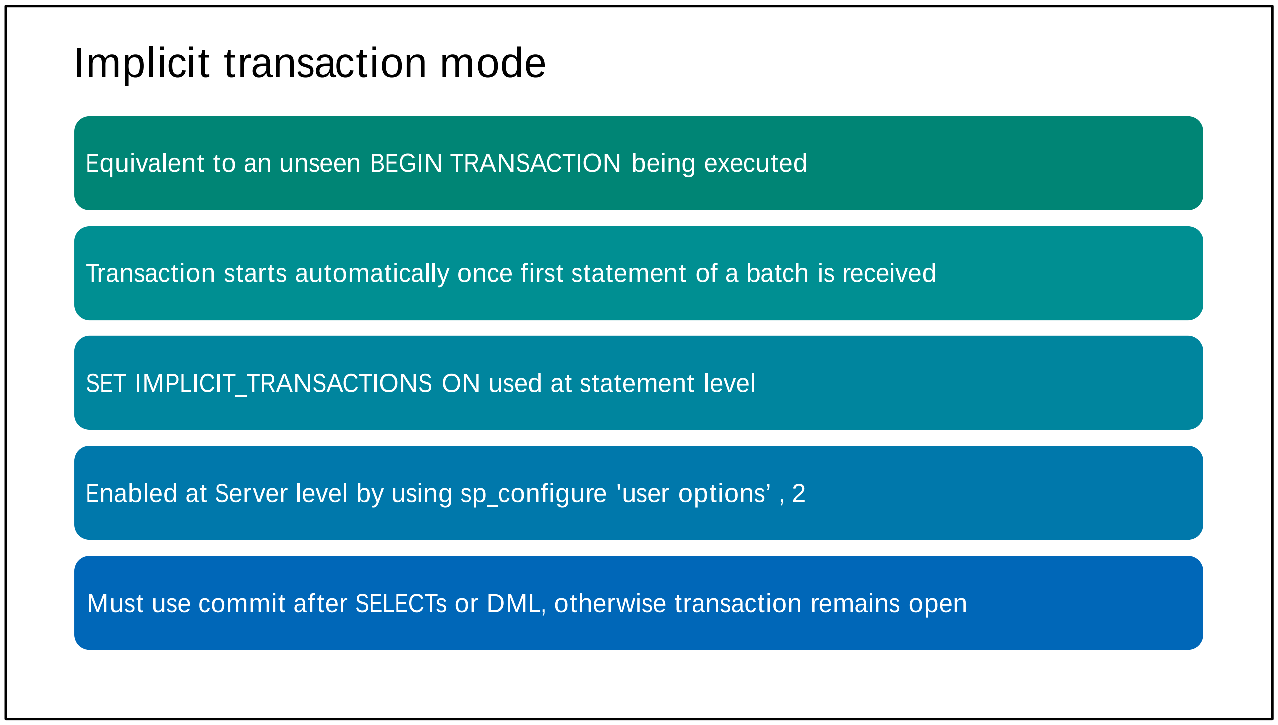
Marked Transactions

The WITH MARK option causes the transaction name to be placed in the transaction log. When restoring a database to an earlier state, the marked transaction can be used in place of a date and time

Demonstration

Transaction SavePoints





Implicit transaction mode

- Equivalent to an unseen BEGIN TRANSACTION being executed
- Transaction starts automatically once first statement of a batch is received
- SET IMPLICIT_TRANSACTIONS ON used at statement level
- Enabled at Server level by using sp_configure 'user options' , 2
- Must use commit after SELECTs or DML, otherwise transaction remains open

Key Point:

- **When ON**, the system is in *implicit* transaction mode. This means that if @@TRANCOUNT = 0, any of the following Transact-SQL statements begins a new transaction. It is equivalent to an unseen BEGIN TRANSACTION being executed
- **IMPLICIT_TRANSACTIONS ON is not popular.** In most cases where IMPLICIT_TRANSACTIONS is **ON, it is because the choice of SET ANSI_DEFAULTS ON** has been made.
- It leaves transaction open in SQL server until Commit/rollback is not used.

Recommendations : Don’t use Implicit transaction until required , Also if SET ANSI_DEFAULTS ON used in code then set it off at end otherwise it will leave IMPLICIT TRANSACTION ON

===============

More Information

===============

The Implicit transaction mode:

- SQL Server can automatically or, more precisely, implicitly starts a transaction for you if the SET IMPLICIT_TRANSACTIONS ON statement is run, or if the implicit transaction option is turned ON globally by running sp_configure user options 2. (The bit mask 0x2 must be turned ON for the user option, so you might have to perform an OR operation with the existing user option value.

- When a connection is operating in implicit transaction mode, the 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.
- After implicit transaction mode has been set on for a connection, the instance of the Database Engine automatically starts a transaction when it first executes any of these statements.
- In most cases, it is best to work in Autocommit mode and define transactions explicitly using the BEGIN TRANSACTION statement. However for applications that were originally developed on systems other than SQL Server, the implicit transaction mode can be useful.

- Implicit transaction mode automatically starts a transaction when you issue certain statements and the transaction then continues until you issue a commit statement or a rollback statement.

Example : SET IMPLICIT_TRANSACTIONS ON;

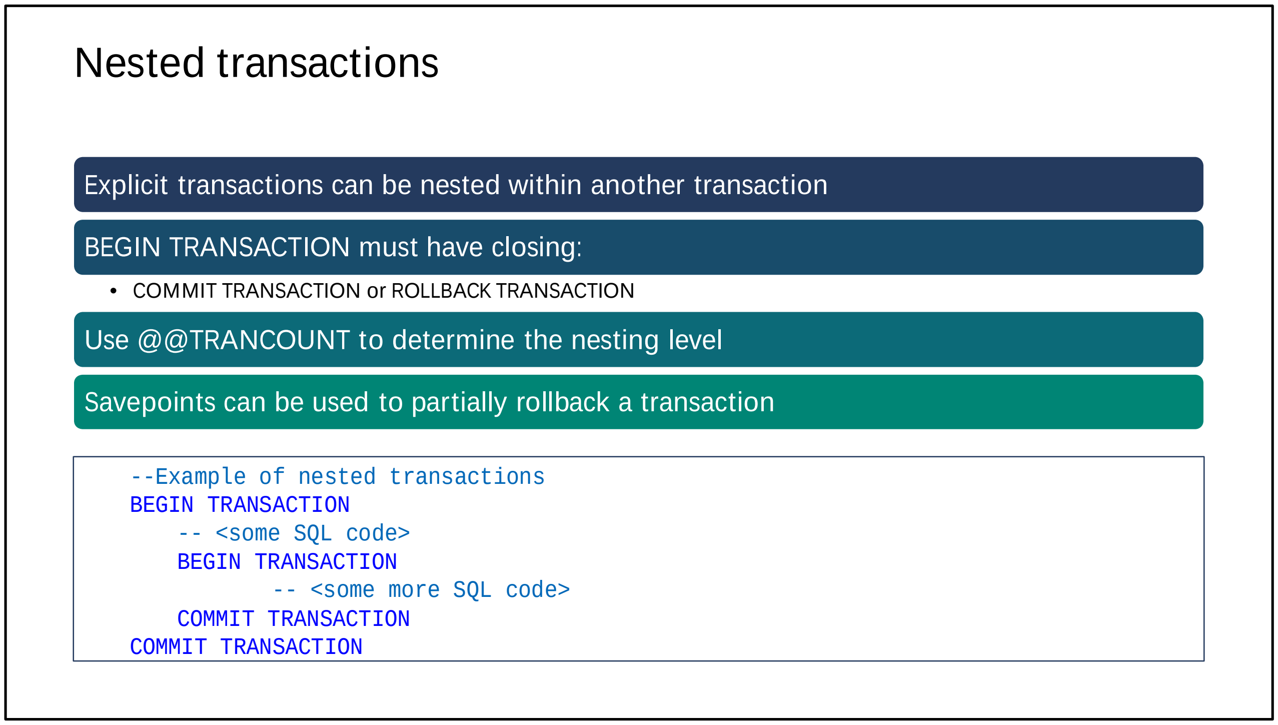
```
--Session 1
SET IMPLICIT_TRANSACTIONS ON;
GO
UPDATE [Demo].[DemoSalesOrderDetailSeed]
  SET [SpecialOfferID] = 0
  WHERE OrderID= 55;
GO
```

--Session 2 –it will show transaction of session 1 is still open

```
select * from Sys.sysprocesses where open_tran=1
```

References:

For more information on SET IMPLICIT_TRANSACTIONS Transact-SQL), refer:
https://docs.microsoft.com/en-us/sql/t-sql/statements/set-implicit-transactions-transact-sql?redirectedfrom=MSDN&view=sql-server-ver15



Nested transactions

- Explicit transactions can be nested within another transaction
- BEGIN TRANSACTION must have closing:
  - COMMIT TRANSACTION or ROLLBACK TRANSACTION
- Use @@TRANCOUNT to determine the nesting level
- Savepoints can be used to partially rollback a transaction

```
--Example of nested transactions
BEGIN TRANSACTION
    -- <some SQL code>
    BEGIN TRANSACTION
            -- <some more SQL code>
    COMMIT TRANSACTION
COMMIT TRANSACTION
```

Key Point:

- Nothing is committed until the final COMMIT TRANSACTION is executed
- Each Begin Transaction Need Commit Transaction whereas
- Only one Rollback will be sufficient to roll back all the statements to the first BEGIN TRANSACTION

The behavior of these nested transactions, however, might not be as expected. The first COMMIT TRANSACTION command above does not actually commit the inner transaction, it simply decrements the transaction count.

Nothing is committed until the final COMMIT TRANSACTION is executed. Rollbacks behave differently.

===============

More Information

===============

In the example above, if we were to replace the first COMMIT TRANSACTION statement with a ROLLBACK TRANSACTION statement, the transaction would be completely rolled back to the first BEGIN TRANSACTION and the transaction count would be set to zero once the ROLLBACK TRANSACTION statement was executed.

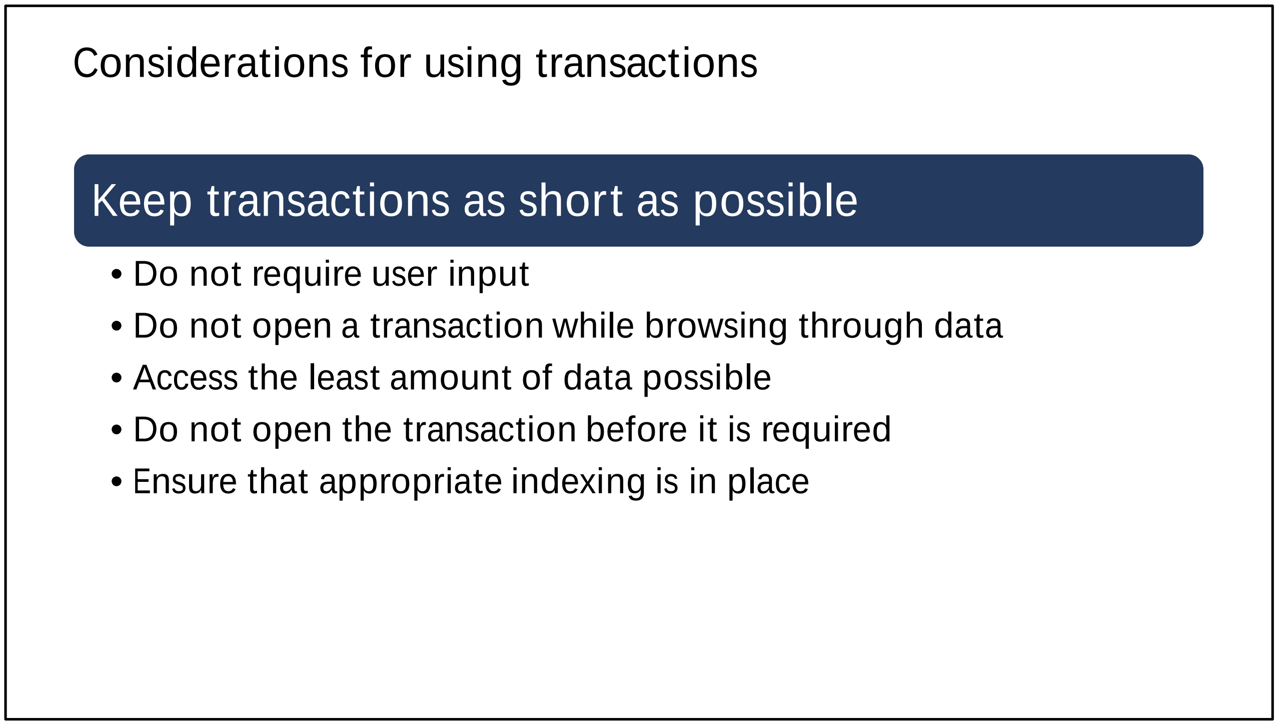
If you would like to partially rollback transactions, you must use a savepoint. Continuing with the example above, if we wanted to be able to rollback the inner transaction, we would have to rewrite the code as follows:

```
BEGIN TRANSACTION
-- <some SQL code>
SAVE TRANSACTION NestedTran
-- <some more SQL code that failed for some reason>
ROLLBACK TRANSACTION NestedTran
COMMIT TRANSACTION
-- commits the code that executed before the SAVE TRANSACTION
statement
```

References:

For more information on ROLLBACK TRANSACTION (Transact-SQL), refer
 https://docs.microsoft.com/en-us/sql/t-sql/language-elements/rollback-transaction-transact-sql

For more information on SAVE TRANSACTION (Transact-SQL), refer:
https://docs.microsoft.com/en-us/sql/t-sql/language-elements/save-transaction-transact-sql



Considerations for using transactions

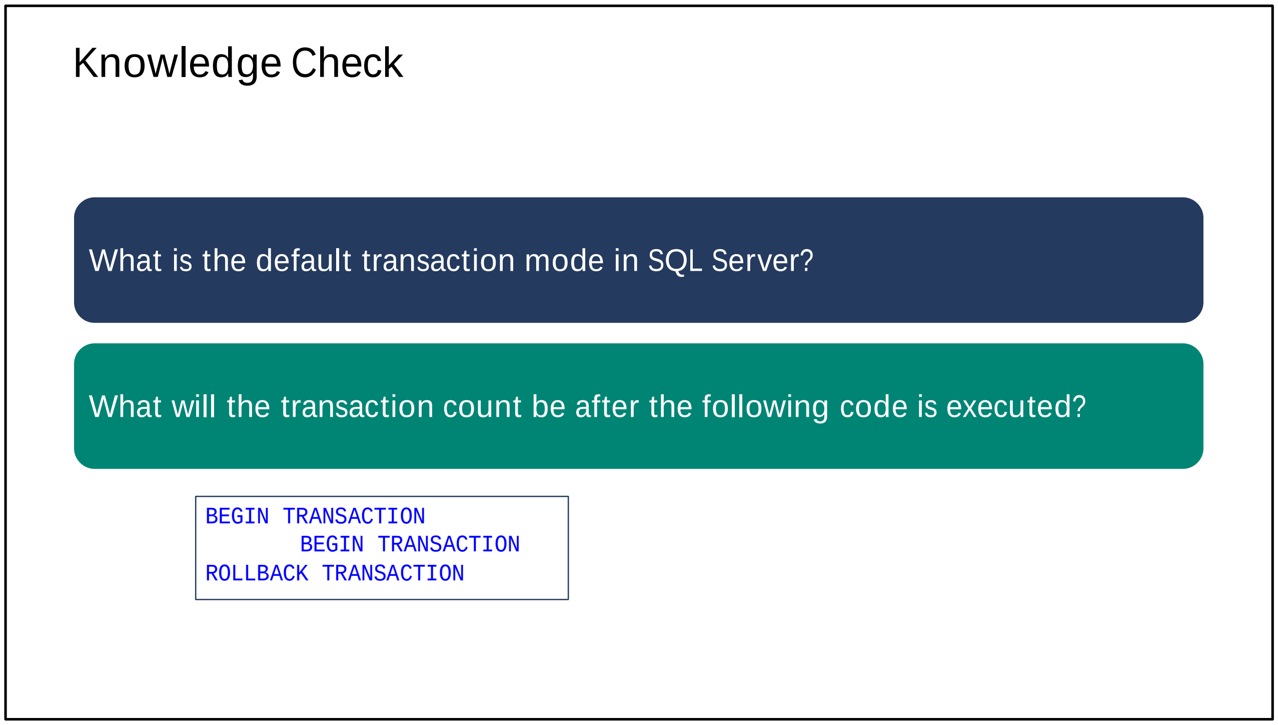
Keep transactions as short as possible

- Do not require user input
- Do not open a transaction while browsing through data
- Access the least amount of data possible
- Do not open the transaction before it is required
- Ensure that appropriate indexing is in place

There are many general considerations that need to be kept in mind when working with transactions.

- Keep transactions as short as possible.
- Transactions must be as short as possible. Longer transactions increase the likelihood that users will not be able to access locked data. Some methods to keep transactions short include the following:
  - Do not require input from users during a transaction. Address issues that require user interaction before you start the transaction. For example, if you are updating a customer record, obtain the necessary information from the user before you begin the transaction. A golden rule here is to never hold a transaction across a user interaction.
  - Do not open a transaction while browsing through data, if at all possible. Transactions must not start until all preliminary data analysis has been completed.
  - INSERT, UPDATE, and DELETE must be the primary statements in a transaction and they must be written to affect the fewest number of rows. A transaction must never be smaller than a logical unit of work.
  - Access the least amount of data possible while in a transaction. This decreases the number of locked rows and reduces contention.
- Ensure that appropriate indexing is in place as this reduces the number of pages that need to be accessed and locked.
- Try to access resources in the same order.
- Accessing resources in the same order within transactions tends to naturally serialize your access to the database and can help to avoid deadlocks. However, doing this is not always possible.

Note: *Deadlocks will be discussed later in the module.*



Knowledge Check

What is the default transaction mode in SQL Server?

What will the transaction count be after the following code is executed?

```
BEGIN TRANSACTION
       BEGIN TRANSACTION
ROLLBACK TRANSACTION
```

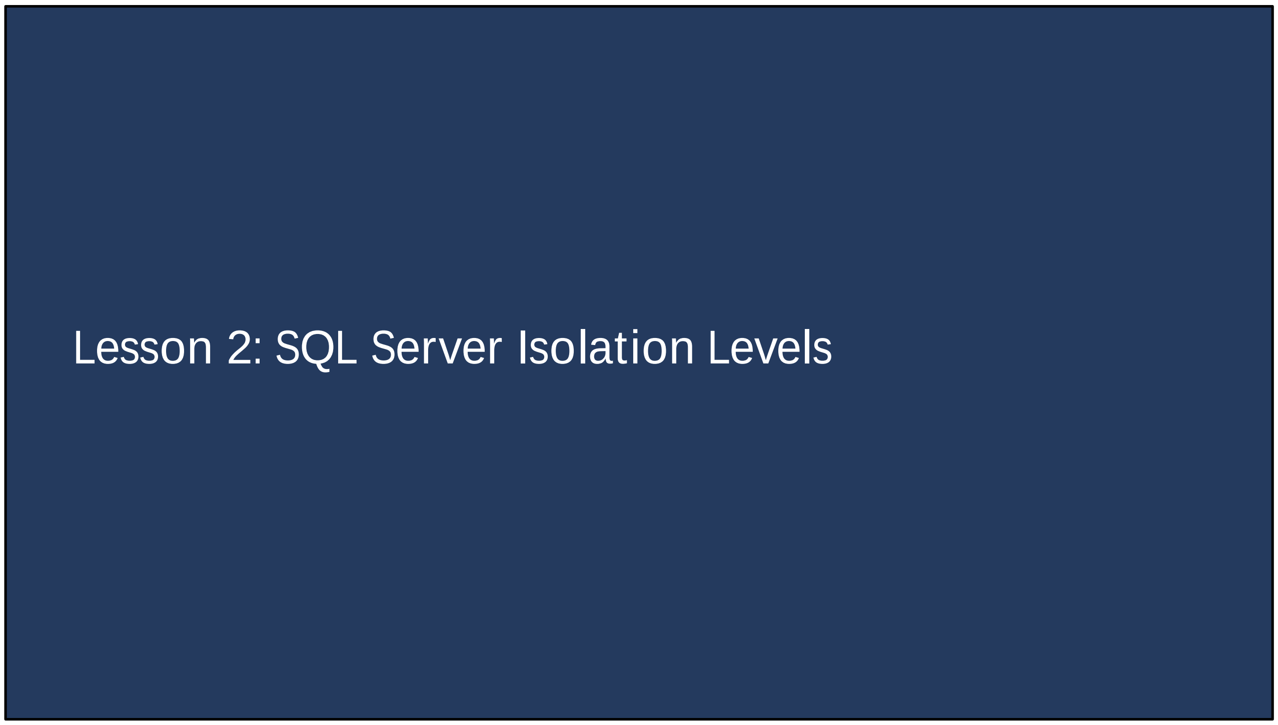
Q1: What is the default transaction mode in SQL Server?
A1: Autocommit

Q2: What will the transaction count be after the following code is executed

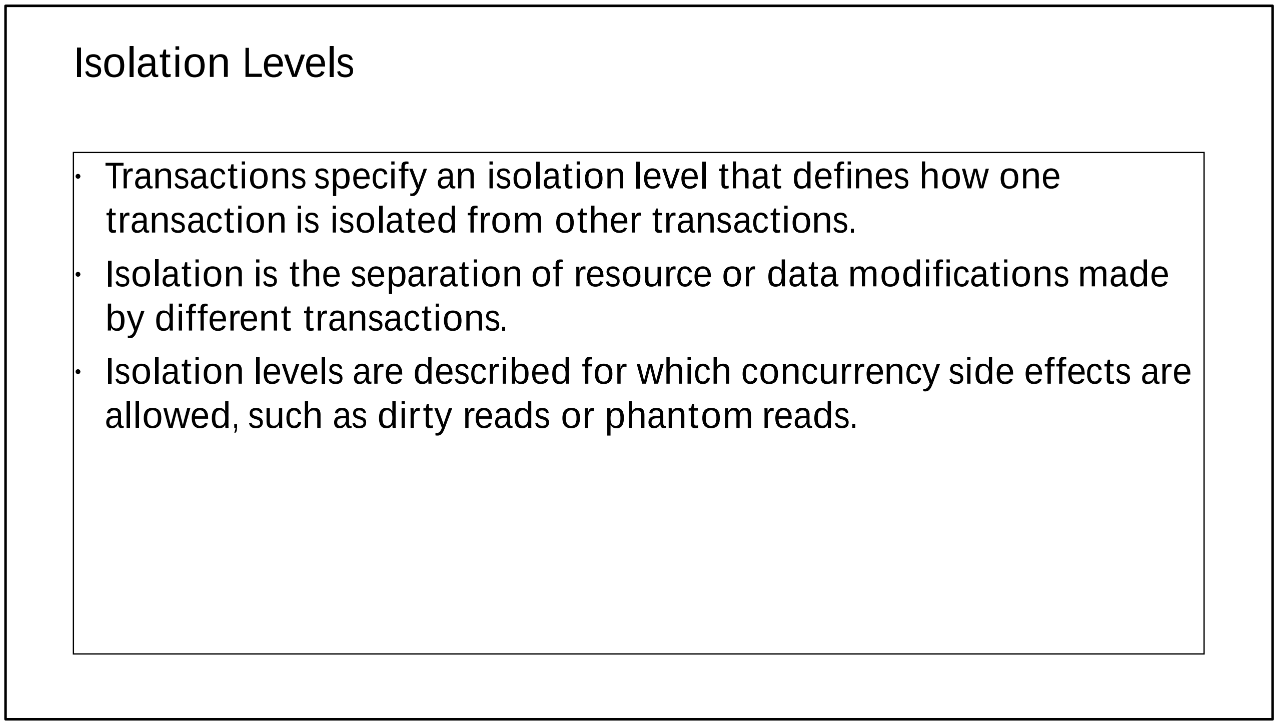
-
-
-

A2: 0

Q3: What are two different concurrency models supported by SQL server ?
A3: Pessimistic and Optimistic



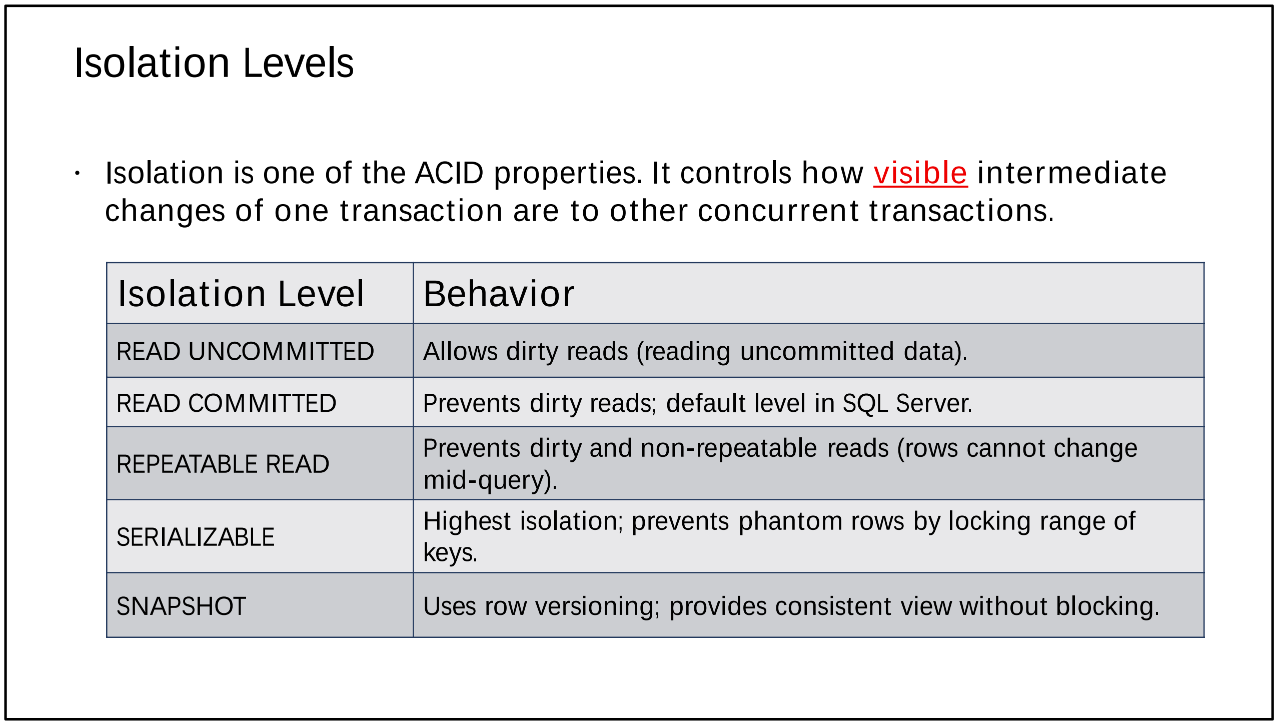
Lesson 2: SQL Server Isolation Levels



Isolation Levels

- Transactions specify an isolation level that defines how one transaction is isolated from other transactions.
- Isolation is the separation of resource or data modifications made by different transactions.
- Isolation levels are described for which concurrency side effects are allowed, such as dirty reads or phantom reads.

https://learn.microsoft.com/en-us/sql/relational-databases/sql-server-transaction-locking-and-row-versioning-guide?view=sql-server-ver17

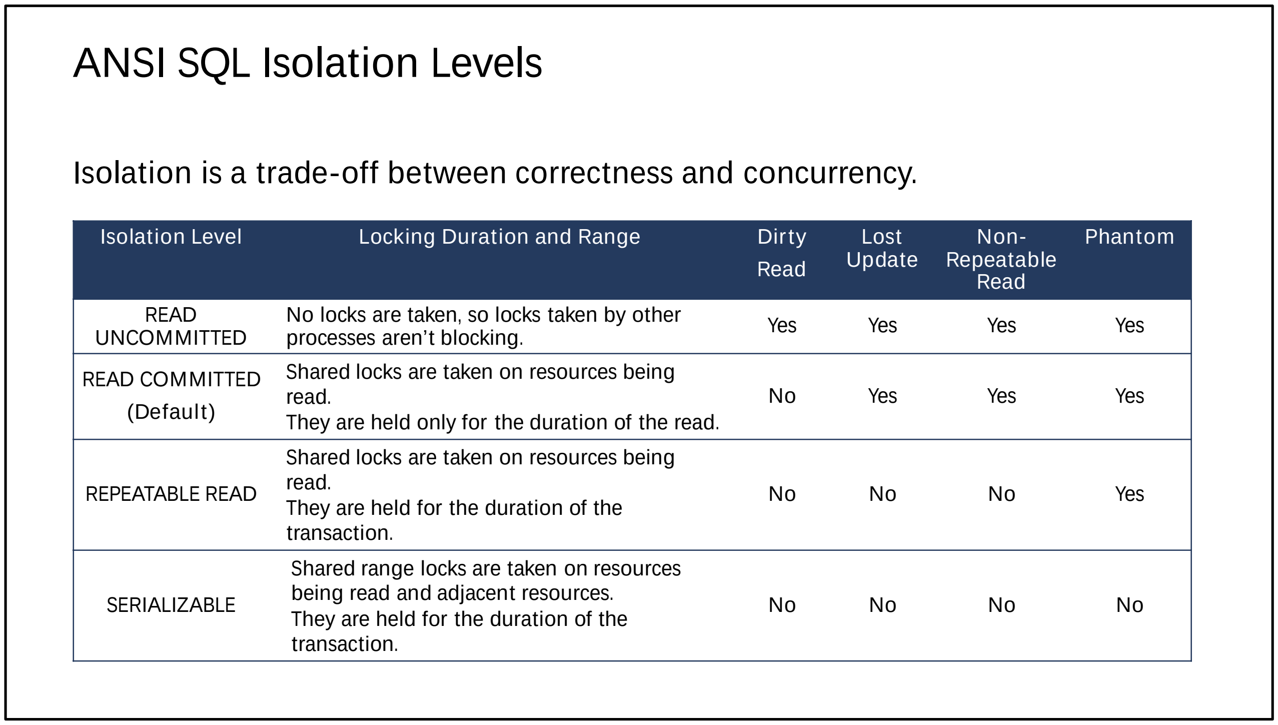


Isolation Levels

- Isolation is one of the ACID properties. It controls how visible intermediate changes of one transaction are to other concurrent transactions.

Isolation Level	Behavior
READ UNCOMMITTED	Allows dirty reads (reading uncommitted data).
READ COMMITTED	Prevents dirty reads; default level in SQL Server.
REPEATABLE READ	Prevents dirty and non-repeatable reads (rows cannot change mid-query).
SERIALIZABLE	Highest isolation; prevents phantom rows by locking range of keys.
SNAPSHOT	Uses row versioning; provides consistent view without blocking.

In SQL Server, phantom rows refer to a concurrency phenomenon that occurs when a transaction reads a set of rows based on a condition, and another transaction inserts new rows that satisfy that condition before the first transaction completes. When the first transaction re-executes the same query, it sees additional rows that were not there before—these are called phantoms.



ANSI SQL Isolation Levels

Isolation is a trade-off between correctness and concurrency.

Isolation Level	Locking Duration and Range	Dirty Read	Lost Update	Non-Repeatable Read	Phantom
READ UNCOMMITTED	No locks are taken, so locks taken by other processes aren't blocking.	Yes	Yes	Yes	Yes
READ COMMITTED (Default)	Shared locks are taken on resources being read. They are held only for the duration of the read.	No	Yes	Yes	Yes
REPEATABLE READ	Shared locks are taken on resources being read. They are held for the duration of the transaction.	No	No	No	Yes
SERIALIZABLE	Shared range locks are taken on resources being read and adjacent resources. They are held for the duration of the transaction.	No	No	No	No

Key Points:

- An isolation level defines the degree to which one transaction must be isolated from resource or data modifications made by other transactions.
- By default, SQL Server operates at an isolation level of READ COMMITTED.

===============

More Information

===============

Isolation levels are described in terms of which concurrency side-effects, such as dirty reads or phantom reads, are allowed.

To make use of either more or less-strict isolation levels in applications, you can customize locking for an entire session by setting the isolation level of the session with the SET TRANSACTION ISOLATION LEVEL statement.

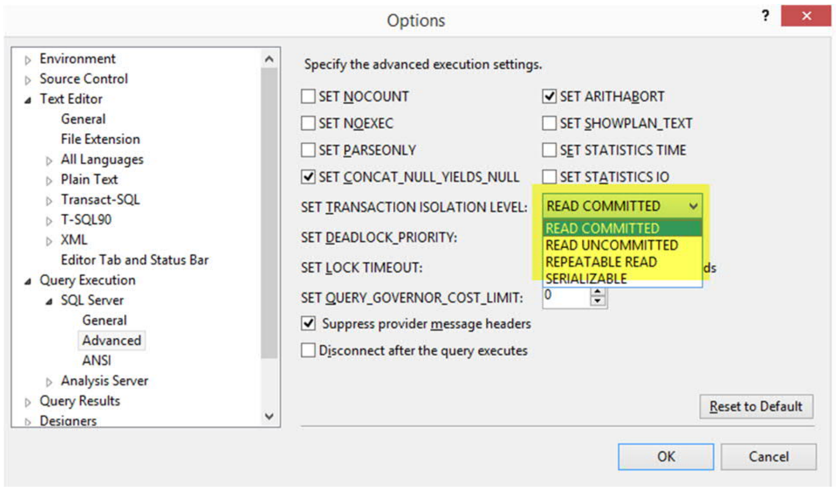
SNAPSHOT isolation is not an ANSI standard isolation level, it is an isolation level introduced in SQL Server 2005 to provide serializable-like consistency without the same concurrency issues. SNAPSHOT isolation makes use of the row-version store in tempdb to create a copy of a row undergoing modification. In a SNAPSHOT transaction, SQL Server will not acquire Shared locks for reading data, it will read the row-version instead. Because of the added overhead of maintaining the row-version store for all data modifications, SNAPSHOT isolation is not on by default and must be enabled in the database(s) where it will be used.

References:

For more information on Isolation Levels in the Database Engine, refer: https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/ms189122(v=sql.105)?redirectedfrom=MSDN

For more information on Understanding Row Versioning-Based Isolation Levels, refer: https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/ms189050(v=sql.105)?redirectedfrom=MSDN

Isolation Levels



Specify the advanced execution settings.

- [ ] SET NOCOUNT
- [ ] SET NOEXEC
- [ ] SET PARSEONLY
- [✓] SET CONCAT_NULL_YIELDS_NULL
- [✓] SET ARITHABORT
- [ ] SET SHOWPLAN_TEXT
- [ ] SET STATISTICS TIME
- [ ] SET STATISTICS IO

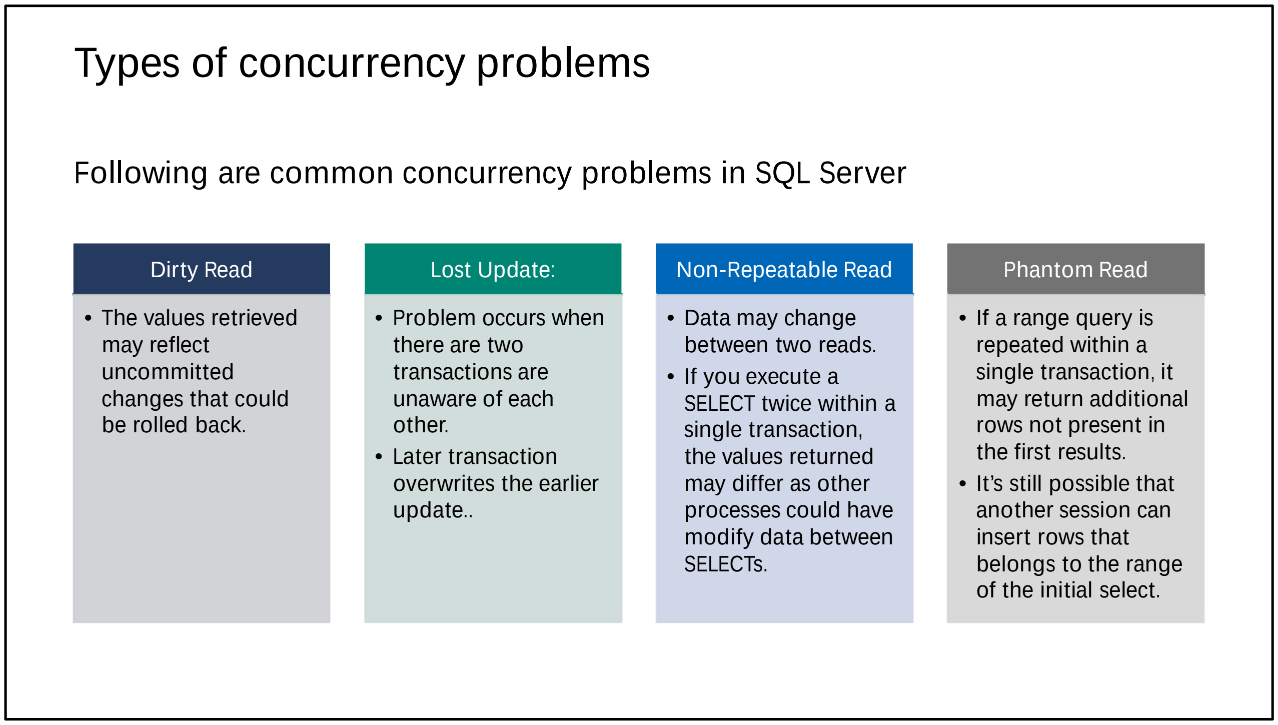
SET TRANSACTION ISOLATION LEVEL: READ COMMITTED
[✓ READ COMMITTED] READ UNCOMMITTED REPEATABLE READ SERIALIZABLE

SET DEADLOCK_PRIORITY:

SET LOCK TIMEOUT: ds

SET QUERY_GOVERNOR_COST_LIMIT: 0

- [✓] Suppress provider message headers
- [ ] Disconnect after the query executes



Types of concurrency problems

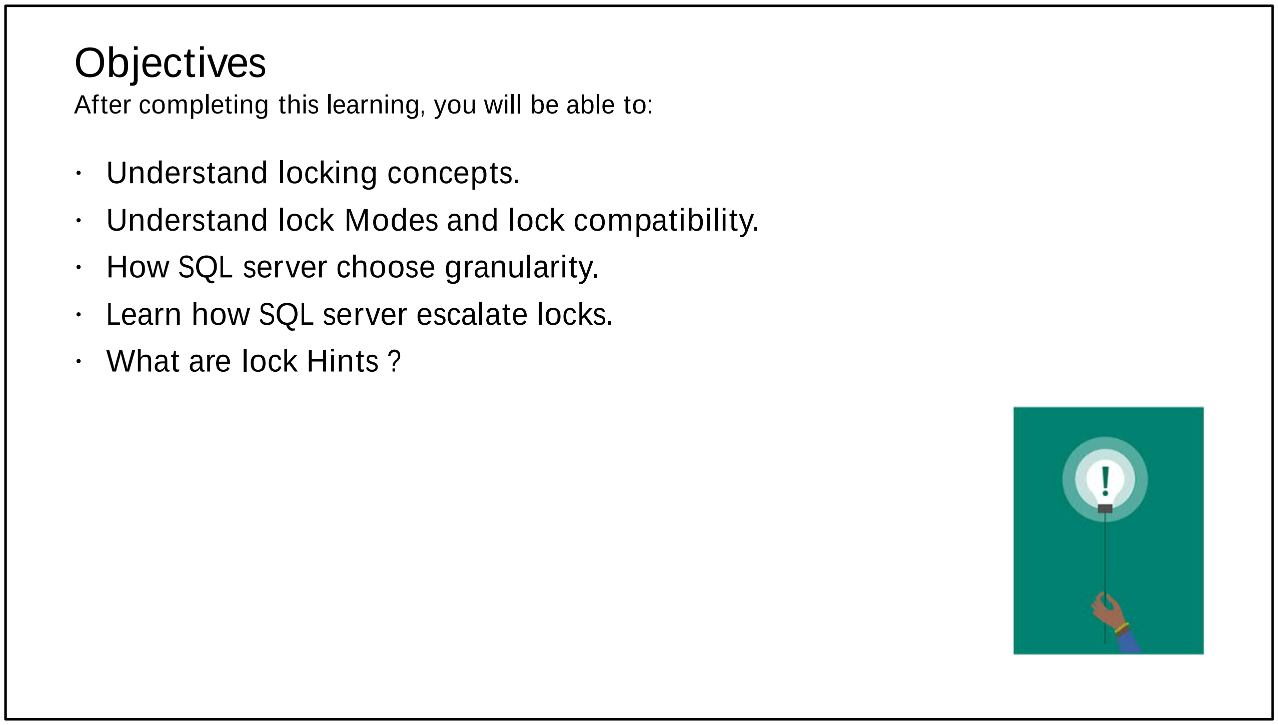
Following are common concurrency problems in SQL Server

Dirty Read	Lost Update:	Non-Repeatable Read	Phantom Read
• The values retrieved may reflect uncommitted changes that could be rolled back.	• Problem occurs when there are two transactions are unaware of each other. • Later transaction overwrites the earlier update..	• Data may change between two reads. • If you execute a SELECT twice within a single transaction, the values returned may differ as other processes could have modify data between SELECTs.	• If a range query is repeated within a single transaction, it may return additional rows not present in the first results. • It's still possible that another session can insert rows that belongs to the range of the initial select.

Key Points:

- Dirty Read: Other users can see uncommitted data.
- Lost Update: Problem occurs when there are two transactions are unaware of each other. Later transaction overwrites the earlier update..
- Non-Repeatable Read: Update made by another session can be seen by transaction which are not completed yet.
- Phantom Read: Insert made by another session can be seen by session whose transaction is not completed yet.

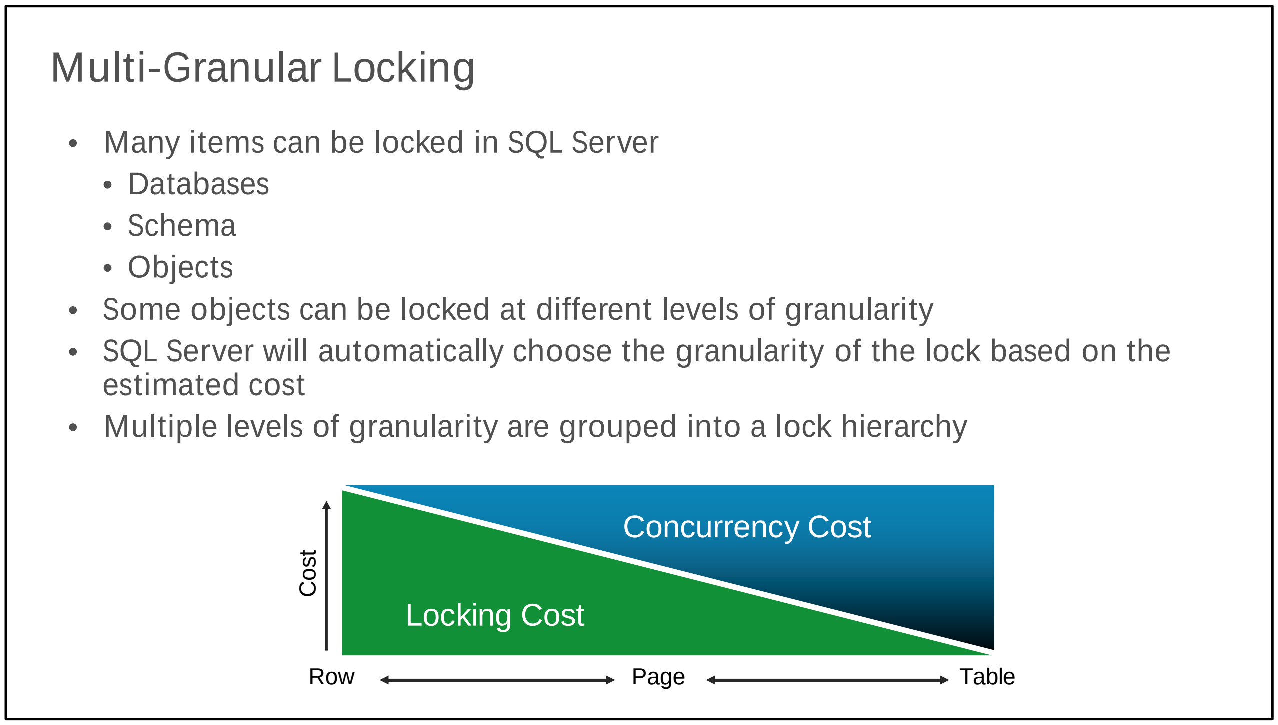
Lesson 3: SQL Server Locking



Objectives

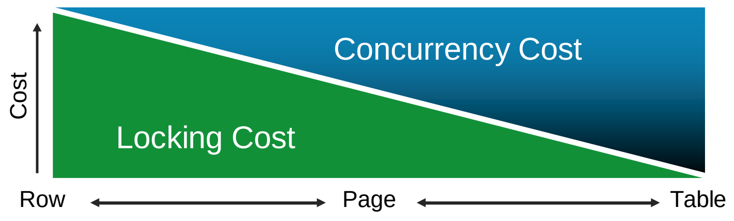
After completing this learning, you will be able to:

- Understand locking concepts.
- Understand lock Modes and lock compatibility.
- How SQL server choose granularity.
- Learn how SQL server escalate locks.
- What are lock Hints ?



Multi-Granular Locking

- Many items can be locked in SQL Server
  - Databases
  - Schema
  - Objects
- Some objects can be locked at different levels of granularity
- SQL Server will automatically choose the granularity of the lock based on the estimated cost
- Multiple levels of granularity are grouped into a lock hierarchy



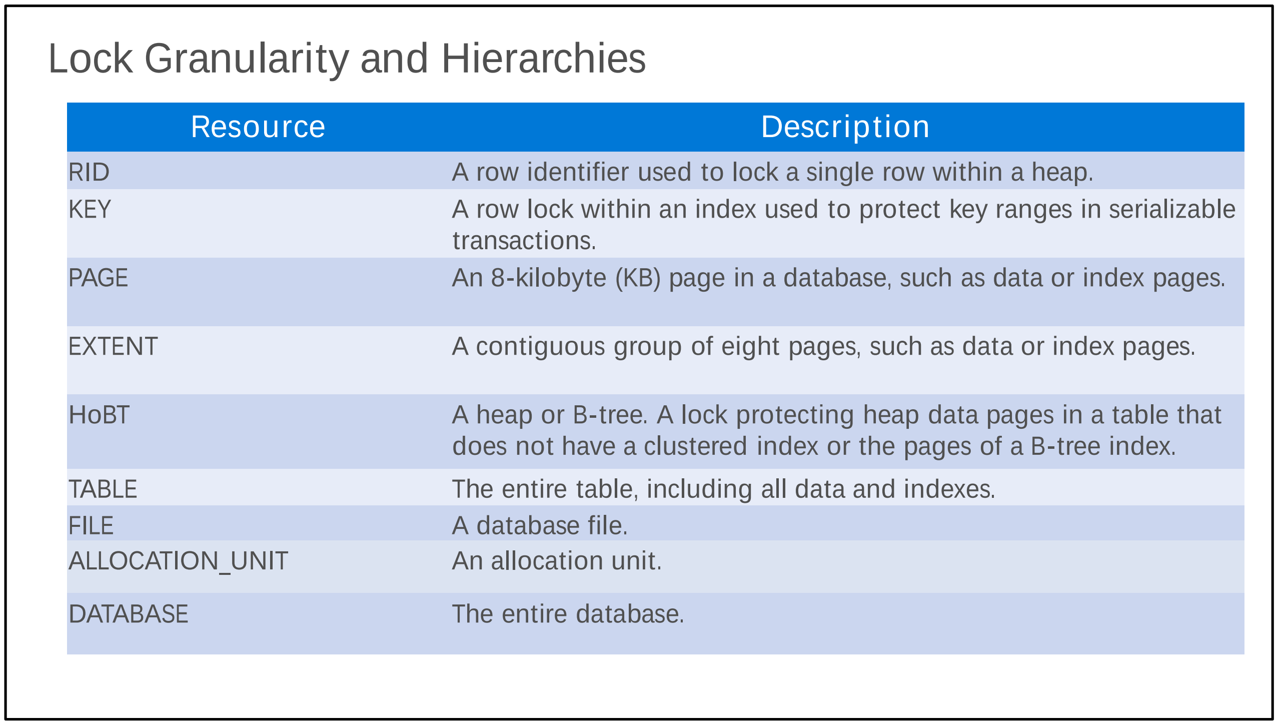
SQL Server locks resources in order to help protect them from concurrent users. Locks can be acquired on many different resources such as:

- A database
- A database file
- An entire table, including all data and indexes
- An 8 kilobyte (KB) data page or index page
- A row within an index
- Metadata information
- Internal storage structures, such as heap or B-tree (HoBT) structures
- Allocation units
- A lock resource defined by an application

Multi-granular locking

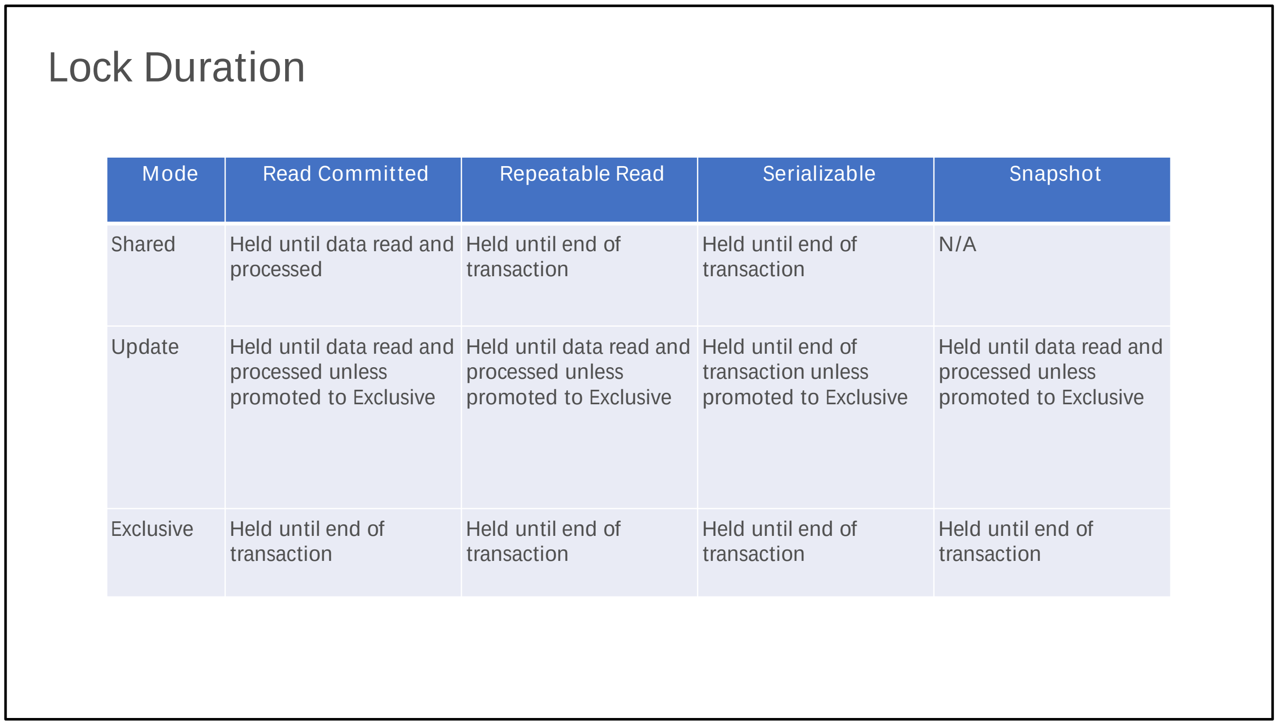
- To minimize the cost of locking, SQL Server locks resources automatically at a level appropriate to the task. SQL Server has multi-granular locking that allows different types of resources to be locked by a transaction
- Locking at a larger granularity, such as tables is expensive in terms of concurrency because locking an entire table restricts access to any part of the table by other transactions but has a lower overhead because fewer locks are being maintained.
- 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

- The Database Engine often needs to acquire locks at multiple levels of granularity to fully help protect a resource. This group of locks at multiple levels of granularity is called a lock hierarchy



Lock Granularity and Hierarchies

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 heap data pages in a table that does not have a clustered index or the pages of a B-tree index.
TABLE	The entire table, including all data and indexes.
FILE	A database file.
ALLOCATION_UNIT	An allocation unit.
DATABASE	The entire database.

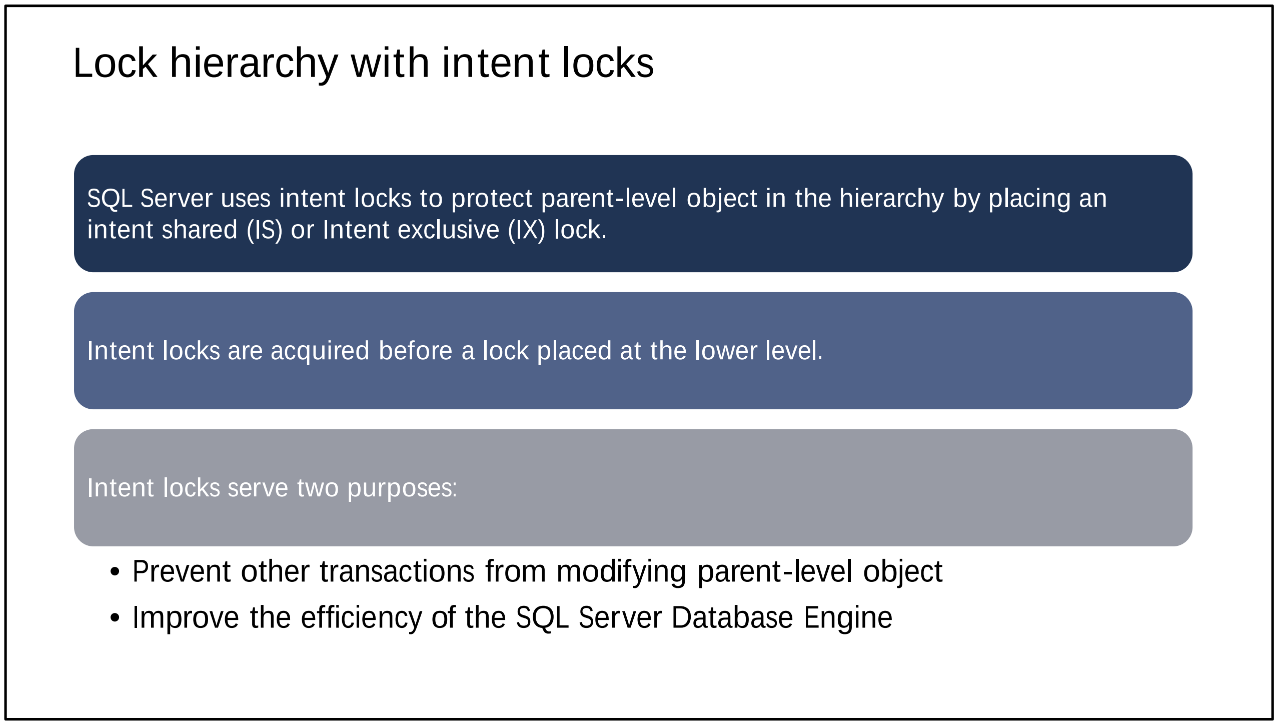


Lock Duration

Mode	Read Committed	Repeatable Read	Serializable	Snapshot
Shared	Held until data read and processed	Held until end of transaction	Held until end of transaction	N/A
Update	Held until data read and processed unless promoted to Exclusive	Held until data read and processed unless promoted to Exclusive	Held until end of transaction unless promoted to Exclusive	Held until data read and processed unless promoted to Exclusive
Exclusive	Held until end of transaction	Held until end of transaction	Held until end of transaction	Held until end of transaction

Lock ownership

- Most of the locking discussion in this topic relates to the locks owned by transactions. In addition to transactions, cursors and sessions can be owners of locks and they both affect the duration for which locks are held.
- When you use the SCROLL_LOCKS option, a cursor lock is held for every row that is fetched until the next row is fetched or the cursor is closed, regardless of the state of a transaction.
- Locks owned by session are outside the scope of a transaction. The duration of these locks is bounded by the connection and the process will continue to hold these locks until the process disconnects. A typical lock owned by session is the database (DB) lock. Shared DB lock is not held when the database is in Single user mode.



Lock hierarchy with intent locks

SQL Server uses intent locks to protect parent-level object in the hierarchy by placing an intent shared (IS) or Intent exclusive (IX) lock.

Intent locks are acquired before a lock placed at the lower level.

Intent locks serve two purposes:

- Prevent other transactions from modifying parent-level object
- Improve the efficiency of the SQL Server Database Engine

The example provided in the slide shows, if one transaction (T1) holds an exclusive lock at the table level, and another transaction (T2) holds an exclusive lock at the row level, each of the transactions believe that they have exclusive access to the resource. In this scenario, because T1 believes that it locks the entire table, it might inadvertently make changes to the same row that T2 thought it has locked exclusively.

In a multi-granular locking environment, there must be a way to effectively overcome this problem. The solution is intent lock.

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.

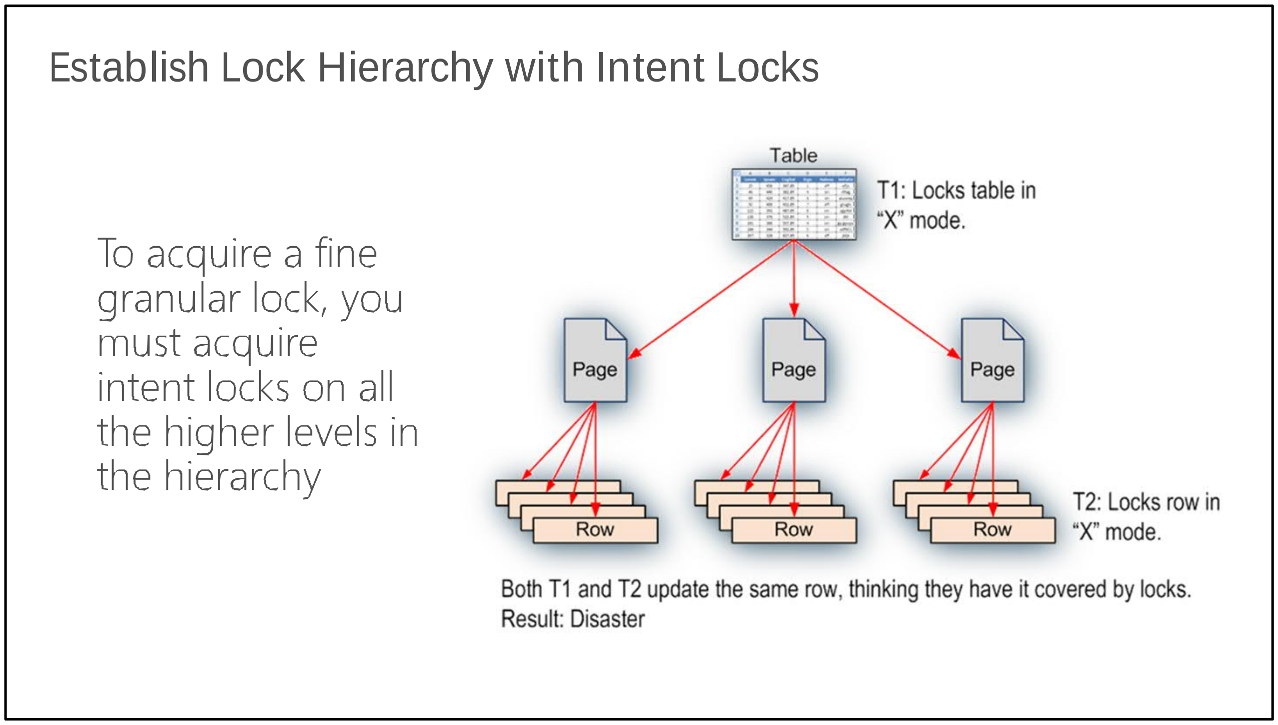
Intent locks serve two purposes:

- 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 placed at the table level means that a transaction intends on placing shared (S) locks 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.

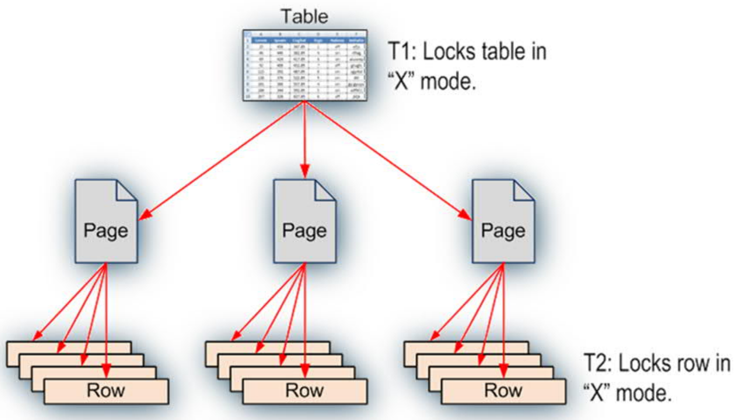
Intent locks improve performance because SQL Server examines intent locks only at the table level to determine whether a transaction can safely acquire a lock on that table. This removes the requirement to examine every row or page lock on the table to determine whether a transaction can lock the entire table.

References https://docs.microsoft.com/en-us/sql/relational-databases/sql-server-transaction-locking-and-row-versioning-guide?view=sql-server-ver15



Establish Lock Hierarchy with Intent Locks

To acquire a fine granular lock, you must acquire intent locks on all the higher levels in the hierarchy



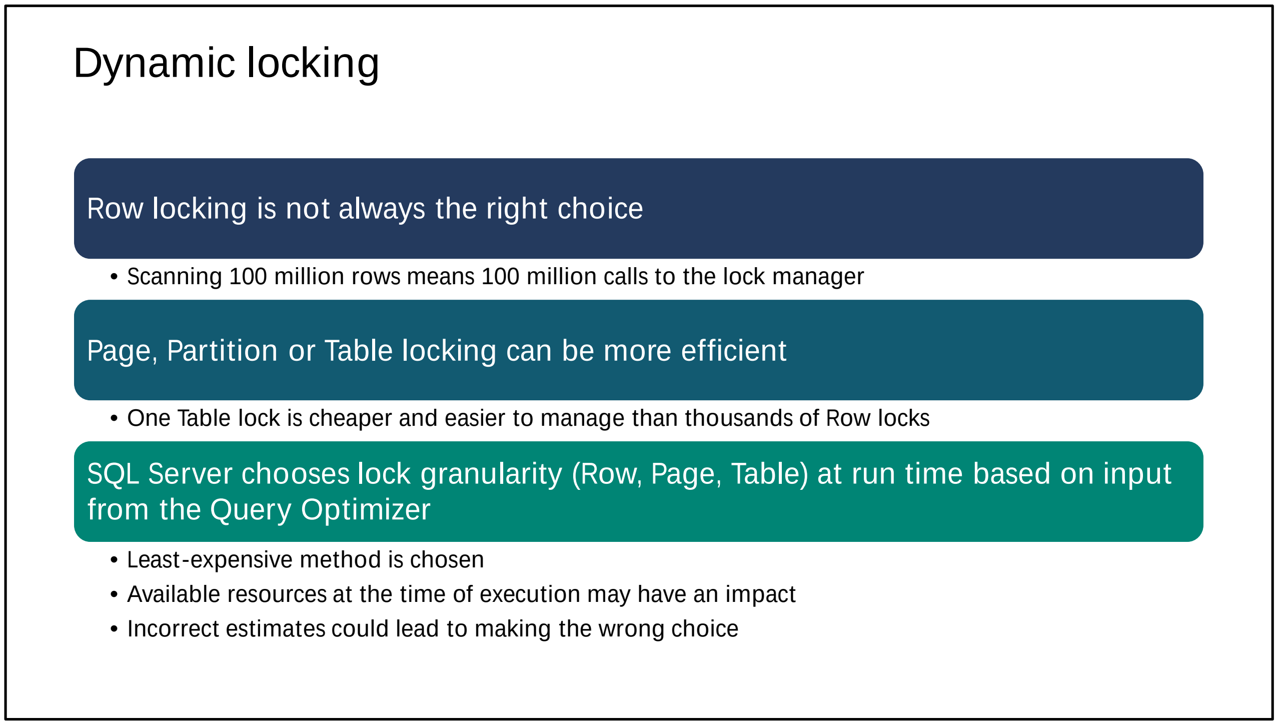
Both T1 and T2 update the same row, thinking they have it covered by locks.
Result: Disaster

The example provided in the slide shows, if one transaction (T1) holds an exclusive lock at the table level, and another transaction (T2) holds an exclusive lock at the row level, each of the transactions believe that they have exclusive access to the resource. In this scenario, because T1 believes that it locks the entire table, it might inadvertently make changes to the same row that T2 thought it has locked exclusively. In a multi-granular locking environment, there must be a way to effectively overcome this problem. The solution is intent lock.

An Intent lock is used to establish a lock hierarchy. SQL Server Database Engine acquires low-level locks and also places intent locks on the objects that contain the lower-level objects as follows:

- When locking rows or index key ranges, intent lock is acquired on the pages that contain the rows or keys
- When locking pages, intent lock is acquired on the higher-level objects that contain the pages. In addition to the intent lock on the object, intent page locks are requested on the following objects:
  - Leaf-level pages of non-clustered indexes
  - Data pages of clustered indexes
  - Heap data pages

An intent lock indicates that SQL Server wants to acquire a shared (S) lock or an exclusive (X) lock on some of the resources lower down in the hierarchy. For example, a shared intent lock placed at the table level means that a transaction intends on placing shared (S) locks 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 SQL Server examines intent locks only at the table level to determine whether 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 whether a transaction can lock the entire table



Dynamic locking

Row locking is not always the right choice

- Scanning 100 million rows means 100 million calls to the lock manager

Page, Partition or Table locking can be more efficient

- One Table lock is cheaper and easier to manage than thousands of Row locks

SQL Server chooses lock granularity (Row, Page, Table) at run time based on input from the Query Optimizer

- Least-expensive method is chosen
- Available resources at the time of execution may have an impact
- Incorrect estimates could lead to making the wrong choice

Key Points

- 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.

===============

More Information

===============

- When modifying individual rows, SQL Server typically takes row locks to maximize concurrency (for example, OLTP and order-entry applications).
- When scanning larger volumes of data, it would be more appropriate to take page or table locks to minimize the cost of acquiring locks (for example, Decision Support System (DSS), data warehouse, and reporting).
- Allowing SQL Server to use locks dynamically is the recommended configuration.
- However, you can set locks and override the ability of SQL Server to allocate lock resources dynamically.

When the server is started with locks set to 0, the lock manager acquires sufficient memory from the Database Engine for an initial pool of 2,500 lock structures. As the lock pool is exhausted, additional memory is acquired for the pool.

Generally, if more memory is required for the lock pool than is available in the Database Engine memory pool, and more computer memory is available (the max server memory threshold has not been reached), the Database Engine allocates memory dynamically to satisfy the request for locks.

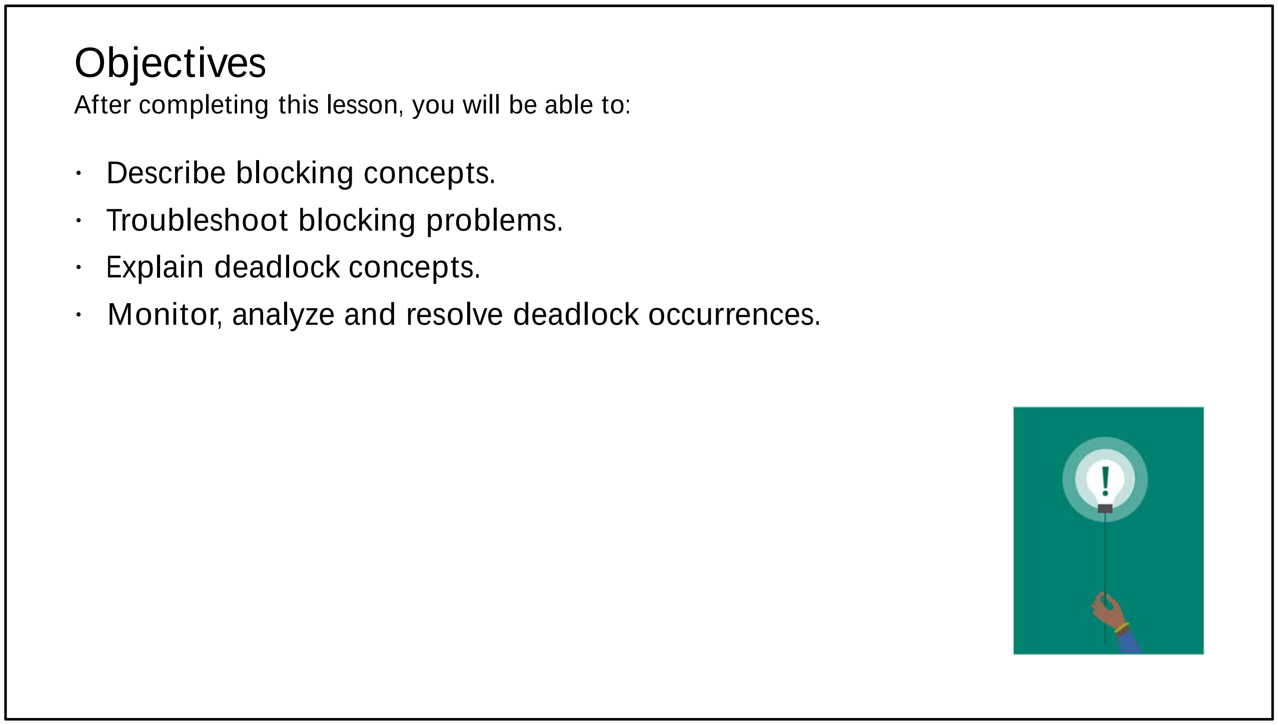
However, if allocating that memory would cause paging at the operating system level (for example, if another application is running on the same computer as an instance of SQL Server and using that memory), more lock space is not allocated.

The dynamic lock pool does not acquire more than 60% of the memory allocated to the Database Engine. After the lock pool has reached 60% of the memory acquired by an instance of the Database Engine, or no more memory is available on the computer, further requests for locks generates an error.

The locks option also affects when lock escalation occurs. When locks are set to 0, lock escalation occurs when the memory used by the current lock structures reaches 40% of the Database Engine memory pool. When locks are not set to 0, lock escalation occurs when the number of locks reaches 40% of the value specified for locks.

For more information on Configure the locks Server Configuration Option, refer: https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/configure-the-locks-server-configuration-option?redirectedfrom=MSDN&view=sql-server-ver15

Lesson 4: Troubleshooting Concurrency Performance



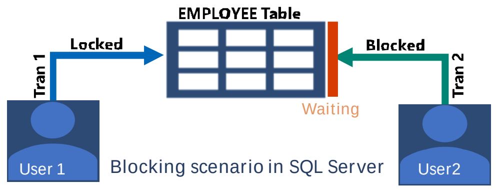
Objectives

After completing this lesson, you will be able to:

- Describe blocking concepts.
- Troubleshoot blocking problems.
- Explain deadlock concepts.
- Monitor, analyze and resolve deadlock occurrences.

Blocking

Blocking is an unavoidable characteristic of any RDBMS with lock-based concurrency.



Blocking scenario in SQL Server

Capturing blocking Information

- A Custom SQL scripts using DMVs that monitor locking and blocking
- Use SSMS standard reports i.e. Activity – All blocking transactions
- Extended events - blocked process report

Key Points:

- Blocking is an unavoidable characteristic of any relational database management system (RDBMS). This is normal behavior and may happen many times throughout the course of a day with no noticeable effect on system performance.

  - On SQL Server, blocking occurs when one SPID holds a lock on a specific resource and a second SPID attempts to acquire a conflicting lock type on the same resource. In this case you can say first SPID is blocking second SPID. SPID stands for Server Process ID. It is a unique identifier assigned by SQL Server to each user connection or process running on the instance.
  - Typically, the time frame for which the first SPID locks the resource is very small.
  - When it releases the lock, the second connection is free to acquire its own lock on the resource and continue processing.
  - sys.dm_tran_locks
    - Each row has information about both the resource and the request.
    - Request_status = WAIT implies blocking.
  - sys.dm_os_waiting_tasks

    Blocking_session_id > 0
  - sys.dm_exec_requests
    - Status = suspended and blocking_session_id > 0
  - Sys.Sysprocesses
    - Blocked_id>0

===============

More Information

===============

The duration and transaction context of a query determine how long its locks are held and, thereby, their impact on other queries.

If the query is not executed within a transaction (and no lock hints are used), the locks *for SELECT statements will only be held on a resource at the time it is actually being read*, not for the duration of the query. For INSERT, UPDATE, and DELETE statements, the locks are held for the duration of the query, both for data consistency and to allow the query to be rolled back if necessary.

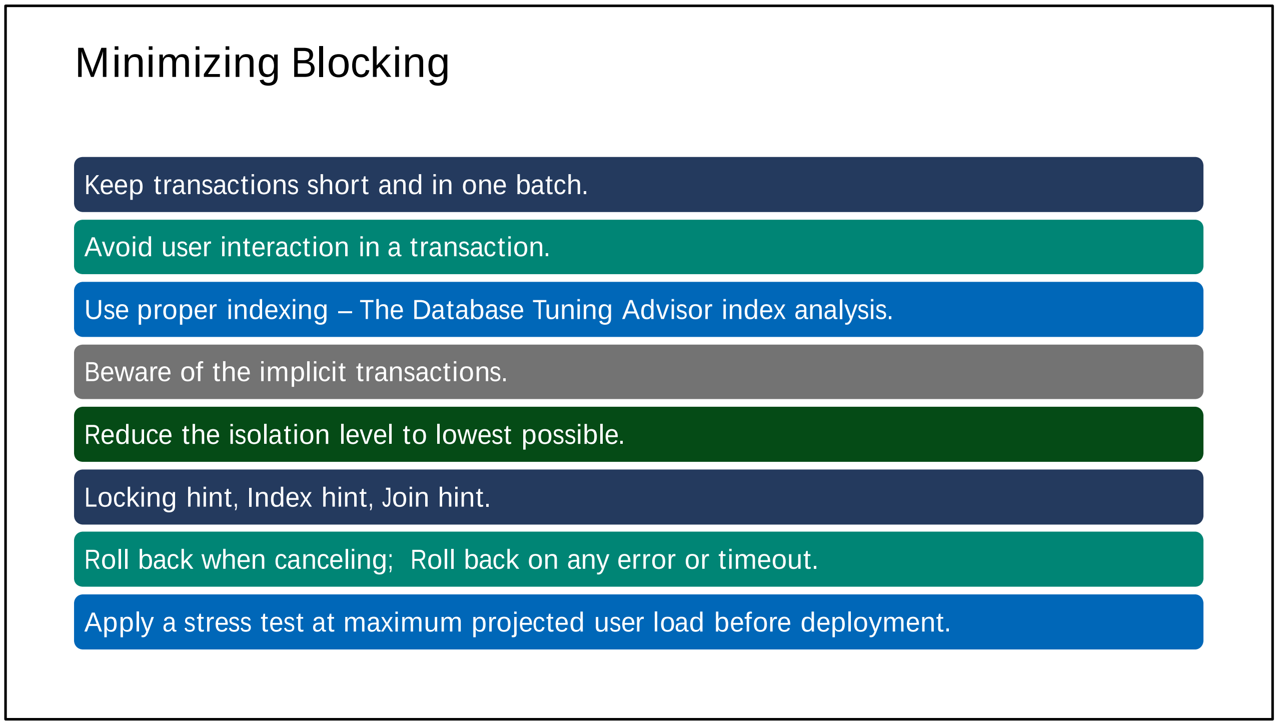
For queries executed within a transaction, the duration for which the locks are held are determined by the type of query, the transaction isolation level, and whether or not lock hints are used in the query.

When locking and blocking increase to the point where there is a detrimental effect on system performance, it is usually due to one of the following reasons:

- A SPID holds locks on a set of resources for an extended period of time before releasing them.

  This type of blocking resolves itself over time, but can cause performance degradation.
- A SPID holds locks on a set of resources and never releases them.

  This type of blocking does not resolve itself and prevents access to the affected resources indefinitely.



Minimizing Blocking

- Keep transactions short and in one batch.
- Avoid user interaction in a transaction.
- Use proper indexing – The Database Tuning Advisor index analysis.
- Beware of the implicit transactions.
- Reduce the isolation level to lowest possible.
- Locking hint, Index hint, Join hint.
- Roll back when canceling; Roll back on any error or timeout.
- Apply a stress test at maximum projected user load before deployment.

Short transactions

- Avoid user interaction in transactions, and keep transactions as short as possible and in a single batch.

Handling cancellations and errors with care

- If the application (or any stored procedures that the application invokes) starts any transactions, the application is responsible for rolling back any active transactions (IF @@TRANCOUNT>0 ROLLBACK TRAN) when any error or query timeout occurs. The application should not assume that an arbitrary error would abort the transaction. A query timeout never aborts transactions automatically.

Proper indexing

- On a table where inserts and updates are common, create a clustered index. Inserts into a heap should not cause a hotspot issue, but they can be much slower than inserts into the same table with a clustered index due to contention on Page Free Space (PFS) (longer transactions = greater chance of blocking). In general, the best column for a clustered index is often an identity column.
- Avoid placing a clustered index on columns that are frequently updated. Updates to clustered index key columns will require locks on the clustered index, data pages (to move the row), and all non-clustered indexes (because non-clustered indexes point to rows via the clustered index key). Consider using a clustered index on an identity column or a column with similar properties, such as small data values or constantly increasing values with very infrequent modifications.
- Analyze indexing for possible improvements. Poor indexes mean more pages visited, more locks acquired, and longer-running transactions (all these contribute to blocking, and in general, may increase the chances of deadlocking). Use the Database Engine Tuning Advisor (DTA) on related queries.
- Remove indexes that have extremely low selectivity. For example, if an indexed column on a large table only has 10 distinct values, a KEY lock on the index can effectively prevent updates to this column for 10 percent of the rows in the table.

Beware of implicit transactions

- With implicit transactions on, every time you make a modification, you get a transaction started automatically that you must commit yourself. Many modifications do not need to be part of a larger transaction. The improper use of implicit transaction tends to end up with two undesirable outcomes:

  - The developer forgets about the open transaction and leaves it open across modifications, which spans much more work than required by the business needs. Longer transactions than necessary mean more locks held, greater chance of deadlock or blocking, and so on.
  - The developer does the right thing and commits the implicit transaction after every unit of work. But this requires an extra round-trip to the server to send the COMMIT command, while autocommit (the opposite of implicit transactions) could have accomplished the same thing with a single query and a single round-trip. Also, because implicit transactions cause every command to open a transaction, the danger due to the coding mistakes that lead to orphaned transactions is increased.
- Therefore, it is important to be careful when using implicit transaction to avoid losing track of transaction nesting.

Use as low an isolation level as possible

- Higher transaction isolation levels (such as SERIALIZABLE) hold more locks for a longer period of time. Microsoft Transaction Server always sets the transaction isolation level to SERIALIZABLE. You can change this either by using the SET TRANSACTION ISOLATION LEVEL command or on a query-by-query basis with a READ COMMITTED query hint.

References:

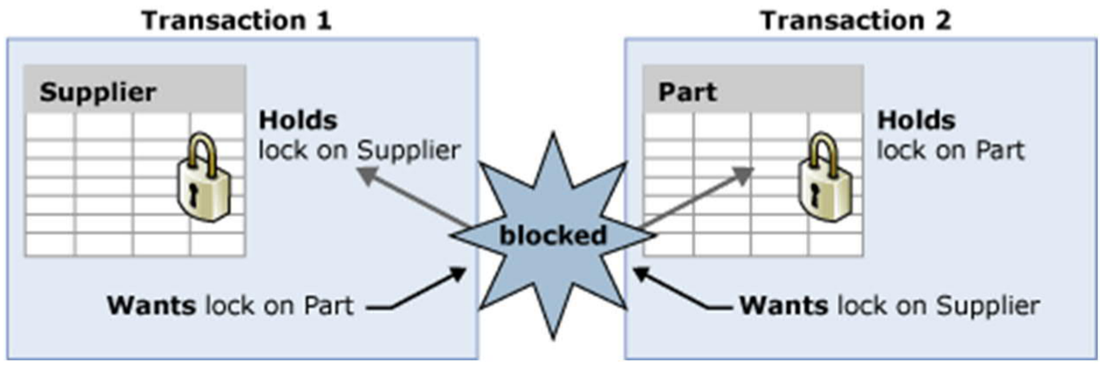
For more information, see: Minimize Blocking in SQL Server

https://docs.microsoft.com/en-us/previous-versions/technet-magazine/cc434694(v=msdn.10)?redirectedfrom=MSDN

For more information, see: SQL Server: Transaction Management

https://docs.microsoft.com/en-us/previous-versions/technet-magazine/hh750281(v=msdn.10)?redirectedfrom=MSDN

What Is a Deadlock?



What is a deadlock?

- A deadlock exists when the following four conditions exist simultaneously:
  - No pre-emption
  - Incremental allocation of resources (Locks are acquired as needed, rather than acquiring them all before beginning the transaction)
  - Incompatible use of resources (Here, compatibility refers to the one defined in our lock compatibility table)
  - Cycle 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.

Deadlocking is not specific to SQL Server or even to relational databases. Any system that must manage shared resources among multiple users, threads, or processes may be susceptible to deadlocks.

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:

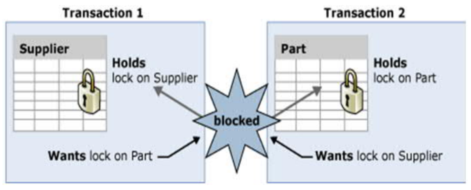
- Locks
- Worker threads
- Memory
- Parallel query execution-related resources
- Multiple Active Result Sets (MARS) resources

SQL Server always resolves deadlocking by choosing a victim and therefore, breaking the cycle. Most often, you can address deadlocking by using methods such as a nolock hint to address a deadlocking situation with item 2 in the list above (worker threads), you first need to understand what is being locked and in what order.

What is a deadlock?

Two or more processes are waiting for one another to obtain locked resources

- Transaction 1 holds a lock on the Supplier table and requires a lock on the Part table.
- Transaction 2 holds a lock on the Part table and wants a lock on the Supplier table
- Both tasks cannot continue until a resource is available and the resources cannot be released until a task continues, and therefore a deadlock state exists



Key Points:

- Occurs when two or more tasks permanently block each other by having a lock on a resource that the other tasks are trying to access.

- SQL Server always resolves deadlocking by choosing a victim and therefore, breaking the cycle. Most often, you can address deadlocking by using methods such as a nolock hint to address a deadlocking situation with item 2 in the list above (worker threads), you first need to understand what is being locked and in what order.
- Distributed deadlocks usually occur when one of the locked resources resides outside SQL Server (or on another SQL Server).

===============
More Information
===============

What is a deadlock?

A deadlock exists when the following four conditions exist simultaneously:

- No pre-emption.
- Incremental allocation of resources (Locks are acquired as needed, rather than acquiring them all before beginning the transaction).
- Incompatible use of resources (Here, compatibility refers to the one defined in our lock compatibility table).
- Cycle 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.

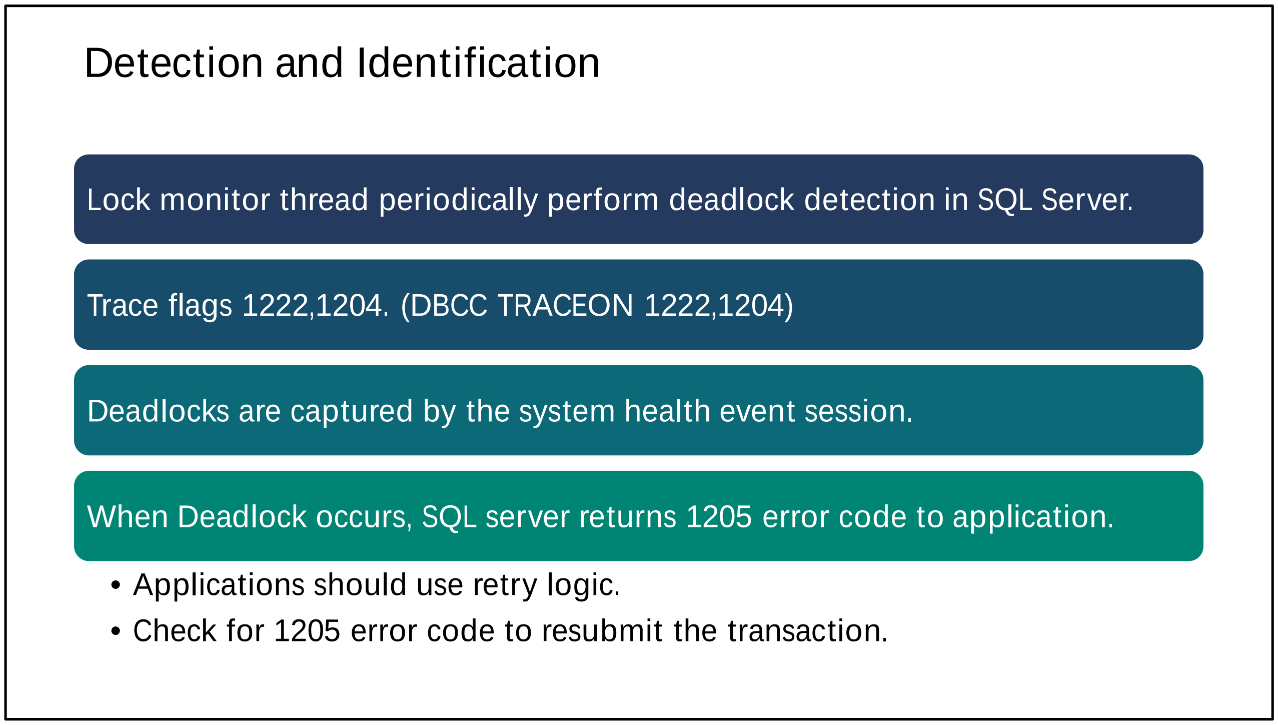
Deadlocking is not specific to SQL Server or even to relational databases. Any system that must manage shared resources among multiple users, threads, or processes may be susceptible to deadlocks.

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
- Worker threads
- Memory
- Parallel query execution-related resources
- Multiple Active Result Sets (MARS) resources

To understand this better, consider the following examples:

- You might address a deadlock between two tables by making sure that you access the two tables in the same order. A key-RID deadlock caused by a SELECT and UPDATE or two updates may be addressed either by converting a non-clustered index to a clustered index, where practical, or by using a lock earlier strategy in the transaction.
- A deadlock can also be caused by two updates that are expensive enough that they go parallel. Therefore, you do not have a guarantee that the two rows in the same table are accessed in the same order. You may resolve such a deadlock by adding an appropriate index so that the update can be performed more efficiently, without getting a parallel execution plan.
- A deadlock involving two rows on the same table with an application that uses positioned updates may be resolved by the use of an order by.



Detection and Identification

Lock monitor thread periodically perform deadlock detection in SQL Server.

Trace flags 1222,1204. (DBCC TRACEON 1222,1204)

Deadlocks are captured by the system health event session.

When Deadlock occurs, SQL server returns 1205 error code to application.

- Applications should use retry logic.
- Check for 1205 error code to resubmit the transaction.

Key points:

- Deadlock detection is performed by a lock monitor thread that periodically initiates a search through all of the tasks in an instance of the 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 Database Engine increases the intervals between searches to 5 seconds.
  - If a deadlock has just been detected, it is assumed that the next threads that must wait for a lock are entering the deadlock cycle.
  - The first couple of lock waits after a deadlock has been detected will immediately trigger a deadlock search rather than wait for the next deadlock detection interval. For example, if the current interval is 5 seconds, and a deadlock was just detected, the next lock wait will kick off the deadlock detector immediately. If this lock wait is part of a deadlock, it will be detected right away rather than during next deadlock search.
  - Profile – Deadlock graph report

===============

More Information

===============

The 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.

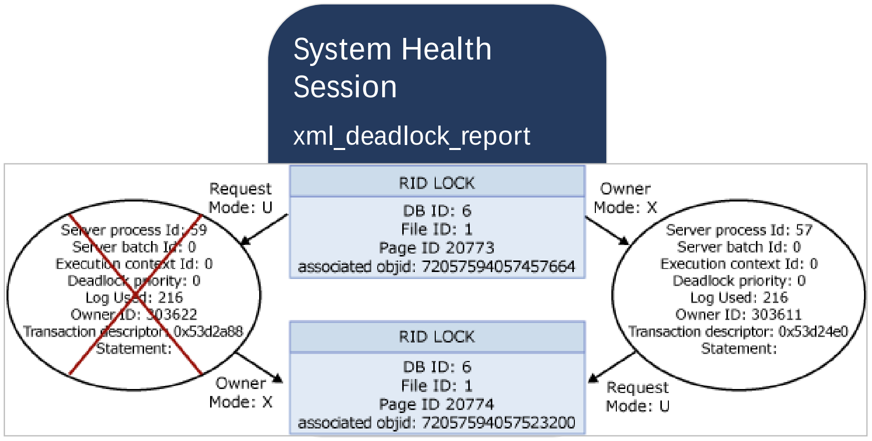
When the lock monitor initiates deadlock search for a particular thread, it identifies the resource on which the thread is waiting. The lock monitor then finds the owner(s) for that particular resource and recursively continues the deadlock search for those threads until it finds a cycle. A cycle identified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

After a deadlock is detected, the Database Engine ends a deadlock by choosing one of the threads as a deadlock victim. The Database Engine terminates the current batch being run 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.

Deadlock Analysis

Using System Health Xevent



To view deadlock information you can use system_health xevent session and Deadlock Graph tab.

Starting in Microsoft SQL Server 2008, the deadlock graph can be extracted from the system_health extended event session using the following query:

```
SELECT XEventData.XEvent.value('(data/value)[1]', 'varchar(max)') as DeadlockGraph
FROM
  (SELECT CAST(target_data as xml) as TargetData
  FROM sys.dm_xe_session_targets st
  JOIN sys.dm_xe_sessions s on s.address = st.event_session_address
  WHERE name = 'system_health') AS Data
  CROSS APPLY TargetData.nodes ('//RingBufferTarget/event') AS XEventData
  (XEvent)
WHERE XEventData.XEvent.value('@name', 'varchar(4000)') = 'xml_deadlock_report'
```

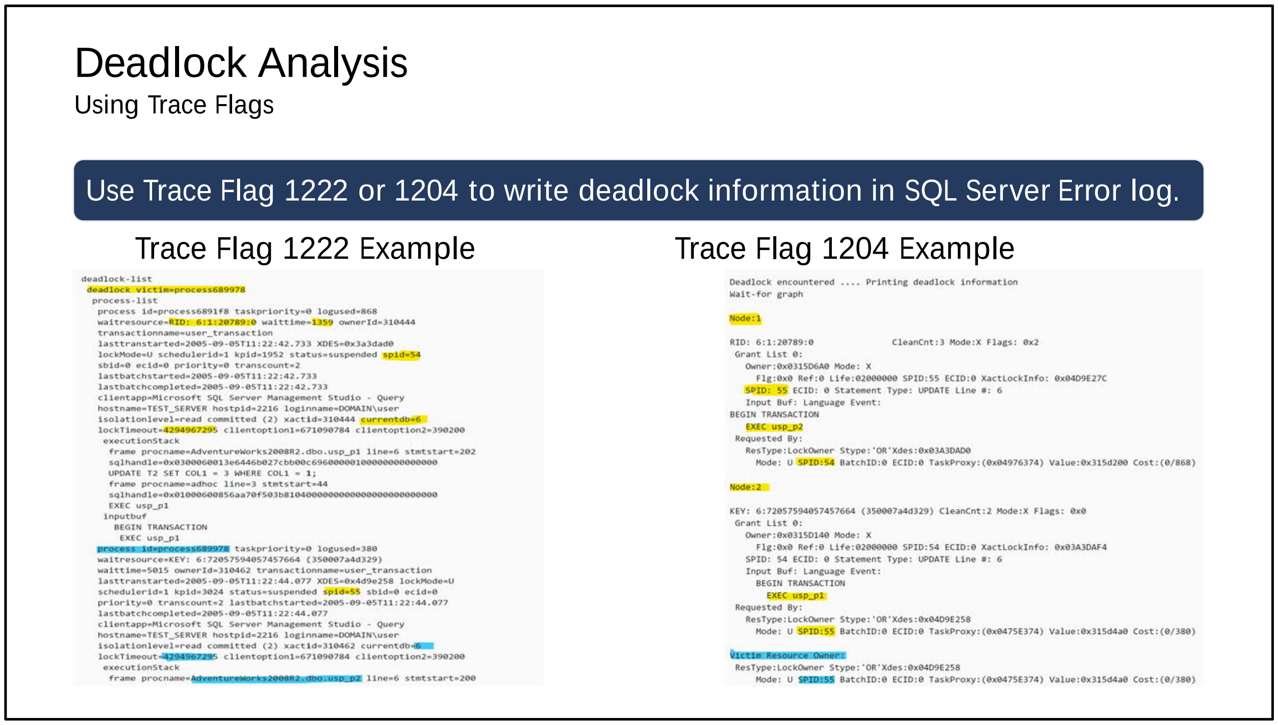
You can extract this data and save it as an .xdl file which can be viewed graphically in Microsoft SQL Server 2014 Management Studio. On a SQL Server 2014 instance, you can view the deadlock graph directly from SQL Server Management Studio without having to run the above query:

1. In the Object Explorer, go to Management > Extended Events > system_health.
2. Right-click package0.event_file and choose View Target Data. Scroll through the events until you find an xml_deadlock_report event.
3. Highlight the event and click the Deadlock tab to view the graph.

- This is an event in SQL Server Profiler that presents a graphical depiction of the tasks and resources involved in a deadlock.

References:

For more information, see: Detecting and Ending Deadlocks – https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/ms178104(v=sql.105)



Deadlock Analysis

Using Trace Flags

Use Trace Flag 1222 or 1204 to write deadlock information in SQL Server Error log.

Trace Flag 1222 Example

```
deadlock-list
 deadlock victim=process689978
  process-list
   process id=process6891f8 taskpriority=0 logused=868
   waitresource=RID: 6:1:20789:0 waittime=1359 ownerId=310444
   transactionname=user_transaction
   lasttranstarted=2005-09-05T11:22:42.733 XDES=0x3a3dad0
   lockMode=U schedulerid=1 kpid=1952 status=suspended spid=54
   sbid=0 ecid=0 priority=0 transcount=2
   lastbatchstarted=2005-09-05T11:22:42.733
   lastbatchcompleted=2005-09-05T11:22:42.733
   clientapp=Microsoft SQL Server Management Studio - Query
   hostname=TEST_SERVER hostpid=2216 loginname=DOMAIN\user
   isolationlevel=read committed (2) xactid=310444 currentdb=6
   lockTimeout=4294967295 clientoption1=671090784 clientoption2=390200
    executionStack
     frame procname=AdventureWorks2008R2.dbo.usp_p1 line=6 stmtstart=202
     sqlhandle=0x0300060013e6446b027cbb00c69600000100000000000000
     UPDATE T2 SET COL1 = 3 WHERE COL1 = 1;
     frame procname=adhoc line=3 stmtstart=44
     sqlhandle=0x01000600856aa70f503b8104000000000000000000000000
     EXEC usp_p1
    inputbuf
      BEGIN TRANSACTION
       EXEC usp_p1
   process id=process689978 taskpriority=0 logused=380
   waitresource=KEY: 6:72057594057457664 (350007a4d329)
   waittime=5015 ownerId=310462 transactionname=user_transaction
   lasttranstarted=2005-09-05T11:22:44.077 XDES=0x4d9e258 lockMode=U
   schedulerid=1 kpid=3024 status=suspended spid=55 sbid=0 ecid=0
   priority=0 transcount=2 lastbatchstarted=2005-09-05T11:22:44.077
   lastbatchcompleted=2005-09-05T11:22:44.077
   clientapp=Microsoft SQL Server Management Studio - Query
   hostname=TEST_SERVER hostpid=2216 loginname=DOMAIN\user
   isolationlevel=read committed (2) xactid=310462 currentdb=6
   lockTimeout=4294967295 clientoption1=671090784 clientoption2=390200
    executionStack
     frame procname=AdventureWorks2008R2.dbo.usp_p2 line=6 stmtstart=200
```

Trace Flag 1204 Example

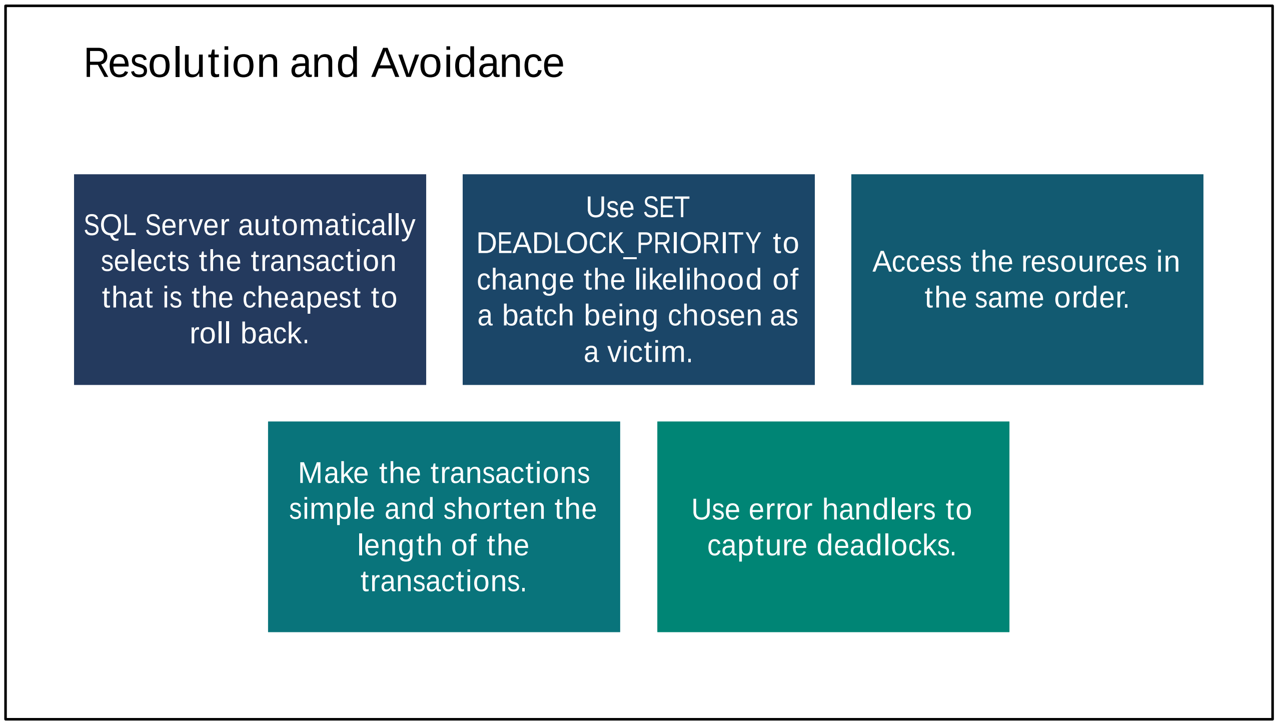
```
Deadlock encountered .... Printing deadlock information
Wait-for graph

Node:1

RID: 6:1:20789:0               CleanCnt:3 Mode:X Flags: 0x2
 Grant List 0:
   Owner:0x0315D6A0 Mode: X
     Flg:0x0 Ref:0 Life:02000000 SPID:55 ECID:0 XactLockInfo: 0x04D9E27C
   SPID: 55 ECID: 0 Statement Type: UPDATE Line #: 6
   Input Buf: Language Event:
BEGIN TRANSACTION
   EXEC usp_p2
 Requested By:
   ResType:LockOwner Stype:'OR'Xdes:0x03A3DAD0
     Mode: U SPID:54 BatchID:0 ECID:0 TaskProxy:(0x04976374) Value:0x315d200 Cost:(0/868)

Node:2

KEY: 6:72057594057457664 (350007a4d329) CleanCnt:2 Mode:X Flags: 0x0
 Grant List 0:
   Owner:0x0315D140 Mode: X
     Flg:0x0 Ref:0 Life:02000000 SPID:54 ECID:0 XactLockInfo: 0x03A3DAF4
   SPID: 54 ECID: 0 Statement Type: UPDATE Line #: 6
   Input Buf: Language Event:
     BEGIN TRANSACTION
       EXEC usp_p1
 Requested By:
   ResType:LockOwner Stype:'OR'Xdes:0x04D9E258
     Mode: U SPID:55 BatchID:0 ECID:0 TaskProxy:(0x0475E374) Value:0x315d4a0 Cost:(0/380)

Victim Resource Owner:
 ResType:LockOwner Stype:'OR'Xdes:0x04D9E258
     Mode: U SPID:55 BatchID:0 ECID:0 TaskProxy:(0x0475E374) Value:0x315d4a0 Cost:(0/380)
```

To view deadlock information, the Database Engine provides monitoring tools in the form of trace flag 1222 deadlock information in error log

1204 disclose the information at node level.

References:

For more information, see: Detecting and Ending Deadlocks - https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/ms178104(v=sql.105)



Resolution and Avoidance

SQL Server automatically selects the transaction that is the cheapest to roll back.

Use SET DEADLOCK_PRIORITY to change the likelihood of a batch being chosen as a victim.

Access the resources in the same order.

Make the transactions simple and shorten the length of the transactions.

Use error handlers to capture deadlocks.

Key points:

Use snapshot isolation i.e. optimistic locking

Although deadlocks cannot be completely avoided, following certain coding conventions can minimize the chance of generating a deadlock. Minimizing deadlocks can increase transaction throughput and reduce system overhead because fewer transactions are:

- Rolled back, undoing all the work performed by the transaction.
- Resubmitted by applications because they were rolled back when deadlocked.

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.

===============

More Information

===============

Access Objects in the Same Order

- If all concurrent transactions access objects in the same order, deadlocks are less likely to occur.

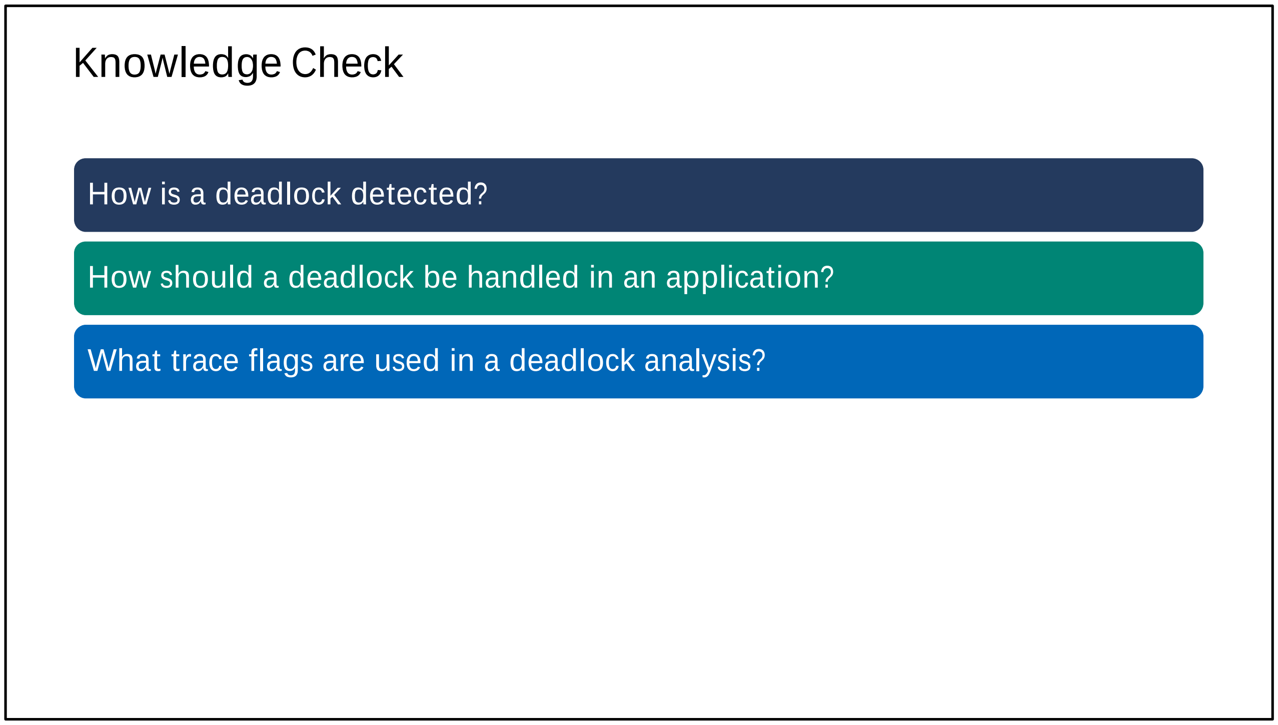
  For example, if two concurrent transactions obtain a lock on the Supplier table and then on the Part table, one transaction is blocked on the Supplier table until the other transaction is completed.

  After the first transaction commits or rolls back, the second continues, and a deadlock does not occur.

  Using stored procedures for all data modifications can standardize the order of accessing objects.

Avoid User Interaction in Transactions

- Avoid writing transactions that include user interaction, because the speed of batches running without user intervention is much faster than the speed at which a user must manually respond to queries, such as replying to a prompt for a parameter requested by an application. For example, if a transaction is waiting for user input and the user goes to lunch or even home for the weekend, the user delays the transaction from completing. This degrades system throughput because any locks held by the transaction are released only when the transaction is committed or rolled back. Even if a deadlock situation does not arise, other transactions accessing the same resources are blocked while waiting for the transaction to complete.



Knowledge Check

How is a deadlock detected?

How should a deadlock be handled in an application?

What trace flags are used in a deadlock analysis?

Q1: How is a deadlock detected?

A1: Deadlock detection is performed by a lock monitor thread that periodically initiates a search through all of the tasks in an instance of the Database Engine.

Q2: How should a deadlock be handled in an application?

A2: Because any application submitting Transact-SQL queries can be chosen as the deadlock victim, applications should have an error handler that can trap error message 1205. If an application does not trap the error, the application can proceed unaware that its transaction has been rolled back and errors can occur.

Q3: What trace flags are used in a deadlock analysis?

A3: 1204 (for Microsoft SQL Server 2000) and 1222 (for Microsoft SQL Server 2005 and later versions).

Q4: What is the Extended Event used in a deadlock analysis?

A4: xml_deadlock_report

Microsoft