**Course Introduction**

Monitoring transactions is a crucial part of database development and administration. SQL Server 2012 provides a variety of tools that enables you to retrieve information about the performance and behavior of the transactions that are running on your SQL Server. Using these tools in combination can provide an overall picture of the processes that are affecting your system.

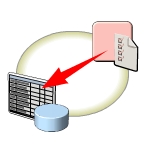
Furthermore, these tools will allow you to pinpoint areas of the SQL Server that may be having problems, and you can also gain more in depth information. These tools, combined with your knowledge on how the SQL Server is implemented, will enable you to solve problems efficiently. It will also enable you to identify areas that can be improved.

**Course Objectives**

At the end of this course, you will be able to:

* Describe transactions and lock.
* Manage transactions.
* Manage locks.
* Monitoring transaction and locks.
* Describe best practices for managing and monitoring transactions in Microsoft SQL Server 2012.

# Introducing Transactions and Locks



### Lesson Introduction

Transactions and locks interact together to affect the data integrity and concurrency of a database. Understanding these concepts are important for ensuring that the database performs efficiently and allows you to manage transactions and locks appropriately.

### Lesson Objectives

At the end of this lesson, you will be able to:

* Describe transactions.
* Describe transaction ACID properties.
* Describe how SQL Server modifies data in tables.
* Explain what is locking.

# Introduction to Transactions

A transaction is a sequence of Transact-SQL commands that are processed as a single logical unit. A command is deemed as a logical unit if it displays four ACID properties: Automicity, Consistency, Isolation, and Durability.

If the transaction issued is successful, then it is committed to the database. If the transaction fails, then it is rolled back. This will ensure that the database is left in a consistent state. These transactions can be held in a transaction log file, which holds a record of the transactions that have run against the database and potentially give you the ability to recover your database to a point in time.

There are different modes in which you can execute transactions. By default, SQL Server runs in autocommit mode. This means that all individual Transact-SQL statements are committed on completion. However, using explicit mode allows you to explicitly define BEGIN TRANSACTION, COMMIT TRANSACTION, and ROLLBACK TRANSACTION within your statements to give you greater control over when these statements can run.

**Concurrency**

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

Concurrency control theory has two classifications for the methods of instituting concurrency control. These classifications are as follows:

* **Pessimistic concurrency control**: A system of locks prevents users from modifying data in a way that affects other users. This is used where the cost of protecting data with locks is less than the cost of rolling back transactions if concurrency conflicts occur. All transaction isolation levels except Snapshot are pessimistic concurrency transaction isolation levels.
* **Optimistic concurrency control**: In optimistic concurrency control, the system does not lock data when it is read. This is used where the cost of occasionally rolling back a transaction is lower than the cost of locking data when read. Snapshot transaction isolation level is an optimistic transaction isolation level.

Microsoft SQL Server supports a range of concurrency control. Users specify the type of concurrency control by selecting transaction isolation levels for connections.

Transaction Isolation levels

Transactions specify an isolation level that defines the degree to which one transaction must be isolated from resource or data modifications made by other transactions. Isolation levels are described in terms of which concurrency side-effects, such as dirty reads, or phantom reads are allowed.

Transaction isolation levels control the following:

* Whether locks are taken when data is read and what type of locks are requested.
* How long the read locks are held.
* Blocks waiting lock requests until the exclusive lock on the row is freed.
* In snapshot isolation level, the system retrieves the committed version of the row that existed at the time the statement or transaction started.
* Can potentially read the uncommitted data modification.

Choosing a transaction isolation level does not affect the locks acquired to protect data modifications. A transaction always gets an exclusive lock on any data it modifies and holds that lock until the transaction completes, regardless of the isolation level set for that transaction.

A logical unit of work must exhibit four properties to qualify as a transaction, namely, atomicity, consistency, isolation, and durability. These properties are also commonly referred to as ACID.

Atomicity requires that each transaction must be an atomic unit of work, which means that all data modifications in a transaction are performed or none of the data modifications in a transaction are performed.

Consistency requires that after a transaction is completed, it must leave all the data in the database in a consistent state to maintain data integrity. For example, all internal data structures must be consistent at the end of the transaction.

Isolation requires that modifications made by a concurrent transaction must be isolated from the modifications made by any other concurrent transaction. Therefore, a transaction must use data either before or after another concurrent transaction modifies it and must never use data in an intermediate state.

Durability requires that after a transaction is completed, the modifications must persist permanently in the system, even if there is a system failure.

**How SQL Server Modifies Data in a Table**

SQL Server uses the deleted and inserted tables when executing Data Manipulation Language (DML) statements. DML trigger statements use the deleted and inserted tables to test the effects of certain data modifications or to set conditions for DML trigger actions. SQL Server creates and manages these temporary memory-resident tables automatically.

During an INSERT or UPDATE transaction, new rows are added simultaneously to both the table and the temporary inserted table. The rows in the inserted table are copies of the new rows in the table affected by the INSERT and UPDATE statements.

During a DELETE or UPDATE transaction, the rows affected by the DELETE or UPDATE statement are removed and transferred to the temporary deleted table. The table and the temporary deleted table affected by the INSERT or UPDATE statements generally have no rows in common.

An UPDATE transaction is similar to a DELETE operation followed by an INSERT operation. During an UPDATE transaction, the old rows are copied to the deleted table first, and then the new rows are copied to the tables and to the inserted table.

# What is Locking?



SQL Server is designed to handle multiple concurrent connections against a database allowing different types of resources to be locked by a transaction. When a connection reads or modifies a SQL Server object, a lock is acquired to ensure that the ACID properties are met. There is a possibility that separate connections may want to update the same object at the same time leading to a process known as deadlocks. Such a situation can cause concurrency issues within the database and reduce the performance of reading and making modification to data.

Locking is acquired on SQL Server objects when a connection is established on an object. There are different locking modes that affect the concurrency within the database. These modes range from shared locks that enable multiple connections to access the same object at the same time to exclusive locks, whereby a SQL Server object cannot be shared with other connections.

SQL Server locks resources automatically. Locking small objects, such as rows, increases concurrency but has a higher overhead because more locks must be held if many rows are locked. Locking larger objects, 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.

What type of lock means a SQL Server object cannot be shared with other connections?

Select the one best answer.



Shared.

That is not correct. A shared lock means a SQL Server object can be shared with other connections. The correct response is: Exclusive. An exclusive lock means a SQL Server object cannot be shared with other connections.



Exclusive.

That is correct. An exclusive lock means a SQL Server object cannot be shared with other connections.



Concurrent.

That is not correct. Concurrent describes multiple connections that can access a SQL Server object at the same time. The correct response is: Exclusive. An exclusive lock means a SQL Server object cannot be shared with other connections.



All of the above.

# Managing Transactions



### Lesson Introduction

There are a number of techniques that you can use to manage the transactions that are issued against your database. Understanding the types of transaction modes available will allow you to control how data is managed within a database. However, there are restrictions to the type of statements that are supported by transaction modes, and these statements must be considered.

### Lesson Objectives

At the end of this lesson, you will be able to:

* Describe transaction execution modes.
* Describe autocommit transaction mode.
* Describe explicit transaction mode.
* Describe implicit transaction mode.
* Describe guidelines for using transactions.

**Transaction Execution Modes**

Transactions in SQL Server can be explicit, implicit, or autocommit transactions.

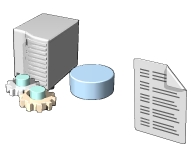
In explicit transactions, you need to explicitly define both the start and end of the transaction. Each transaction is explicitly started with the BEGIN TRANSACTION statement and explicitly ended with a COMMIT TRANSACTION or ROLLBACK TRANSACTION statement. Multiple statements for a transaction can be grouped into explicit transactions.

The default mode in SQL Server is autocommit transactions, where each individual T-SQL statement is committed when it is completed. However, if the statement encounters any error, then it is rolled back.

SQL Server also supports implicit transactions, where it automatically starts a new transaction after the current transaction is committed or rolled back. A new transaction is implicitly started when the prior transaction completes, but each transaction is explicitly completed with a COMMIT TRANSACTION or ROLLBACK TRANSACTION statement.

Each SQL statement such as an INSERT, UPDATE, or DELETE is an implicit or automatic transaction.

# Autocommit Transaction Mode



A connection to the Database Engine operates in autocommit mode until a BEGIN TRANSACTION statement starts an explicit transaction, or implicit transaction is set on. When the explicit transaction is committed or rolled back, or when implicit transaction mode is turned off, the connection returns to autocommit mode.

Within autocommit mode, when compile errors are encountered, it sometimes appears as if an instance of the Database Engine has rolled back an entire batch instead of just one SQL statement. A compile error prevents the Database Engine from building an execution plan, so nothing in the batch is executed. As a result, the error prevents anything in the batch from being executed.

In the following example, none of the INSERT statements in the third batch are executed because of a **compile** error. It appears that the first two INSERT statements are rolled back when they are never executed.

USE AdventureWorks2012R2;

GO

CREATE TABLE Employee (EmployeeID INT PRIMARY KEY, EmployeeBadge CHAR(3));

GO

INSERT INTO Employee VALUES (1, 'aaa');

INSERT INTO Employee VALUES (2, 'bbb');

INSERT INTO Employee **VALUSE** (3, 'ccc');  -- Syntax error.

GO

SELECT \* FROM Employee;  -- Returns no rows.

GO

In the following example, the third INSERT statement generates a run-time duplicate primary key error. The first two INSERT statements are successful and committed, so they remain after the run-time error.

USE AdventureWorks2012R2;

GO

CREATE TABLE Employee (EmployeeID INT PRIMARY KEY, EmployeeBadge CHAR(3));

GO

INSERT INTO Employee VALUES (1, 'aaa');

INSERT INTO Employee VALUES (2, 'bbb');

INSERT INTO Employee **VALUES** (3, 'ccc');  -- Syntax error.

GO

SELECT \* FROM Employee;  -- Returns row 1 and 2.

GO

# Explicit Transaction Mode

When controlling transactions in your Transact-SQL code, you can use the BEGIN, SAVE, COMMIT, and ROLLBACK TRANSACTION clause. This provides the opportunity to use these clauses within the logic of your Transact-SQL statement. For example, you may have a transaction that updates one table and then updates another table based on the information of the first update. If the first update statement succeeds and the second update fails, you could use a ROLLBACK TRANSACTION statement that ensures that the data is not committed to the database.

BEGIN TRANSACTION

Explicit transactions start with the BEGIN TRANSACTION statement. It is not necessary to specify this when SQL Server is running in auto commit mode.

SAVE TRANSACTION

You can place the SAVE TRANSACTION statement if you wish to mark the transaction with a save point. The savepoint defines a location to which a transaction can return, if part of the transaction is conditionally canceled.

COMMIT TRANSACTION

The COMMIT TRANSACTION statement will permanently commit a successful transaction to the database. It also frees the transaction from SQL Server resources that can be used by other transactions.

ROLLBACK TRANSACTION

The ROLLBACK TRANSACTION statement will cancel a transaction and return the data back to the state that it was in prior to the transaction being issued.

# Implicit Transaction Mode

When operating in implicit transaction mode, SQL Server 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. The transaction remains in effect until you issue a COMMIT or ROLLBACK statement. After the first transaction is committed or rolled back, SQL Server automatically starts a new transaction the next time any of these statements are executed by the connection.

Implicit transactions must be turned on with the following statement.

SET IMPLICIT\_TRANSACTIONS ON;

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 the following statements.

|  |  |  |
| --- | --- | --- |
| T-SQL statement | T-SQL statement | T-SQL statement |
| ALTER TABLE | UPDATE | GRANT |
| INSERT | FETCH | TRUNCATE TABLE |
| CREATE | OPEN | DELETE |
| REVOKE | DROP | SELECT |

# Transaction Guidelines



The following are guidelines for using transactions within Microsoft SQL Server 2012.

* Create transactions that perform one task.
* For greater control of transactions, use explicit transactions.
* Keep transactions short to limit the amount of locking on database objects.
* Limit the number of rows read within a transaction as exclusive locks are acquired.
* Remember that the following commands cannot be executed within a transaction.
  + CREATE DATABASE
  + ALTER DATABASE
  + DROP DATABASE
  + LOAD DATABASE
  + LOAD TRANSACTION
  + BACKUP LOG
  + BACKUP DATABASE
  + RESTORE LOG
  + RESTORE DATABASE
  + RECONFIGURE
  + UPDATE STATISTICS
  + DISK INIT

You are a database developer for Adventure Works. You are developing a database. You want to issue a transaction that may damage the database. You want to ensure that you can roll back to the transaction should a problem occur. Which statement should you use?

Select the one best answer.



BEGIN TRANSACTION.

That is not correct. BEGIN TRANSACTION starts a transaction. The correct response is: SAVE TRANSACTION. This statement will mark the transaction in the transaction log that can be rolled back.



COMMIT TRANSACTION.

That is not correct. COMMIT TRANSACTION commits a transaction to the database. The correct response is: SAVE TRANSACTION. This statement will mark the transaction in the transaction log that can be rolled back.



SAVE TRANSACTION.

That is correct. SAVE TRANSACTION will mark the transaction in the transaction log that can be rolled back.



ROLLBACK TRANSACTION.

# Managing Locks



### Lesson Introduction

Locks are an internal mechanism of Microsoft SQL Server 2012 that ensures the integrity of the data whilst at the same time allowing concurrent access. Understanding how locks operate in SQL Server is fundamental information when you embark upon performance tuning operations.

### Lesson Objectives

At the end of this lesson, you will be able to:

* Describe the concurrency problems.
* Identify lockable resources.
* Describe the types of locks.
* Describe lock compatibility.
* Describe transaction isolation levels.
* Describe query hints and plan guides.

# Database Concurrency Problems

Without transaction isolation levels being implemented against a database, different types of problems can occur with the data that can affect the results that are returned to the result set. The combination of applications modifying data in the database and applications reading data can lead to the following concurrency problems.

**Lost Updates**

This problem occurs when two separate applications, issue an update transaction to the same data on the same row, on the value originally selected. The last update that is committed is applied and overwrites the update made by another transaction. This results in a lost update.

**Dirty Reads**

This problem occurs when two separate applications, one issuing a modification statement and the other reading the data, work on the same row of data. Changes occur to the data that has not been committed to the database. In this scenario, it is possible that the data being read is not consistent with the change being made. This results in a dirty read.

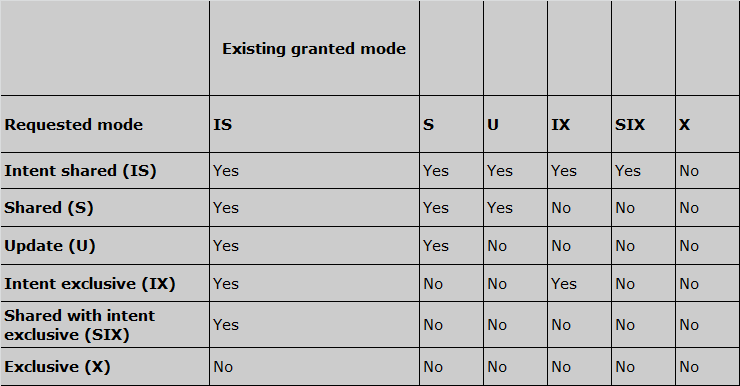
**Non Repeatable Read**

This problem occurs when two separate applications work on the same row of data. One application may read the same row of data more than once. In between the reads, a second application issues a transaction that modifies the data in the row between reads. This can cause inconsistent analysis. This results in a non repeatable read.

**Phantom Reads**

This problem occurs when one application will issue a transaction that will insert or delete a row of data that is part of a range of rows that is read by a second application. This will lead to data that is inconsistent. This results in a phantom read.

# 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.

The table above shows the locking compatibility between the different types of locks.

# Transaction Isolation Levels

Microsoft SQL Server 2012 is a multi user system that can handle thousands of connections against a single database. These connections will consist of transactions that need to be managed.

One aspect of SQL Server that can have an impact on this is transaction isolation levels. This controls the locking behavior of your transactions against the database. One of five transaction isolation levels can be used. Each has an effect on both the integrity of the data that you are working with and the speed with which you can access the data.

The choice of transaction isolation levels that you use is dependant upon the business requirement to have data as fast as possible or as clean as possible. The transaction isolation levels can be set as a database option. However, you can also define a different transaction isolation level within a transaction just for the duration that the transaction runs.

For example, the default transaction isolation level for a SQL Server 2012 database is READ COMMITTED. This may be desirable for the majority of transactions that are issued against a database. However, this transaction isolation level can cause lost updates. If you have a transaction that cannot afford this concurrency issue, you can the change the transaction isolation level to one to does not suffer from lost updates, such as SERIALIZABLE.

Understanding the different transaction isolation levels that can be set will help you to understand the impact it has on database integrity and performance. The following table describes the different transaction isolation levels.

READ UNCOMMITTED

At the READ UNCOMMITTED level, SQL Server does not request any locks while reading a row and does not honor any existing locks. You can read rows with values that have been previously updated but not yet committed. Lower level isolations such as the READ UNCOMMITTED level increase the ability of many users to access data at the same time. But it also increases the number of concurrency effects, such as dirty reads or lost updates that users might encounter. You need to be careful when using the READ UNCOMMITTED level as it allows dirty reads and therefore, it is not recommended.

READ COMMITTED

At the READ COMMITTED level, SQL Server acquires a shared lock while reading data, but frees the lock immediately after reading the data. Because shared lock requests are blocked by an exclusive lock, it is not possible to read rows that have been updated by another task but have not yet been committed. READ COMMITTED is the default isolation level setting for both SQL Server and ODBC. This level solves concurrency issues such as dirty reads.

REPEATABLE READ

At the REPEATABLE READ level, SQL Server 2012 requests a shared lock on each row as it is read in the READ COMMITTED level. However, if the row is accessed within a transaction, the shared locks are held until the end of the transaction, instead of being freed after the row is read. This has the same effect as specifying the HOLDLOCK operator in a SELECT statement. The REPEATABLE READ level solves the concurrency issue of dirty reads and non repeatable reads.

SERIALIZABLE

SERIALIZABLE transaction isolation level specifies the following:

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

Range locks are placed in the range of key values that match the search conditions of each statement executed in a transaction. This blocks other transactions from updating or inserting any rows that would qualify for any of the statements executed by the current transaction. This means that if any of the statements in a transaction are executed a second time, they will read the same set of rows. The range locks are held until the transaction completes. This is the most restrictive of the isolation levels because it locks entire ranges of keys and holds the locks until the transaction completes.

Because concurrency is lower, use this option only when necessary.

SNAPSHOT

SNAPSHOT transaction isolation level specifies that data read by any statement in a transaction will be a transactionally consistent version of the data that existed at the start of the transaction. The version store is maintained in tempdb that holds the last consistent values of a record before the transaction started. The transaction can only recognize data modifications that were committed before the start of the transaction. Data modifications made by other transactions after the start of the current transaction are not visible to statements executing in the current transaction. The effect is as if the statements in a transaction get a snapshot of the committed data as it existed at the start of the transaction. Snapshot Isolation solves the pessimistic concurrency issue of readers blocking writers.

The ALLOW\_SNAPSHOT\_ISOLATION database option must be set to ON before you can start a transaction that uses the SNAPSHOT isolation level. If a transaction using the SNAPSHOT isolation level accesses data in multiple databases, ALLOW\_SNAPSHOT\_ISOLATION must be set to ON in each database.

A transaction cannot be set to SNAPSHOT isolation level that started with another isolation level; doing so will cause the transaction to abort. If a transaction starts in the SNAPSHOT isolation level, you can change it to another isolation level, and then back to SNAPSHOT. A transaction starts the first time it accesses data.

A transaction running under SNAPSHOT isolation level can view changes made by that transaction. For example, if the transaction performs an UPDATE on a table, and then issues a SELECT statement against the same table, the modified data will be included in the result set.

The following is a code example of how to change the transaction isolation levels in a transaction.

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

GO

BEGIN TRANSACTION;

GO

SELECT \*     FROM HumanResources.EmployeePayHistory;

COMMIT TRANSACTION;

GO

## **Query Hints**

Query hints can be embedded within transactions to dictate how the locking behavior should occur within the transaction. This will ignore the SQL Server query optimizer best execution plan for a transaction. Query hints should therefore be used with care and should be continually reviewed. A query hint that may be appropriate for transactions today could become obsolete as the data changes within the database.

Query hints can be applied to all DML operations, including Merge. The following example retrieves all rows from the Person.Person table using a TABLOCK query hint that forces the query optimizer to lock the entire table.

## SELECT \* FROM Person.Person WITH TABLOCK

Query hints specify that the indicated hints should be used throughout the query. Query hints affect all operators in the statement. If UNION is involved in the main query, only the last query involving a UNION operation can have the OPTION clause. Query hints are specified as part of the OPTION clause. If one or more query hints cause the query optimizer not to generate a valid plan, error 8622 is raised.

**Transact-SQL Syntax Conventions**

**Syntax**

|  |
| --- |
| **<query\_hint > ::=**  { { HASH | ORDER } GROUP    | { CONCAT | HASH | MERGE } UNION    | { LOOP | MERGE | HASH } JOIN    | FAST *number\_rows*    | FORCE ORDER    | MAXDOP *number\_of\_processors*  | OPTIMIZE FOR ( *@variable\_name* { UNKNOWN | = *literal\_constant }* [ **,** ...*n* ] )  | OPTIMIZE FOR UNKNOWN  | PARAMETERIZATION { SIMPLE | FORCED }  | RECOMPILE    | ROBUST PLAN    | KEEP PLAN    | KEEPFIXED PLAN    | EXPAND VIEWS    | MAXRECURSION *number*  | USE PLAN N**'***xml\_plan***'**  | TABLE HINT **(** *exposed\_object\_name* [ **,** <table\_hint> [ [**,** ]...*n* ] ] **)**  **<table\_hint> ::=**  [ NOEXPAND ] {      INDEX **(** *index\_value* [ **,**...*n* ] **)** | INDEX = **(** *index\_value* **)**    | FASTFIRSTROW    | FORCESEEK    | HOLDLOCK    | NOLOCK  | NOWAIT    | PAGLOCK    | READCOMMITTED    | READCOMMITTEDLOCK    | READPAST    | READUNCOMMITTED    | REPEATABLEREAD    | ROWLOCK    | SERIALIZABLE    | TABLOCK    | TABLOCKX    | UPDLOCK    | XLOCK  } |

The following table describes the different arguments.

|  |  |
| --- | --- |
| **Argument** | **Description** |
| { HASH | ORDER } GROUP | Specifies that aggregations described in the GROUP BY, DISTINCT, or COMPUTE clause of the query should use hashing or ordering. |
| { MERGE | HASH | CONCAT } UNION | Specifies that all UNION operations are performed by merging, hashing, or concatenating UNION sets. If more than one UNION hint is specified, the query optimizer selects the least expensive strategy from those hints specified. |
| { LOOP | MERGE | HASH } JOIN | Specifies that all join operations are performed by LOOP JOIN, MERGE JOIN, or HASH JOIN in the whole query. If more than one join hint is specified, the optimizer selects the least expensive join strategy from the allowed ones.  If in the same query, a join hint is also specified in the FROM clause for a specific pair of tables. This join hint takes precedence in the joining of the two tables, although the query hints still must be honoured. Therefore, the join hint for the pair of tables may only restrict the selection of allowed join methods in the query hint. |
| FAST *number\_rows* | Specifies that the query is optimized for fast retrieval of the first *number\_rows.* This is a non negative integer. After the first *number\_rows* are returned, the query continues execution and produces its full result set. |
| FORCE ORDER | Specifies that the join order indicated by the query syntax is preserved during query optimization. Using FORCE ORDER does not affect possible role reversal behaviour of the query optimizer.   |  |  | | --- | --- | | Note | **Note**  In a MERGE statement, the source table is accessed before the target table as the default join order, unless the WHEN SOURCE NOT MATCHED clause is specified. Specifying FORCE ORDER preserves this default behavior. | |
| MAXDOP *number* | Overrides the **max degree of parallelism** configuration option of **sp\_configure** and Resource Governor for the query specifying this option. The MAXDOP query hint can exceed the value configured with **sp\_configure**. If MAXDOP exceeds the value configured with Resource Governor, the Database Engine uses the Resource Governor MAXDOP value, described in ALTER WORKLOAD GROUP (Transact-SQL). All semantic rules used with the **max degree of parallelism** configuration option are applicable when you use the MAXDOP query hint. |
| OPTIMIZE FOR ( *@variable\_name* { UNKNOWN | = *literal\_constant }* [ **,** ...*n* ] ) | Instructs the query optimizer to use a particular value for a local variable when the query is compiled and optimized. The value is used only during query optimization, and not during query execution.  *@variable\_name*  Is the name of a local variable used in a query, to which a value may be assigned for use with the OPTIMIZE FOR query hint.  *UNKNOWN*  Specifies that the query optimizer use statistical data instead of the initial value to determine the value for a local variable during query optimization.  *literal\_constant*  Is a literal constant value that can be assigned @variable\_name in conjunction with the OPTIMIZE FOR query hint. literal\_constant is used only during query optimization, and not as the value of @variable\_name during query execution. literal\_constant can be of any SQL Server system data type that can be expressed as a literal constant. The data type of literal\_constant must be implicitly convertible to the data type that @variable\_name references in the query.  OPTIMIZE FOR can counteract the default parameter detection behaviour of the optimizer or can be used when you create plan guides. |
| OPTIMIZE FOR UNKNOWN | Instructs the query optimizer to use statistical data instead of the initial values for all local variables when the query is compiled and optimized, including parameters created with forced parameterization.  If OPTIMIZE FOR @variable\_name = *literal\_constant* and OPTIMIZE FOR UNKNOWN are used in the same query hint, the query optimizer will use the *literal\_constant* that is specified for a specific value and UNKNOWN for the remaining variable values. The values are used only during query optimization, and not during query execution. |
| PARAMETERIZATION { SIMPLE | FORCED } | Specifies the parameterization rules that the SQL Server query optimizer applies to the query when it is compiled.   |  |  | | --- | --- | | clip_image001 | **Important**  The PARAMETERIZATION query hint can only be specified inside a plan guide. It cannot be specified directly within a query. |   SIMPLE instructs the query optimizer to attempt simple parameterization. FORCED instructs the optimizer to attempt forced parameterization. The PARAMETERIZATION query hint is used to override the current setting of the PARAMETERIZATION database SET option inside a plan guide. |
| RECOMPILE | Instructs the SQL Server Database Engine to discard the plan generated for the query after it executes, forcing the query optimizer to recompile a query plan the next time the same query is executed. Without specifying RECOMPILE, the Database Engine caches query plans and reuses them. When compiling query plans, the RECOMPILE query hint uses the current values of any local variables in the query and, if the query is inside a stored procedure, the current values passed to any parameters.  RECOMPILE is a useful alternative to creating a stored procedure that uses the WITH RECOMPILE clause when only a subset of queries inside the stored procedure, instead of the whole stored procedure, must be recompiled. RECOMPILE is also useful when you create plan guides. |
| ROBUST PLAN | Forces the query optimizer to try a plan that works for the maximum potential row size, possibly at the expense of performance. When the query is processed, intermediate tables and operators may have to store and process rows that are wider than any one of the input rows. The rows may be so wide that, sometimes, the particular operator cannot process the row. If this occurs, the Database Engine produces an error during query execution. By using ROBUST PLAN, you instruct the query optimizer not to consider any query plans that may encounter this problem.  If such a plan is not possible, the query optimizer returns an error instead of deferring error detection to query execution. Rows may contain variable-length columns; the Database Engine allows for rows to be defined that have a maximum potential size beyond the ability of the Database Engine to process them. Generally, despite the maximum potential size, an application stores rows that have actual sizes within the limits that the Database Engine can process. If the Database Engine encounters a row that is too long, an execution error is returned. |
| KEEP PLAN | Forces the query optimizer to relax the estimated recompile threshold for a query. The estimated recompile threshold is the point at which a query is automatically recompiled when the estimated number of indexed column changes have been made to a table by running UPDATE, DELETE, MERGE, or INSERT statements. Specifying KEEP PLAN makes sure that a query will not be recompiled as frequently when there are multiple updates to a table. |
| KEEPFIXED PLAN | Forces the query optimizer not to recompile a query due to changes in statistics. Specifying KEEPFIXED PLAN makes sure that a query will be recompiled only if the schema of the underlying tables is changed or if **sp\_recompile** is executed against those tables. |
| EXPAND VIEWS | Specifies that the indexed views are expanded and the query optimizer will not consider any indexed view as a substitute for any part of the query. A view is expanded when the view name is replaced by the view definition in the query text.  This query hint virtually disallows direct use of indexed views and indexes on indexed views in the query plan.  The indexed view is not expanded only if the view is directly referenced in the SELECT part of the query and WITH (NOEXPAND) or WITH (NOEXPAND, INDEX( *index\_value* [ **,***...n* ] ) ) is specified.  Only the views in the SELECT part of statements, including those in INSERT, UPDATE, MERGE, and DELETE statements are affected by the hint. |
| MAXRECURSION *number* | Specifies the maximum number of recursions allowed for this query. *number* is a non-negative integer between 0 and 32767. When 0 is specified, no limit is applied. If this option is not specified, the default limit for the server is 100.  When the specified or default number for MAXRECURSION limit is reached during query execution, the query is ended and an error is returned.  Because of this error, all effects of the statement are rolled back. If the statement is a SELECT statement, partial results or no results may be returned. Any partial results returned may not include all rows on recursion levels beyond the specified maximum recursion level. |
| USE PLAN N**'***xml\_plan***'** | Forces the query optimizer to use an existing query plan for a query that is specified by **'***xml\_plan***'**. USE PLAN cannot be specified with INSERT, UPDATE, MERGE, or DELETE statements. |
| TABLE HINT **(** *exposed\_object\_name* [ **,** <table\_hint> [ [**,** ]...*n* ] ] **)** | Applies the specified table hint to the table or view that corresponds to *exposed\_object\_name*. We recommend using a table hint as a query hint only in the context of a plan guide.  *exposed\_object\_name* can be one of the following references:   * When an alias is used for the table or view in the FROM clause of the query, *exposed\_object\_name* is the alias. * When an alias is not used, *exposed\_object\_name* is the exact match of the table or view referenced in the FROM clause. For example, if the table or view is referenced using a two-part name, *exposed\_object\_name* is the same two-part name.   When *exposed\_object\_name* is specified without also specifying a table hint, any indexes specified in the query as part of a table hint for the object are disregarded and index usage is determined by the query optimizer. You can use this technique to eliminate the effect of an INDEX table hint when you cannot modify the original query. |
| **<table\_hint> ::=** { [ NOEXPAND ] { INDEX ( *index\_value* [ ,...*n* ] ) | INDEX = ( *index\_value* ) | FASTFIRSTROW | FORCESEEK | HOLDLOCK | NOLOCK | NOWAIT | PAGLOCK | READCOMMITTED | READCOMMITTEDLOCK | READPAST | READUNCOMMITTED | REPEATABLEREAD | ROWLOCK | SERIALIZABLE | TABLOCK | TABLOCKX | UPDLOCK | XLOCK } | Is the table hint to apply to the table or view that corresponds to *exposed\_object\_name* as a query hint. For a description of these hints, see Table Hints (Transact-SQL).  Table hints other than INDEX and FORCESEEK are disallowed as query hints unless the query already has a WITH clause specifying the table hint. |

You have had a request that all data that is returned from the database should not contain data that may result from dirty reads. Which Transaction Isolation Level would you NOT use to achieve the requirement?

Select the one best answer.



READ UNCOMMITTED.

That is correct. The nature of the READ UNCOMMITTED Transaction Isolation Level means that there could be dirty reads.



READ COMMITTED.

That is not correct. READ COMMITTED does not suffer from dirty reads. The correct response is: READ UNCOMMITTED. The nature of the READ UNCOMMITTED Transaction Isolation Level means that there could be dirty reads.



SERIALIZABLE.

That is not correct. SERIALIZABLE does not suffer from dirty reads. The correct response is: READ UNCOMMITTED. The nature of the READ UNCOMMITTED Transaction Isolation Level means that there could be dirty reads.



REPEATABLE READ.

You have had a request that you should maximize the concurrent access to the database. Which Transaction Isolation Level would you use to achieve the requirement?

Select the one best answer.



REPEATABLE READ.

That is not the correct answer. The correct answer is 4. The nature of the READ UNCOMMITTED Transaction Isolation level means that it allows more concurrent database access. REPEATABLE READ is more restrictive in terms of concurrent database access.



SERIALIZABLE.

That is not the correct answer. The correct answer is 4. The nature of the READ UNCOMMITTED Transaction Isolation level means that it allows more concurrent database access. SERIALIZABLE is the most restrictive in terms of concurrent database access.



READ COMMITED.

That is not the correct answer. The correct answer is 4. The nature of the READ UNCOMMITTED Transaction Isolation level means that it allows more concurrent database access. READ COMMITTED is more restrictive in terms of concurrent database access.



READ UNCOMMITTED.

You need to change the Transaction Isolation Level for a particular transaction to SERIALIZABLE. What should you do?

Select the one best answer.



Change the transaction Isolation Level in the database properties of the SQL Server.

That is not correct. Setting the Transaction Isolation Level at the database level will affect all transactions. The correct response is: In the transaction, begin with the statement: SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;. This statement will run that transaction at SERIALIZABLE TRANSACTION ISOLATION LEVEL.



Do nothing. SERIALIZABLE transaction Isolation Level is the SQL Server default Isolation Level.

That is not correct. The default Transaction Isolation Level for a SQL Server database is Read Committed. The correct response is: In the transaction, begin with the statement: SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;. This statement will run that transaction at SERIALIZABLE TRANSACTION ISOLATION LEVEL.



In the transaction, begin with the statement: SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;.

That is correct. Beginning your transaction SET TRANSACTION ISOLATION LEVEL SERIALIZABLE; will run that transaction at SERIALIZABLE TRANSACTION ISOLATION LEVEL.



In the transaction, begin with the statement: TRANSACTION ISOLATION LEVEL = SERIALIZABLE;.

Which SQL Server resource is not lockable?

Select the one best answer.



Database.

That is not correct. A database is a lockable resource. The correct response is: Filegroup. This is not a lockable resource.



Table.

That is not correct. A table is a lockable resource. The correct response is: Filegroup. This is not a lockable resource.



Filegroup.

That is correct. A filegroup is not a lockable resource.



File.

You are a database administrator for Adventure Works. You want to optimize the performance of a poor running query that is generated by a third party application without directly modifying the code. What tool can be used within SQL Server to optimize the query?

Select the one best answer.



Query Hints.

That is not correct. Query hints would require you to modify the code. The correct response is: Plan guides. This tool can be used to optimize the performance of a poor running query without directly modifying the code.



Read uncommitted.

That is not correct. Read uncommitted is a transaction Isolation Level. The correct response is: Plan guides. This tool can be used to optimize the performance of a poor running query without directly modifying the code.



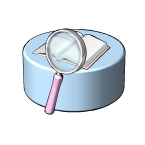
Plan guides.

That is correct. Plan guides can be used to optimize the performance of a poor running query without directly modifying the code.



All of the above.

# Monitoring Transactions and Locks



### Lesson Introduction

There are a number of Microsoft SQL Server tools that allow you to monitor transactions and locks that occur on your database. These tools can be used to view the locking activity occurring in the database and end processes that are preventing a database from functioning properly. You can also use tools to set up proactive monitoring.

### Lesson Objectives

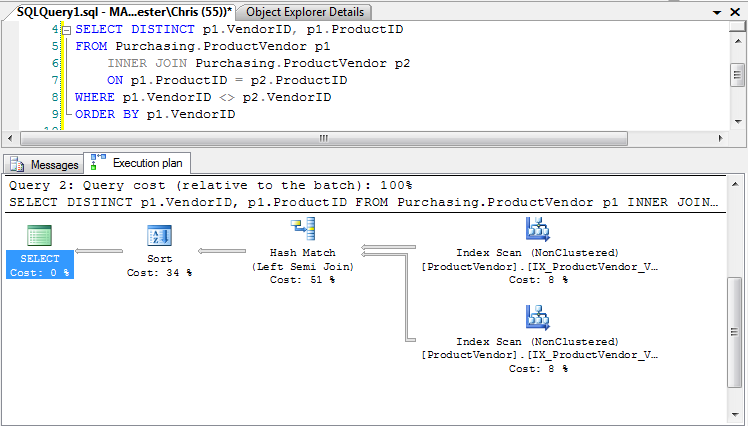
At the end of this lesson, you will be able to:

* Describe Activity Monitor.
* Describe execution plans.
* Describe Dynamic management views.
* Describe SQL Server Profiler.
* Use Reliability and Performance Monitor.

# Activity Monitor

When you need to monitor the connections of SQL Server 2012 in real time and assess the impact of the connections, it is advisable to use the Activity Monitor. It is an extremely useful tool that provides this information. A very useful feature of the Activity Monitor is the ability to provide information on the most recent expensive queries. This demonstration will show you how to use the Activity Monitor and how you can remove connections that place deadlocks on SQL Server resources.

# Execution Plans



Execution plans is a graphical tool that allows you to write queries, execute multiple queries simultaneously, view results, and analyze query execution plans. The Execution Plan options graphically display the data retrieval methods chosen by the SQL Server query optimizer. The graphical execution plan uses icons to represent the execution of specific statements.

The execution plan can be accessed within SQL Server Management Studio when a Query Editor window is opened by clicking **Query** on the menu bar and selecting **Display Estimated Execution Plan**.

There are a wide range of icons that appear depending on how the query optimizer decides how to retrieve the data. The graphical execution plan output in SQL Server Management Studio is read from right to left and from top to bottom. Each query in the batch that is analyzed is displayed, including the cost of each query as a percentage of the total cost of the batch.

**Execution Plans**

## **Graphical Execution Plan Icons**

The following icons displayed in the SQL Server Management Studio graphical execution plan represent operators used by SQL Server to execute statements.

Operators that are executed in parallel are displayed with the following parallel process icon above the operator. Description: Parallel process icon

|  |  |
| --- | --- |
| **Icon** | **Operator** |
| Description: Arithmetic expression operator icon | Arithmetic Expression |
| Description: Assert operator icon | Assert |
| Description: Bitmap operator icon | Bitmap |
| Description: Bookmark lookup operator icon | Bookmark Lookup |
| Description: Clustered index delete operator icon | Clustered Index Delete |
| Description: Clustered index insert operator icon | Clustered Index Insert |
| Description: Clustered index scan operator icon | Clustered Index Scan |
| Description: Clustered index seek operator icon | Clustered Index Seek |
| Description: Clustered index update operator icon | Clustered Index Update |
| Description: Collapse operator icon | Collapse |
| Description: Compute scalar operator icon | Compute Scalar |
| Description: Concatenation operator icon | Concatenation |
| Description: Constant scan operator icon | Constant Scan |
| Description: Delete (Database Engine) operator icon | Delete |
| Description: Delete scan operator icon | Deleted Scan |
| Description: Spool operator icon | Eager Spool |
| Description: Filter (Database Engine) operator icon | Filter |
| Description: Hash match operator icon | Hash Match |
| Description: Hash match root operator icon | Hash Match Root |
| Description: Hash match team operator icon | Hash Match Team |
| Description: Insert (Database Engine) operator icon | Insert |
| Description: Inserted scan operator icon | Inserted Scan |
| Description: Iterator catchall operator icon | Iterator Catchall |
| Description: Spool operator icon | Lazy Spool |
| Description: Log row scan operator icon | Log Row Scan |
| Description: Merge interval operator icon | Merge Interval |
| Description: Merge join operator icon | Merge Join |
| Description: Nested loops operator icon | Nested Loops |
| Description: Nonclustered index delete operator icon | Nonclustered Index Delete |
| Description: Nonclustered index insert operator icon | Nonclustered Index Insert |
| Description: Nonclustered index scan operator icon | Nonclustered Index Scan |
| Description: Nonclustered index seek operator icon | Nonclustered Index Seek |
| Description: Nonclustered index spool operator icon | Nonclustered Index Spool |
| Description: Nonclustered index update operator icon | Nonclustered Index Update |
| Description: Online index insert operator icon | Online Index Insert |
| Description: Parameter table scan operator icon | Parameter Table Scan |
| Description: Remote delete operator icon | Remote Delete |
| Description: Remote insert operator icon | Remote Insert |
| Description: Remote query operator icon | Remote Query |
| Description: Remote scan operator icon | Remote Scan |
| Description: Remote update operator icon | Remote Update |
| Description: RID lookup operator icon | RID Lookup |
| Description: Row count spool operator icon | Row Count Spool |
| Description: Segment operator icon | Segment |
| Description: Sequence operator icon | Sequence |
| Description: Sequence project operator icon | SequenceProject |
| Description: Sort operator icon | Sort |
| Description: Split operator icon | Split |
| Description: Spool operator icon | Spool |
| Description: Stream aggregate operator icon | Stream Aggregate |
| Description: Switch operator icon | Switch |
| Description: Table delete operator icon | Table Delete |
| Description: Table insert operator icon | Table Insert |
| Description: Table scan operator icon | Table Scan |
| Description: Table spool operator icon | Table Spool |
| Description: Table update operator icon | Table Update |
| Description: Table-valued function operator icon | Table-valued Function |
| Description: Top operator icon | Top |
| Description: Extended operator (UDX) icon | UDX |
| Description: Update (Database Engine) operator icon | Update |

The following icons displayed in the graphical execution plan represent the Cursor Logical and Physical Showplan Operators used by SQL Server to execute statements.

|  |  |
| --- | --- |
| **Icon** | **Cursor Physical Operator** |
| Description: Cursor catchall cursor operator icon | Cursor Catchall |
| Description: Dynamic cursor operator icon | Dynamic |
| Description: Fetch query cursor operator icon | Fetch Query |
| Description: Keyset cursor operator icon | Keyset |
| Description: Population query cursor operator icon | Population Query |
| Description: Refresh query cursor operator icon | Refresh Query |
| Description: Snapshot cursor operator icon | Snapshot |

The following icons displayed in the graphical execution plan represent the Parallelism Showplan Operator physical operators used by SQL Server to execute statements.

|  |  |
| --- | --- |
| **Icon** | **Parallelism Physical Operator** |
| Description: Distribute streams parallelism operator icon | Distribute Streams |
| Description: Repartition streams parallelism operator icon | Repartition Streams |
| Description: Gather streams parallelism operator icon | Gather Streams |

The following icons displayed in the graphical execution plan represent the Transact-SQL language elements used by SQL Server.

|  |  |
| --- | --- |
| **Icon** | **Language Element** |
| Description: Assign language element icon | Assign |
| Description: Convert (Database Engine) language element icon | Convert |
| Description: Declare language element icon | Declare |
| Description: If language element icon | If |
| Description: Intrinsic language element icon | Intrinsic |
| Description: Language element catchall icon | Language Element Catchall |
| Description: Result language element icon | Result |
| Description: While language element icon | While |

# **Graphical Execution Plan Node Tooltip**

Each node displays ToolTip information when the cursor is pointed at it as described in the following table. Not all nodes in a graphical execution plan contain all ToolTips items described here.

|  |  |
| --- | --- |
| **ToolTip Item** | **Description** |
| **Physical Operation** | The physical operator used, such as Hash Join or Nested Loops. Physical operators displayed in red indicate that the query optimizer has issued a warning, such as missing column statistics or missing join predicates. This can cause the query optimizer to choose a less-efficient query plan than otherwise expected.  When the graphical execution plan suggests creating or updating statistics, or creating an index, the missing column statistics, and indexes can be immediately created or updated using the shortcut menus in SQL Server Management Studio Object Explorer. |
| **Logical Operation** | The logical operator that matches the physical operator, such as the Inner Join operator. The logical operator is listed after the physical operator at the top of the ToolTip. |
| **Estimated Row Size** | The estimated size of the row produced by the operator (bytes). |
| **Estimated I/O Cost** | The estimated cost of all I/O activity for the operation. This value should be as low as possible. |
| **Estimated CPU Cost** | The estimated cost of all CPU activity for the operation. |
| **Estimated Operator Cost** | The cost to the query optimizer for executing this operation. The cost of this operation as a percentage of the total cost of the query is displayed in parentheses. Because the query engine selects the most efficient operation to perform the query or execute the statement, this value should be as low as possible. |
| **Estimated Subtree Cost** | The total cost to the query optimizer for executing this operation and all operations preceding it in the same subtree. |
| **Estimated Number of Rows** | The number of rows produced by the operator. This ToolTip item displays as **Number of Rows** in an Actual Execution Plan. |

# Dynamic Management Views

Dynamic management views can return information about a wide range of areas within a SQL Server instance or database. Microsoft SQL Server 2012 provides a number of dynamic management views regarding transaction. They can be accessed through select statements. The table below outlines some common dynamic management views relating to transactions.

**sys.dm\_tran\_active\_transactions**

**Columns Returned**

Returns information about transactions for the instance of SQL Server providing an overview of the activity that occurs on your SQL instance.

Some of the columns that are returned include:

**Transaction\_id -** ID of the transaction at the instance level.

**Name** is the name of the transaction.

**Transaction\_begin\_time** denotes the time the transaction started.

**Transaction\_type** displays the type of transaction, whether it is a read, write, system, or distributed transaction.

**sys.dm\_tran\_locks**

Returns information about currently active lock manager resources. Each row represents a currently active request to the lock manager for a lock that has been granted or is waiting to be granted.

Some of the columns that are returned include:

**Resource\_type** represents the resource type that is being locked.

**Resource\_database\_id** shows the database id in which the lock is present.

**Request\_mode** represents the type of lock being requested.

**Request\_status** represents if the request mode is granted.

**Request\_exec\_context\_id** provide the execution id of the request.

**SQL Server Profiler**

SQL Server Profiler is a graphical tool that enables you to monitor many components of SQL Server 2012.

Components can include SQL Server 2012 security, transactions, stored procedures, and locks. You can use SQL Server Profiler to monitor these components in real time, known as a trace.

Typically, you store the profiler information in a file or table for later review.

When creating a trace, you firstly create a new trace. You must first set up the general properties of the trace, specifying a trace name and whether to save the trace information to a trace file, to a SQL table, or to both.

Predefined templates are also available, which contain predefined events to monitor. You can also create your own templates for custom event monitoring. To view all the available events, select the blank template.

A trace stop time can also be defined in the general properties.

The Events tab allows you to define the events that you wish to monitor. These events are organized into categories that represent a component of SQL Server 2012.

You can select the events manually. For example, if you wish to monitor users connecting to SQL Server 2012, you can use the Audit Login event in the Security Audit category.

To monitor SQL transactions, select the SQL Transactions event from the Transaction category.

You can then specify the information that is logged for a given event by clicking on the check box for the column next to the event.

This enables you to determine the specific information about the event you wish to log. You can also set up filters to add conditions to the event that are traced and organize the column for improved readability. The trace is ready to be run.

The trace file can be stopped manually at any time. Once the trace is created, it can be replayed in SQL Server Profiler for review.

You can also use the results of a trace file with other SQL Server 2012 tools such as the Database Engine Tuning advisor to tune SQL Server 2012 or a specific database.

# Using Reliability and Performance Monitor

For a broad monitoring solution that encompasses Microsoft Windows, SQL Server, and the hardware that hosts these components consider using Reliability and Performance Monitor. This will enable you to rule out problems with the underlying operating system and hardware when perceived problems with transactions exist.

Performance Monitor can be used to provide real time information about hardware, services, and components on a physical server. Performance Monitor consists of objects that describe a component or area of the operating system such as the CPU or the memory object. When an instance of SQL Server is installed, the installation will add objects to performance monitor such as SQL Server:Databases and SQL Server:Transactions.

Each object contains counters that will measure a specific part of the object. For example, in the memory object the available megabyte counter can be used to monitor the amount of free space in the memory. In the SQL Server:Databases object the Percent Log Used counter can indicate how full a transaction log file is.

Some counters can also be broken down into instances. For example, the Available MBytes counter does not have instances as the Performance monitor looks at the Memory as on complete unit. However, with the Percent Log Used counter, you can select an instance of a database that you wish to monitor. For example, the Adventureworks database. You typically will have an instance named Total that would apply the counter to all instances.

Which tool should you use to kill a process that is causing a deadlock within SQL Server?

Select the one best answer.



Activity Monitor.

That is correct. Activity Monitor should be used to kill a process that is causing a deadlock within SQL Server.



SQL Server Profiler.

That is not correct. SQL Server Profiler is used to monitor specific areas of SQL Server. The correct response is: Activity Monitor. This tool should be used to kill a process that is causing a deadlock within SQL Server.



Reliability and Performance Monitor.

That is not correct. Reliability and Performance Monitor is used to provide a broad monitoring of the SQL Server, Windows and the server hardware. The correct response is: Activity Monitor. This tool should be used to kill a process that is causing a deadlock within SQL Server.



Dynamic Management Views.

Which Dynamic Management View returns information about transactions for the instance of SQL Server?

Select the one best answer.



sys.dm\_tran\_locks.

That is not correct. sys.dm\_tran\_locks returns information about locks for the instance of SQL Server. The correct answer is: sys.dm\_tran\_active\_transactions. This view returns information about locks for the instance of SQL Server.



sys.dm\_tran\_active\_transactions.

That is correct. sys.dm\_tran\_active\_transactions returns information about transactions for the instance of SQL Server. The correct answer is: sys.dm\_tran\_active\_transactions. This view returns information about locks for the instance of SQL Server.



SQL Server Profiler.

That is not correct. SQL Server Profiler is used to monitor specific areas of SQL Server. The correct answer is: sys.dm\_tran\_active\_transactions. This view returns information about locks for the instance of SQL Server.



All of the above.

# Best Practices



The following are best practices to follow when managing and monitoring transactions in Microsoft SQL Server 2012.

* Keep transactions as short a possible to reduce the amount of locking on SQL Server resources.
* Use explicit transactions for the best control of transactions within the SQL Server database.
* Choose the appropriate transaction isolation level that balances data integrity with database concurrency.
* Avoid using query hints in transactions.
* Use plan guides on transactions that you have no opportunity to change.
* Use Activity Monitor to kill processes that cause deadlocks.
* Use Windows Reliability and Performance Monitor to ensure the underlying Windows server and hardware is not the cause of slow performance.
* Use SQL Server Profiler to focus monitoring on specific transactions within SQL Server.
* Use Execution Plans to understand how the query optimizer is retrieving data from a SQL Server database.

**Introducing Transactions and Locks**

In this lesson, you have learned the following the key points.

* A transaction is a sequence of Transact-SQL commands that are processed as a single logical unit.
* If the transaction issued is successful, then it is committed to the database. If the transaction fails, then it is rolled back.
* A logical unit of work must exhibit four properties to qualify as a transaction; atomicity, consistency, isolation, and durability.
* How SQL Server modified data within a table.

How locking works in SQL Server and the impact it has on database concurrency.

**Managing Transactions**

In this lesson, you have learned the following the key points.

* The different transaction execution modes that are available within SQL Server.
* The role of autocommit mode and that it is overridden by explicit transaction mode or implicit transaction mode.
* That explicit transaction mode allows for greater control of transaction using the BEGIN TRANSACTION, COMMIT TRANSACTION, and ROLLBACK TRANSACTION.
* With implicit transaction mode, SQL Server automatically starts a new transaction after the current transaction is committed or rolled back.
* In implicit transaction mode, the transaction remains in effect until you issue a COMMIT or ROLLBACK statement.

An understanding of the guidelines for using transactions within Microsoft SQL Server 2012.

**Managing Locks**

In this lesson, you have learned the following the key points.

* The different types of problems can occur with the data that can affect the results that are returned to the result set including:
  + Lost updates
  + Dirty reads
  + Non repeatable read
  + Phantom reads
* The resources that can be locked within SQL Server.
* The types of locks that can be placed on SQL Server resources.
* The lock compatibility between the different types of locking modes.
* The importance of transaction isolation levels in maintaining concurrency and data integrity.

How query hints and plan guides can be used to override the query optimization process.

**Monitoring Transactions and Locks**

In this lesson, you have learned the following the key points.

* How activity monitor can be used to view the most recent expensive queries and kill processes that are causing deadlocks within SQL Server.
* How Execution plans can be used to graphically provide information on the way the query optimization process for a query has worked.
* How Dynamic management views can be used to provide information about transactions that are occurring on the system.
* An overview of the role of SQL Server profiler in monitoring transactions.

How Windows Reliability and Performance monitor can be used to provide about the SQL Server environment and the server and hardware on which SQL server is installed.

**Best Practices**

In this lesson, you learnt about the best practices that you can follow when managing and monitoring transactions in SQL Server 2012.

The following are best practices to be followed when managing and monitoring transactions in Microsoft SQL Server 2012.

* Keep transactions as short a possible to reduce the amount of locking on SQL Server resources.
* Use explicit transactions for the best control of transactions within the SQL Server database.
* Choose the appropriate transaction isolation level that balances data integrity with database concurrency.
* Avoid using query hints in transactions.
* Use plan guides on transactions that you have no opportunity to change.
* Use Activity Monitor to kill processes that cause deadlocks.
* Use Windows Reliability and Performance Monitor to ensure the underlying Windows server and hardware is not the cause of slow performance.
* Use SQL Server Profiler to focus monitoring on specific transactions within SQL Server.

Use Execution Plans to understand how the query optimizer is retrieving data from a SQL Server database