**Query Tuning**

**SQL Server 2008 R2**

[Other Versions](javascript:;)

http://i.msdn.microsoft.com/Areas/Epx/Content/Images/ImageSprite.png

* [SQL Server 2008](http://msdn.microsoft.com/en-us/library/ms176005(d=printer,v=sql.100).aspx)
* [SQL Server 2005](http://msdn.microsoft.com/en-us/library/ms176005(d=printer,v=sql.90).aspx)

Many administrators address performance problems solely by tuning system-level server performance: for example, memory size, type of file system, number and type of processors, and so on. However, many performance problems cannot be resolved this way. They are better addressed by also analyzing the application queries and updates that the application submits to the database, and how these queries and updates interact with the data contained in the database and the database schema.

[In This Section](javascript:void(0))

[Analyzing a Query](http://msdn.microsoft.com/en-us/library/ms191227(v=sql.105).aspx)

Contains information about displaying query execution plans by using Microsoft SQL Server Management Studio, Transact-SQL SET options, and by using SQL Server Profiler event classes in traces.

[Finding Missing Indexes](http://msdn.microsoft.com/en-us/library/ms345417(v=sql.105).aspx)

Contains information about the SQL Server 2005 "missing indexes" feature. When the query optimizer analyzes a query to determine the best execution plan, it automatically generates information about indexes that would improve query performance even if the indexes do not exist. The missing indexes feature enables you to retrieve this information about missing indexes.

[Query Tuning Recommendations](http://msdn.microsoft.com/en-us/library/ms188722(v=sql.105).aspx)

Contains information about what you can do to improve query performance.

[Advanced Query Tuning Concepts](http://msdn.microsoft.com/en-us/library/ms191426(v=sql.105).aspx)

Contains information about SQL Server join operations.

**Analyzing a Query**

**SQL Server 2008 R2**

[Other Versions](javascript:;)



* [SQL Server 2008](http://msdn.microsoft.com/en-us/library/ms191227(d=printer,v=sql.100).aspx)
* [SQL Server 2005](http://msdn.microsoft.com/en-us/library/ms191227(d=printer,v=sql.90).aspx)

The SQL Server Database Engine can display how it navigates tables and uses indexes to access or process the data for a query or other DML statement, such as an update. This is a display of an execution plan. To analyze a slow-running query, it is useful to examine the query execution plan to determine what is causing the problem. For more information about how SQL Server creates and uses execution plans, see [SQL Statement Processing](http://msdn.microsoft.com/en-us/library/ms190623(v=sql.105).aspx) and [Execution Plan Caching and Reuse](http://msdn.microsoft.com/en-us/library/ms181055(v=sql.105).aspx).

You can display execution plans by using the following methods:

* SQL Server Management Studio

Displays either an estimated graphical execution plan (statements do not execute) or an actual graphical execution plan (on executed statements), which you can save and view in Management Studio.

* Transact-SQL SET statement options

When you use the Transact-SQL SET statement options, you can produce estimated and actual execution plans in XML or text.

* SQL Server Profiler event classes

You can select SQL Server Profiler event classes to include in traces that produce estimated and actual execution plans in XML or text in the trace results.

When you use one of these methods to display execution plans, the best execution plan used by the Database Engine for individual data manipulation language (DML) and Transact-SQL statements is displayed. The plan reveals compile-time information about stored procedures and called stored procedures that are invoked to an arbitrary number of calling levels. For example, executing a SELECT statement may show that the Database Engine uses a table scan to obtain the data. Execution of the SELECT statement may also show that an index scan will be used if the Database Engine determines that an index scan is a faster method of retrieving the data from the table.

[In This Section](javascript:void(0))

[Checklist for Analyzing Slow-Running Queries](http://msdn.microsoft.com/en-us/library/ms177500(v=sql.105).aspx)

Lists and describes common causes for slow-running queries and what you can do to improve query performance.

[Displaying Graphical Execution Plans (SQL Server Management Studio)](http://msdn.microsoft.com/en-us/library/ms178071(v=sql.105).aspx)

Contains information about using SQL Server Management Studio to display execution plans. Also provides a reference describing all icons that are used to graphically display execution plans in Management Studio.

[Displaying Execution Plans by Using the Showplan SET Options (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms180765(v=sql.105).aspx)

Contains information about using the Transact-SQL SET statement options to display execution plans in XML format or text.

[Displaying Execution Plans by Using SQL Server Profiler Event Classes](http://msdn.microsoft.com/en-us/library/ms190233(v=sql.105).aspx)

Contains information about using SQL Server Profiler event classes in traces to display execution plans in XML format or text.

[Showplan Security](http://msdn.microsoft.com/en-us/library/ms189602(v=sql.105).aspx)

Contains information about the SHOWPLAN permission and about what permissions are required for using the various methods to display execution plans.

[XML Showplans](http://msdn.microsoft.com/en-us/library/ms189298(v=sql.105).aspx)

Contains information about the Showplan XML schema.

[Transact-SQL Statements That Produce Showplans](http://msdn.microsoft.com/en-us/library/ms187886(v=sql.105).aspx)

Contains information about which Transact-SQL statements produce Showplan execution plan information.

[Interpreting Execution Plans Containing Bitmap Filters](http://msdn.microsoft.com/en-us/library/bb510541(v=sql.105).aspx)

Describes how to understand the query execution plans that contain dynamic filtering.

[Logical and Physical Operators Reference](http://msdn.microsoft.com/en-us/library/ms191158(v=sql.105).aspx)

Contains reference information about all possible logical and physical operators that are displayed in execution plans. Use this reference to read execution plan output.

# SQL Statement Processing

**SQL Server 2008 R2**

[Other Versions](javascript:;)



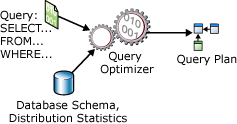
* [SQL Server 2008](http://msdn.microsoft.com/en-us/library/ms190623(d=printer,v=sql.100).aspx)
* [SQL Server 2005](http://msdn.microsoft.com/en-us/library/ms190623(d=printer,v=sql.90).aspx)

Processing a single SQL statement is the most basic way that SQL Server executes SQL statements. The steps used to process a single SELECT statement that references only local base tables (no views or remote tables) illustrates the basic process.

[Optimizing SELECT Statements](javascript:void(0))

A SELECT statement is nonprocedural; it does not state the exact steps that the database server should use to retrieve the requested data. This means that the database server must analyze the statement to determine the most efficient way to extract the requested data. This is referred to as optimizing the SELECT statement. The component that does this is called the query optimizer. The input to the optimizer consists of the query, the database schema (table and index definitions), and the database statistics. The output of the optimizer is a query execution plan, sometimes referred to as a query plan or just a plan. The contents of a query plan are described in more detail later in this topic.

The inputs and outputs of the query optimizer during optimization of a single SELECT statement are illustrated in the following diagram:



A SELECT statement defines only the following:

* The format of the result set. This is specified mostly in the select list. However, other clauses such as ORDER BY and GROUP BY also affect the final form of the result set.
* The tables that contain the source data. This is specified in the FROM clause.
* How the tables are logically related for the purposes of the SELECT statement. This is defined in the join specifications, which may appear in the WHERE clause or in an ON clause following FROM.
* The conditions that the rows in the source tables must satisfy to qualify for the SELECT statement. These are specified in the WHERE and HAVING clauses.

A query execution plan is a definition of the following:

* The sequence in which the source tables are accessed.

Typically, there are many sequences in which the database server can access the base tables to build the result set. For example, if the SELECT statement references three tables, the database server could first access **TableA**, use the data from **TableA** to extract matching rows from **TableB**, and then use the data from **TableB** to extract data from **TableC**. The other sequences in which the database server could access the tables are:

**TableC**, **TableB**, **TableA**, or

**TableB**, **TableA**, **TableC**, or

**TableB**, **TableC**, **TableA**, or

**TableC**, **TableA**, **TableB**

* The methods used to extract data from each table.

Generally, there are different methods for accessing the data in each table. If only a few rows with specific key values are required, the database server can use an index. If all the rows in the table are required, the database server can ignore the indexes and perform a table scan. If all the rows in a table are required but there is an index whose key columns are in an ORDER BY, performing an index scan instead of a table scan may save a separate sort of the result set. If a table is very small, table scans may be the most efficient method for almost all access to the table.

The process of selecting one execution plan from potentially many possible plans is referred to as optimization. The query optimizer is one of the most important components of a SQL database system. While some overhead is used by the query optimizer to analyze the query and select a plan, this overhead is typically saved several-fold when the query optimizer picks an efficient execution plan. For example, two construction companies can be given identical blueprints for a house. If one company spends a few days at the beginning to plan how they will build the house, and the other company begins building without planning, the company that takes the time to plan their project will probably finish first.

The SQL Server query optimizer is a cost-based optimizer. Each possible execution plan has an associated cost in terms of the amount of computing resources used. The query optimizer must analyze the possible plans and choose the one with the lowest estimated cost. Some complex SELECT statements have thousands of possible execution plans. In these cases, the query optimizer does not analyze all possible combinations. Instead, it uses complex algorithms to find an execution plan that has a cost reasonably close to the minimum possible cost.

The SQL Server query optimizer does not choose only the execution plan with the lowest resource cost; it chooses the plan that returns results to the user with a reasonable cost in resources and that returns the results the fastest. For example, processing a query in parallel typically uses more resources than processing it serially, but completes the query faster. The SQL Server optimizer will use a parallel execution plan to return results if the load on the server will not be adversely affected.

The query optimizer relies on distribution statistics when it estimates the resource costs of different methods for extracting information from a table or index. Distribution statistics are kept for columns and indexes. They indicate the selectivity of the values in a particular index or column. For example, in a table representing cars, many cars have the same manufacturer, but each car has a unique vehicle identification number (VIN). An index on the VIN is more selective than an index on the manufacturer. If the index statistics are not current, the query optimizer may not make the best choice for the current state of the table. For more information about keeping index statistics current, see [Using Statistics to Improve Query Performance](http://msdn.microsoft.com/en-us/library/ms190397(v=sql.105).aspx).

The query optimizer is important because it enables the database server to adjust dynamically to changing conditions in the database without requiring input from a programmer or database administrator. This enables programmers to focus on describing the final result of the query. They can trust that the query optimizer will build an efficient execution plan for the state of the database every time the statement is run.

[Processing a SELECT Statement](javascript:void(0))

The basic steps that SQL Server uses to process a single SELECT statement include the following:

1. The parser scans the SELECT statement and breaks it into logical units such as keywords, expressions, operators, and identifiers.
2. A query tree, sometimes referred to as a sequence tree, is built describing the logical steps needed to transform the source data into the format required by the result set.
3. The query optimizer analyzes different ways the source tables can be accessed. It then selects the series of steps that returns the results fastest while using fewer resources. The query tree is updated to record this exact series of steps. The final, optimized version of the query tree is called the execution plan.
4. The relational engine starts executing the execution plan. As the steps that require data from the base tables are processed, the relational engine requests that the storage engine pass up data from the rowsets requested from the relational engine.
5. The relational engine processes the data returned from the storage engine into the format defined for the result set and returns the result set to the client.

[Processing Other Statements](javascript:void(0))

The basic steps described for processing a SELECT statement apply to other SQL statements such as INSERT, UPDATE, and DELETE. UPDATE and DELETE statements both have to target the set of rows to be modified or deleted. The process of identifying these rows is the same process used to identify the source rows that contribute to the result set of a SELECT statement. The UPDATE and INSERT statements may both contain embedded SELECT statements that provide the data values to be updated or inserted.

Even Data Definition Language (DDL) statements, such as CREATE PROCEDURE or ALTER TABLE, are ultimately resolved to a series of relational operations on the system catalog tables and sometimes (such as ALTER TABLE ADD COLUMN) against the data tables.

# Checklist for Analyzing Slow-Running Queries

**SQL Server 2008 R2**

[Other Versions](javascript:;)



* [SQL Server 2008](http://msdn.microsoft.com/en-us/library/ms177500(d=printer,v=sql.100).aspx)
* [SQL Server 2005](http://msdn.microsoft.com/en-us/library/ms177500(d=printer,v=sql.90).aspx)

Queries or updates that take longer than expected to execute can be caused by a variety of reasons. Slow-running queries can be caused by performance problems related to your network or the computer where SQL Server is running. Slow-running queries can also be caused by problems with your physical database design.

There are a number of common reasons for slow-running queries and updates:

* Slow network communication.
* Inadequate memory in the server computer, or not enough memory available for SQL Server.
* Lack of useful statistics
* Lack of useful indexes.
* Lack of useful indexed views.
* Lack of useful data striping.
* Lack of useful partitioning.

When a query or update takes longer than expected, ask yourself the following questions, which address the reasons for slow-running queries that are listed in the previous section:

|  |
| --- |
| **TipTip** |
| To save time, consult this checklist before you contact your technical support provider. |

1. Is the performance problem related to a component other than queries? For example, is the problem slow network performance? Are there any other components that might be causing or contributing to performance degradation?

The Windows System Monitor can be used to monitor the performance of SQL Server and non-SQL Server related components. For more information, see [Monitoring Resource Usage (System Monitor)](http://msdn.microsoft.com/en-us/library/ms191246(v=sql.105).aspx).

1. If the performance issue is related to queries, which query or set of queries is involved?

Use SQL Server Profiler to help identify the slow query or queries. For more information, see [Using SQL Server Profiler](http://msdn.microsoft.com/en-us/library/ms187929(v=sql.105).aspx). Use the [sys.dm\_exec\_query\_stats](http://msdn.microsoft.com/en-us/library/ms189741(v=sql.105).aspx) and [sys.dm\_exec\_requests](http://msdn.microsoft.com/en-us/library/ms177648(v=sql.105).aspx) dynamic management views to find similar queries that collectively consume a large number of resources. For more information, see [Finding and Tuning Similar Queries by Using Query and Query Plan Hashes](http://msdn.microsoft.com/en-us/library/cc645887(v=sql.105).aspx).

1. How do I analyze the performance of a slow-running query?

After you have identified the slow-running query or queries, you can further analyze query performance by producing a Showplan, which can be a text, XML, or graphical representation of the query execution plan that the query optimizer generates. You can produce a Showplan using Transact-SQL SET options, SQL Server Management Studio, or SQL Server Profiler.

For information about using Transact-SQL SET options to display text and XML execution plans, see [Displaying Execution Plans by Using the Showplan SET Options (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms180765(v=sql.105).aspx).

For information about using SQL Server Management Studio to display graphical execution plans, see [Displaying Graphical Execution Plans (SQL Server Management Studio)](http://msdn.microsoft.com/en-us/library/ms178071(v=sql.105).aspx).

For information about using SQL Server Profiler to display text and XML execution plans, see [Displaying Execution Plans by Using SQL Server Profiler Event Classes](http://msdn.microsoft.com/en-us/library/ms190233(v=sql.105).aspx).

The information gathered by these tools allows you to determine how a query is executed by the SQL Server query optimizer and which indexes are being used. Using this information, you can determine if performance improvements can be made by rewriting the query, changing the indexes on the tables, or perhaps modifying the database design. For more information, see [Analyzing a Query](http://msdn.microsoft.com/en-us/library/ms191227(v=sql.105).aspx).

1. Was the query optimized with useful statistics?

The query optimizer uses statistics to create query plans that improve query performance. For most queries, the query optimizer already generates the necessary statistics for a high-quality query plan; in a few cases, you need to create additional statistics or modify the query design for best results.

For more information, see [Using Statistics to Improve Query Performance](http://msdn.microsoft.com/en-us/library/ms190397(v=sql.105).aspx). This topic gives guidelines to improve the effectiveness of statistics for query performance. The guidelines include the following:

* + Using the Database-Wide Statistics Options. For example, you should verify that the automatic create statistics, AUTO\_CREATE\_STATISTICS, and automatic update statistics, AUTO\_UPDATE\_STATISTICS, database-wide options are on. If they are off, query plans can be suboptimal and query performance can degrade.
  + Determining When to Create Statistics. In a few cases, you can improve query plans by creating additional statistics with the [CREATE STATISTICS (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms188038(v=sql.105).aspx) statement. These additional statistics can capture statistical correlations that the query optimizer does not account for when it creates statistics for indexes or single columns.
  + Determining When to Update Statistics. In some cases you can improve the query plan and therefore improve query performance by updating statistics more frequently than when AUTO\_UPDATE\_STATISTICS is on. You can update statistics with the [UPDATE STATISTICS](http://msdn.microsoft.com/en-us/library/ms187348(v=sql.105).aspx) statement or the stored procedure [sp\_updatestats](http://msdn.microsoft.com/en-us/library/ms173804(v=sql.105).aspx).
  + Designing Queries That Use Statistics Effectively. Certain query implementations, such as local variables and complex expressions in the query predicate, can lead to suboptimal query plans. Following query design guidelines for using statistics effectively can help to avoid this.

1. Are suitable indexes available? Would adding one or more indexes improve query performance? For more information, see [General Index Design Guidelines](http://msdn.microsoft.com/en-us/library/ms191195(v=sql.105).aspx), [Finding Missing Indexes](http://msdn.microsoft.com/en-us/library/ms345417(v=sql.105).aspx), and [Database Engine Tuning Advisor Overview](http://msdn.microsoft.com/en-us/library/ms173494(v=sql.105).aspx). Database Engine Tuning Advisor can also recommend the creation of necessary statistics.
2. Are there any data or index hot spots? Consider using disk striping. Disk striping can be implemented by using RAID (redundant array of independent disks) level 0, where data is distributed across multiple disk drives. For more information, see [Using Files and Filegroups](http://msdn.microsoft.com/en-us/library/ms187087(v=sql.105).aspx) and [RAID](http://msdn.microsoft.com/en-us/library/ms184252(v=sql.105).aspx).
3. Is the query optimizer provided with the best opportunity to optimize a complex query? For more information, see [Query Tuning Recommendations](http://msdn.microsoft.com/en-us/library/ms188722(v=sql.105).aspx).
4. If you have a large volume of data, do you need to partition it? Data manageability is the main benefit of partitioning, but if your tables and indexes on them are partitioned similarly, partitioning can also improve query performance. For more information, see [Understanding Partitioning](http://msdn.microsoft.com/en-us/library/ms188232(v=sql.105).aspx) and [Tuning the Physical Database Design](http://msdn.microsoft.com/en-us/library/ms191531(v=sql.105).aspx).

# Displaying Graphical Execution Plans (SQL Server Management Studio)

**SQL Server 2008 R2**

[Other Versions](javascript:;)



* [SQL Server 2008](http://msdn.microsoft.com/en-us/library/ms178071(d=printer,v=sql.100).aspx)
* [SQL Server 2005](http://msdn.microsoft.com/en-us/library/ms178071(d=printer,v=sql.90).aspx)

SQL Server Management Studio is an interactive, graphical tool that enables a database administrator or developer to write queries, execute multiple queries simultaneously, view results, analyze the query plan, and receive assistance to improve the query performance. 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 and queries in SQL Server rather than the tabular representation produced by the Transact-SQL SET statement options SET SHOWPLAN\_ALL or SET SHOWPLAN\_TEXT, or the XML for representation produced by SET SHOWPLAN\_XML. The graphical display is very useful for understanding the performance characteristics of a query. SQL Server Management Studio shows which statistics are missing, thereby forcing the query optimizer to make estimates about predicate selectivity, and then permits those missing statistics to be easily created.

|  |
| --- |
| **NoteNote** |
| Execution plans are not displayed for encrypted stored procedures or for triggers. |

[Using the Execution Plan Options](javascript:void(0))

Open or type a Transact-SQL script that contains the queries you want to analyze into the Management Studio query editor. After the script has been loaded into the Management Studio query editor, you can choose to either display an estimated execution plan or the actual execution plan by clicking the Display Estimated Execution Plan or the Include Actual Execution Plan button on the query editor toolbar. If you click Display Estimated Execution Plan, the script is parsed and an estimated execution plan is generated. If you click Include Actual Execution Plan, you must execute the script before the execution plan is generated. After the script is parsed or executed, click the Execution plan tab to see a graphical representation of execution plan output.

To use the graphical execution plan feature in Management Studio, and to use the Showplan Transact-SQL SET statement options, users must have sufficient permissions to execute the Transact-SQL statements and queries. Users must also be granted the SHOWPLAN permission for all databases containing referenced objects. For more information, see [Showplan Security](http://msdn.microsoft.com/en-us/library/ms189602(v=sql.105).aspx).

[Reading the Graphical Execution Plan Output](javascript:void(0))

To view the execution plan, click the Execution plan tab in the results pane. 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. For more information about the icons used to display execution plans in Management Studio, see [Graphical Execution Plan Icons (SQL Server Management Studio)](http://msdn.microsoft.com/en-us/library/ms175913(v=sql.105).aspx).

The following descriptions provide guidelines for interpreting the graphical execution output in Management Studio:

* Each node in the tree structure is represented as an icon that specifies the logical and physical operator used to execute that part of the query or statement.
* Each node is related to a parent node. Child nodes of the same parents are drawn in the same column. However, all nodes in the same column do not necessarily have the same parent. Rules with arrowheads connect each node to its parent.
* Operators are shown as symbols related to a specific parent.
* Arrow width is proportional to the number of rows. Actual number of rows is used when it is available. If not, then the estimated number of rows is used.
* When the query contains multiple statements, multiple query execution plans are drawn.
* The parts of the tree structures are determined by the type of statement executed.
* For parallel queries, which involve multiple CPUs, the Properties for each node in the graphical execution plan displays information about the operating system threads used. To view the properties for a node, right-click the node, and then click Properties. For more information about parallel queries, see [Parallel Query Processing](http://msdn.microsoft.com/en-us/library/ms178065(v=sql.105).aspx).

|  |  |
| --- | --- |
| **Type of statement** | **Tree structure element** |
| Transact-SQL and stored procedures | If the statement is a stored procedure or Transact-SQL statement, it becomes the root of the graphical execution plan tree structure. The stored procedure can have multiple children that represent statements called by the stored procedure. Each child is a node or branch of the tree. |
| Data manipulation language (DML) | If the statement analyzed by the SQL Server query optimizer is a DML statement, such as SELECT, INSERT, DELETE, or UPDATE, the DML statement is the root of the tree. DML statements can have up to two children. The first child is the execution plan for the DML statement. The second child represents a trigger, if used in or by the statement. |
| Conditional | The graphical execution plan divides conditional statements such as IF...ELSE statements (if condition exists, then do the following, else do this statement instead) into three children. The IF...ELSE statement is the root of the tree. The IF condition becomes a subtree node. The THEN and ELSE conditions are represented as statement blocks. WHILE and DO-UNTIL statements are represented using a similar plan. IF and WHILE have their own icons. |
| Relational operators | Operations performed by the query engine, such as table scans, joins, and aggregations, are represented as nodes on the tree. |
| DECLARE CURSOR | The DECLARE CURSOR statement is the root of the graphical execution plan tree, with its related statement as a child or node. |

Individual colors have been associated with each of the three icon types: iterator (logical and physical operators) icons are blue, cursors icons are yellow, and language elements are green.

[Graphical Execution Plan Node ToolTips](javascript:void(0))

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. For more information about column statistics, see [Using Statistics to Improve Query Performance](http://msdn.microsoft.com/en-us/library/ms190397(v=sql.105).aspx).  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. For more information, see [Indexes How-to Topics](http://msdn.microsoft.com/en-us/library/ms189630(v=sql.105).aspx). |
| 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 1 | The number of rows produced by the operator. |

1 This ToolTip item displays as Number of Rows in an Actual Execution Plan.

|  |
| --- |
| **NoteNote** |
| The information that was available in the Argument fields of the graphical execution plans in SQL Server 2000 now appears separately in the ToolTips of the graphical execution plans. |

[Graphical Execution Plan Node Properties](javascript:void(0))

You can view detailed information about nodes in the graphical execution plan by right-clicking the node, and then clicking Properties.

# Displaying Execution Plans by Using the Showplan SET Options (Transact-SQL)

**SQL Server 2008 R2**

[Other Versions](javascript:;)



* [SQL Server 2008](http://msdn.microsoft.com/en-us/library/ms180765(d=printer,v=sql.100).aspx)
* [SQL Server 2005](http://msdn.microsoft.com/en-us/library/ms180765(d=printer,v=sql.90).aspx)

The Transact-SQL SET statement options for displaying execution plan information produce output in XML and text. These options are listed and described in this topic.

[Showplan SET Statement Options](javascript:void(0))

Transact-SQL provides the following options to its SET statement that enable you to display a query execution plan:

|  |
| --- |
| **NoteNote** |
| The SHOWPLAN\_XML, SHOWPLAN\_ALL, and SHOWPLAN\_TEXT SET options produce one rowset for each batch. The STATISTICS XML and STATISTICS PROFILE SET options produce one rowset for each query in a batch. |

* SET SHOWPLAN\_XML ON

This statement causes SQL Server not to execute Transact-SQL statements. Instead, Microsoft SQL Server returns execution plan information about how the statements are going to be executed in a well-formed XML document. For more information, see [SET SHOWPLAN\_XML (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms187757(v=sql.105).aspx).

* SET SHOWPLAN\_TEXT ON

After this SET statement is executed, SQL Server returns the execution plan information for each query in text. The Transact-SQL statements or batches are not executed. For more information, see [SET SHOWPLAN\_TEXT (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms176058(v=sql.105).aspx).

* SET SHOWPLAN\_ALL ON

This statement is similar to SET SHOWPLAN\_TEXT, except that the output is in a format more verbose than that of SHOWPLAN\_TEXT. For more information, see [SET SHOWPLAN\_ALL (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms187735(v=sql.105).aspx).

* SET STATISTICS XML ON

Returns execution information for each statement after the statement executes in addition to the regular result set the statement returns. The output is a set of well-formed XML documents. SET STATISTICS XML ON produces an XML output document for each statement that executes. The difference between SET SHOWPLAN\_XML ON and SET STATISTICS XML ON is that the second SET option executes the Transact-SQL statement or batch. SET STATISTICS XML ON output also includes information about the actual number of rows processed by various operators and the actual number of executes of the operators. For more information, see [SET STATISTICS XML (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms176107(v=sql.105).aspx).

* SET STATISTICS PROFILE ON

Returns the execution information for each statement after the statement executes in addition to the regular result set the statement returns. Both SET statement options provide output in text. The difference between SET SHOWPLAN\_ALL ON and SET STATISTICS PROFILE ON is that the second SET option executes the Transact-SQL statement or batch. SET STATISTICS PROFILE ON output also includes information about the actual number of rows processed by various operators and the actual number of executes of the operators. For more information, see [SET STATISTICS PROFILE (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms188752(v=sql.105).aspx).

* SET STATISTICS IO ON

Displays information about the amount of disk activity that is generated by Transact-SQL statements after the statements execute. This SET option produces text output. For more information, see [SET STATISTICS IO (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms184361(v=sql.105).aspx).

* SET STATISTICS TIME ON

Displays the number of milliseconds required to parse, compile, and execute each Transact-SQL statement after statements execute. This SET option produces text output. For more information, see [SET STATISTICS TIME (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms190287(v=sql.105).aspx).

[Considerations for Using the Showplan SET Statement Options](javascript:void(0))

When you display an execution plan using the SHOWPLAN SET options, the statements you submit to the server are not executed. Instead, SQL Server analyzes the query and displays, in a series of operators, how the statements would have been executed.

|  |
| --- |
| **NoteNote** |
| Because statements are not executed when the execution plan is displayed, Transact-SQL operations are not actually carried out. So, for example, if a CREATE TABLE statement is part of an execution plan, any later operations involving the "created" table returns errors, because the table does not exist. However, there are two exceptions to this rule: temporary tables are created when using the SHOWPLAN SET options, and USE db\_name statements are executed and attempt to change the database context to the db\_name specified when using the SHOWPLAN SET options. |

When you display an execution plan using the STATISTICS SET options, the Transact-SQL statements you submit to the server are executed.

|  |
| --- |
| **NoteNote** |
| The Showplan SET options display no information about encrypted stored procedures or triggers. |

[SET Options Scheduled for Deprecation in Future Showplan Versions](javascript:void(0))

In a future version of SQL Server, the following Showplan SET options will be deprecated. We recommend that users move to the newer modes as soon as possible. The following table lists the Showplan SET options that are scheduled for deprecation with the new SET options that users should start using.

|  |  |
| --- | --- |
| **Deprecated SET Option** | **Use New SET Option** |
| SET SHOWPLAN\_TEXT | SET SHOWPLAN\_XML |
| SET SHOWPLAN\_ALL | SET SHOWPLAN\_XML |
| SET STATISTICS PROFILE | SET STATISTICS XML |

# Displaying Execution Plans by Using SQL Server Profiler Event Classes

**SQL Server 2008 R2**

[Other Versions](javascript:;)



* [SQL Server 2008](http://msdn.microsoft.com/en-us/library/ms190233(d=printer,v=sql.100).aspx)
* [SQL Server 2005](http://msdn.microsoft.com/en-us/library/ms190233(d=printer,v=sql.90).aspx)

The following SQL Server Profiler event classes capture Showplan information. To display execution plan information by using these event classes, you must also include the appropriate event classes from the Stored Procedures and Transact-SQL Event Categories in your trace definition. For more information, see [SQL Server Event Class Reference](http://msdn.microsoft.com/en-us/library/ms175481(v=sql.105).aspx).

Users must be granted the ALTER TRACE permission to use SQL Server Profiler to display execution plans.

|  |
| --- |
| **NoteNote** |
| The SQL Server Profiler events that capture Showplan information produce one rowset for each query. No Showplan information is generated for encrypted stored procedures or for triggers. |

|  |  |
| --- | --- |
| **Event class** | **Description** |
| **Showplan XML** | Occurs when a query executes on SQL Server. It captures the estimated execution plan in XML format with full compile-time details in the **TextData** data column of the trace. For more information, see [Showplan XML Event Class](http://msdn.microsoft.com/en-us/library/ms189318(v=sql.105).aspx). |
| **Showplan XML For Query Compile** | Occurs when a query is compiled or recompiled on SQL Server. This is the compile time counterpart of the **Showplan XML** event. **Showplan XML** occurs when a query is executed. **Showplan XML For Query Compile** occurs when a query is compiled. For more information, see [Showplan XML for Query Compile Event Class](http://msdn.microsoft.com/en-us/library/ms186344(v=sql.105).aspx). |
| **Showplan Text** | Occurs when a query executes on SQL Server. It displays the estimated query execution plan tree of the Transact-SQL statement being executed. For more information, see [Showplan Text Event Class](http://msdn.microsoft.com/en-us/library/ms191314(v=sql.105).aspx). |
| **Showplan Text (Unencoded)** | Occurs when SQL Server executes a Transact-SQL statement. It displays the same information as the **Showplan Text** event class, except the event information is formatted as a string rather than as binary data. For more information, see [Showplan Text (Unencoded) Event Class](http://msdn.microsoft.com/en-us/library/ms175465(v=sql.105).aspx). |
| **Showplan All** | Occurs when a query executes on SQL Server. It displays the estimated execution plan with compile-time details. For more information, see [Showplan All Event Class](http://msdn.microsoft.com/en-us/library/ms191283(v=sql.105).aspx). |
| **Showplan All For Query Compile** | Occurs when a query is compiled or recompiled on SQL Server. This is the compile time counterpart of the **Showplan All** event. **Showplan All** occurs when a query is executed. **Showplan All For Query Compile** occurs when a query is compiled. For more information, see [Showplan All for Query Compile Event Class](http://msdn.microsoft.com/en-us/library/ms190408(v=sql.105).aspx). |
| **Showplan XML Statistics Profile** | Occurs during run time. It captures the actual execution plan in XML format with full run-time details in the **TextData** data column of the trace. For more information, see [Showplan XML Statistics Profile Event Class](http://msdn.microsoft.com/en-us/library/ms188661(v=sql.105).aspx). |
| **Showplan Statistics Profile** | Occurs during run time. It displays the actual execution plan with full run-time details in textual format. For more information, see [Showplan Statistics Profile Event Class](http://msdn.microsoft.com/en-us/library/ms191427(v=sql.105).aspx). |
| **Performance statistics** | This event is similar to **Showplan XML For Query Compile**. It occurs when a compiled query plan is cached for the first time, compiled or recompiled any number of times, and when the plan is flushed from the cache. In some cases, the **TextData** data column for this event contains the plan in XML format that is being compiled or recompiled. For more information, see [Performance Statistics Event Class](http://msdn.microsoft.com/en-us/library/ms190971(v=sql.105).aspx). |

[Event Classes Scheduled for Deprecation in Future Showplan Versions](javascript:void(0))

In a future version of SQL Server, the following SQL Server Profiler event classes will be deprecated. We recommend that users move to using the newer event classes as soon as possible. The event classes that are scheduled for deprecation are listed in the following table with the new event class that users should use.

|  |  |
| --- | --- |
| **Deprecated SQL Server Profiler event** | **Use new SQL Server Profiler event** |
| Showplan All | Showplan XML |
| Showplan All For Query Compile | Showplan XML For Query Compile |
| Showplan Statistics Profile | Showplan XML Statistics Profile |
| Showplan Text | Showplan XML |
| Showplan Text (Unencoded) | Showplan XML |

# Showplan Security

**SQL Server 2008 R2**

[Other Versions](javascript:;)



* [SQL Server 2008](http://msdn.microsoft.com/en-us/library/ms189602(d=printer,v=sql.100).aspx)
* [SQL Server 2005](http://msdn.microsoft.com/en-us/library/ms189602(d=printer,v=sql.90).aspx)

Showplan execution plan information can be produced by various ways. You can use Transact-SQL SET statement options, SQL Server Profiler event classes, or you can query the dynamic management function **sys.dm\_exec\_query\_plan**. Each method requires a different set of permissions, which are described in the following sections. For more information about how the SHOWPLAN permission is checked for Transact-SQL batches, see [SHOWPLAN Permission and Transact-SQL Batches](http://msdn.microsoft.com/en-us/library/ms178086(v=sql.105).aspx).

|  |
| --- |
| **NoteNote** |
| When a SQL Server database compatibility level is set to **80** by using the **sp\_dbcmptlevel** stored procedure the current SHOWPLAN permission still applies. Setting the compatibility level to **80** does not produce the Showplan permissions behavior of Microsoft SQL Server 2000. |

[About the SHOWPLAN Permission](javascript:void(0))

To produce execution plan output by using most Showplan Transact-SQL SET options, users must have:

* The SHOWPLAN permission on the databases that contain objects referred to in the Transact-SQL statement, such as views, stored procedures, or user-defined functions.
* The appropriate permission to execute the Transact-SQL statement itself.

|  |
| --- |
| **Security noteSecurity Note** |
| Users who have SHOWPLAN, ALTER TRACE, or VIEW SERVER STATE permission can view queries that are captured in Showplan output. These queries may contain sensitive information such as passwords. Therefore, we recommend that you only grant these permissions to users who are authorized to view sensitive information, such as members of the **db\_owner** fixed database role, or members of the **sysadmin** fixed server role. We also recommend that you only save Showplan files or trace files that contain Showplan-related events to a location that uses the NTFS file system, and that you restrict access to users who are authorized to view sensitive information.  For example, consider the following query:  SELECT COUNT(\*)  FROM table\_1  WHERE column\_1 < 10  If a malicious user produces Showplan output for a set of queries like this example, and replaces the value "10" in the predicate with different constants each time, the user could infer an approximate data distribution of the column values for **column\_1** in **table\_1** by reading the estimated row counts. |

The SHOWPLAN permission is a database-level permission which:

* Can be granted, denied, or revoked only by the following users:
  + Members of the **sysadmin** fixed server role. By default, all members of this fixed server role have the SHOWPLAN permission on all of the databases on the server.
  + Members of the **dbcreator** fixed server role for databases they create and thus own. By default, all members of this fixed server role have the SHOWPLAN permission on databases they create and thus own.
  + Members of the **db\_owners** fixed database role for databases they own. By default, all members of this fixed database role have the SHOWPLAN permission on databases they own.
* Supports ownership chaining. When the ownership chain is broken, the permission is checked again at the node where the break occurred. However, because the SHOWPLAN permission is a database-level permission, this check only occurs when queries reference objects in two or more databases. For more information about ownership chaining, see [Ownership Chains](http://msdn.microsoft.com/en-us/library/ms188676(v=sql.105).aspx).

For information about the syntax used to grant, deny, or revoke the SHOWPLAN permission, see [Syntax for Granting, Denying, and Revoking the SHOWPLAN Permission](http://msdn.microsoft.com/en-us/library/ms175117(v=sql.105).aspx).

### Example

If User1 has CREATE TABLE, INSERT, and SELECT permissions, and he creates table **T** (he is the table owner) in database **D**, inserts rows into the table, and then writes a SELECT query on the table, the query executes successfully. However, User1 is not able to generate a Showplan until he is granted the SHOWPLAN permission on database **D**.

#### Caveat

In the previous example, suppose that database **D** contains view **V** for which User1 has SELECT permission. After User1 has been granted the SHOWPLAN permission for **D**, although he does not own **V**, he can still generate a Showplan on a query posed to **V**. This Showplan enables him to see the view definition for **V**, including the tables and views on which **V** is based. However, if **V** contains an object, such as a table, that is owned by User1 and which exists in a different database, **D2**, and User1 is not the owner of **D2**, the SHOWPLAN permission on **D2** is checked and required.

[Permissions Required to Use Showplan SET Options](javascript:void(0))

The permissions required to use the various Showplan SET statement options are listed in the following table:

|  |  |
| --- | --- |
| **Showplan SET options** | **Permissions required** |
| SET SHOWPLAN\_XML ON  SET SHOWPLAN\_ALL ON  SET SHOWPLAN\_TEXT ON | For SELECT, INSERT, UPDATE, DELETE, EXEC stored\_prodedure, and EXEC user\_defined\_function statements, the following permissions are required to produce a Showplan:   * Appropriate permissions to execute the Transact-SQL statements. * SHOWPLAN permission on all databases containing objects referenced by the Transact-SQL statements, such as tables, views, and so on.   For all other statements, such as DDL, USE database\_name, SET, DECLARE, dynamic Transact-SQL, and so on, only the appropriate permissions to execute the Transact-SQL statement are needed. For more information, see [SHOWPLAN Permission and Transact-SQL Batches](http://msdn.microsoft.com/en-us/library/ms178086(v=sql.105).aspx). |
| SET STATISTICS XML ON  SET STATISTICS PROFILE ON | * Appropriate permissions to execute the Transact-SQL statements. * SHOWPLAN permission on all databases containing objects referenced by the Transact-SQL statements.   For Transact-SQL statements that do not produce STATISTICS PROFILE or STATISTICS XML result sets, only the appropriate permissions to execute the Transact-SQL statements are required. For Transact-SQL statements that do produce STATISTICS PROFILE or STATISTICS XML result sets, checks for both the Transact-SQL statement execution permission and the SHOWPLAN permission must succeed, or the Transact-SQL statement execution is aborted and no Showplan information is generated. For information about which Transact-SQL statements produce Showplan information, see [Transact-SQL Statements That Produce Showplans](http://msdn.microsoft.com/en-us/library/ms187886(v=sql.105).aspx). |
| SET STATISTICS TIME  SET STATISTICS IO | * Appropriate permissions to execute the Transact-SQL statements.   Neither of these SET statement options check for or require the SHOWPLAN permission. |

### When Is the SHOWPLAN Permission Checked?

The SHOWPLAN permission is checked when a Transact-SQL statement or batch executes and Showplan information is generated. The check does not occur when a Showplan SET option is set to ON.

|  |
| --- |
| **NoteNote** |
| The context database for a Transact-SQL batch is set by using a USE <database\_name> statement. The SHOWPLAN permission is not checked on USE <database\_name> statements and is not checked on the context database. |

For more information about the Showplan SET statement options, see the following topics:

* [SET SHOWPLAN\_XML (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms187757(v=sql.105).aspx)
* [SET SHOWPLAN\_ALL (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms187735(v=sql.105).aspx)
* [SET SHOWPLAN\_TEXT (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms176058(v=sql.105).aspx)
* [SET STATISTICS XML (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms176107(v=sql.105).aspx)
* [SET STATISTICS PROFILE (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms188752(v=sql.105).aspx)
* [SET STATISTICS TIME (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms190287(v=sql.105).aspx)
* [SET STATISTICS IO (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms184361(v=sql.105).aspx)

[Permissions Required to Display Graphical Execution Plans by Using SQL Server Management Studio](javascript:void(0))

The permissions required to display graphical execution plans in SQL Server Management Studio are listed in the following table:

|  |  |
| --- | --- |
| **Management Studio Execution Plan option** | **Permissions Required** |
| Display Estimated Execution Plan | Requires the same permissions needed to use the SHOWPLAN\_XML SET statement option |
| Include Actual Execution Plan | Requires the same permissions needed to use the STATISTICS XML SET statement option |

For more information, see [Displaying Graphical Execution Plans (SQL Server Management Studio)](http://msdn.microsoft.com/en-us/library/ms178071(v=sql.105).aspx).

[Permissions Required to Display Execution Plans by Using SQL Server Profiler Event Classes](javascript:void(0))

To display execution plans by using SQL Server Profiler event classes, users must be a member of the **sysadmin** fixed server role, or be granted the ALTER TRACE permission. The SHOWPLAN permission is not checked nor is it required.

For more information, see [Displaying Execution Plans by Using SQL Server Profiler Event Classes](http://msdn.microsoft.com/en-us/library/ms190233(v=sql.105).aspx).

[Permissions Required to Display Execution Plans by Using the sys.dm\_exec\_query\_plan Dynamic Management Function](javascript:void(0))

To display execution plans by using the **sys.dm\_exec\_query\_plan** dynamic management function, users must be granted the VIEW SERVER STATE permission only.

For more information, see [sys.dm\_exec\_query\_plan (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms189747(v=sql.105).aspx).

# XML Showplans

**SQL Server 2008 R2**

[Other Versions](javascript:;)



* [SQL Server 2008](http://msdn.microsoft.com/en-us/library/ms189298(d=printer,v=sql.100).aspx)
* [SQL Server 2005](http://msdn.microsoft.com/en-us/library/ms189298(d=printer,v=sql.90).aspx)

In Microsoft SQL Server, Showplan execution plan output can be generated in XML format by various methods. Showplan output in XML format can be moved from one computer to another and thus rendered on any computer, even on computers where SQL Server is not installed. Showplan output in XML format can also be programmatically processed using XML technologies, such as XPath, XQuery, XSLT, SAX, DOM, and so on. XML Showplan processing is supported in SQL Server, which contains a built-in query evaluation engine for XPath and XQuery.

You can generate XML Showplan output by using the following methods:

* Selecting Display Estimated Execution Plan or Include Actual Execution Plan from the query editor toolbar in SQL Server Management Studio
* Using the Transact-SQL Showplan SET statement options SHOWPLAN\_XML and STATISTICS XML
* Selecting the SQL Server Profiler event classes **Showplan XML**, **Showplan XML for Query Compile**, and **Showplan XML Statistics Profile** for tracing
* Using the **sys.dm\_exec\_query\_plan** dynamic management function

XML Showplans are returned in the **nvarchar(max)** data type for all of these methods, except when you use **sys.dm\_exec\_query\_plan**. XML Showplans are returned in the **xml** data type when you use this dynamic management view.

The XML schema for Showplan is available with the SQL Server installation files at the following location:

C:\Program Files\Microsoft SQL Server\100\Tools\Binn\schemas\sqlserver\2004\07\showplan\showplanxml.xsd

|  |
| --- |
| **NoteNote** |
| If the query optimizer prematurely terminates query optimization, the StatementOptmEarlyAbortReason attribute is returned for the StmtSimple element in XML Showplan output. The possible values that can display for this attribute are **TimeOut**, **GoodEnoughPlanFound**, and **MemoryLimitExceeded**. If **TimeOut** or **GoodEnoughPlanFound** are returned for this attribute, no action is necessary. The Showplan returned contains correct results.  If **MemoryLimitExceeded** is returned for the StatementOptmEarlyAbortReason attribute, the XML Showplan produced will still be correct, but it may not be optimal. Try one of the following methods to increase available memory: 1) Reduce the load on the server. 2) Increase memory available to SQL Server. For more information, see [Managing Memory for Large Databases](http://msdn.microsoft.com/en-us/library/ms191481(v=sql.105).aspx). 3) Check the **max server memory** option that is set with **sp\_configure**, and increase the value if it is too low. For more information, see [Server Memory Options](http://msdn.microsoft.com/en-us/library/ms178067(v=sql.105).aspx). |

[About the Showplan XML Schema](javascript:void(0))

Version designation of the Showplan XML Schema consists of two parts, such as m.n, where m is the major version number and n is the minor version number. For example, "Version 2.5." This version number appears in the Showplan XML Schema document root element. For example: version="0.5"

### Backward Compatibility of Showplan XML Schemas

When the major version number increments, new XML Showplan output may not validate against the old schema. However, if the minor version number increments, there is no effect. For example, if you have generated XML Showplan output with schema version 0.5, that output validates against a Showplan XML schema version 0.6.

[Encoding of XML Showplans](javascript:void(0))

SQL Server sends the XML Showplan output to the client in Unicode format using two bytes for each character sent. The encoding rules are as follows:

* **If** (char >= 0x0020 && char <= 0xD7FF) or (char == 0x0009) or

(char == 0x000A) or (char == 0x000D) or (char >= 0xE000 && char <= 0xFFFD)

**then** send the char as 2 bytes, **else** send '?'

* Tab, newline, and linefeed characters are encoded as follows:
  + \t is encoded as '&#x9;'
  + \n is encoded as '&#xa;'
  + \r is encoded as '&#xd;'

[Saving XML Showplan Output to a File](javascript:void(0))

After Showplan output has been generated by using either the SHOWPLAN\_XML or the STATISTICS XML Transact-SQL SET statements, you can save the output to a file with the extension **.sqlplan**. For example, **MyXMLShowplan.sqlplan**. These **.sqlplan** files can then be opened and viewed in SQL Server Management Studio. For more information, see [How to: Save an Execution Plan in XML Format](http://msdn.microsoft.com/en-us/library/ms190646(v=sql.105).aspx).

# Transact-SQL Statements That Produce Showplans

**SQL Server 2008 R2**

[Other Versions](javascript:;)



* [SQL Server 2008](http://msdn.microsoft.com/en-us/library/ms187886(d=printer,v=sql.100).aspx)
* [SQL Server 2005](http://msdn.microsoft.com/en-us/library/ms187886(d=printer,v=sql.90).aspx)

This topic lists the types of Transact-SQL statements that produce Showplan output for Showplan SET options and SQL Server Profiler event classes.

|  |
| --- |
| **NoteNote** |
| No Showplan information is generated for encrypted stored procedures or for triggers. |

The following table lists which Transact-SQL statements produce Showplan information.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Transact-SQL statement type** | **Showplan SET Option mode1** | **Statistics SET Option mode2** | **Query Compile SQL Server Profiler events3** | **Query Execute SQL Server Profiler events 4** |
| All DML (SELECT, INSERT, UPDATE, and DELETE) | Yes | Yes | Yes | Yes |
| All DDL that affects metadata only. For example, CREATE TABLE.  Exceptions are listed separately. | No | No | No | No |
| CREATE INDEX | No | Yes, if table is not empty. | Yes | Yes, if table is not empty. |
| INSERT INTO ... SELECT FROM  (subqueries) | Yes | Yes | Yes | Yes |
| INSERT INTO ... EXEC  (subqueries) | No | No | Yes | Yes |
| Automatically created or updated statistics | No | No | Yes | Yes |
| Manually created or updated statistics | No | No | Yes | Yes |
| Dynamic SQL | No | Yes | Yes | Yes |
| EXECUTE stored\_procedure | Yes | Yes | Yes | Yes |
| CREATE PROCEDURE store\_procedure | No | No | No | No |
| Triggers  (Not called directly, but caused by an INSERT, UPDATE, or DELETE statement) | No | Yes | Yes | Yes |
| CLR triggers | No | No | No | No |
| CLR user-defined functions, user-defined aggregates, and user-defined procedures | No | No | No | No |
| Queries that reference a user-defined function one or more times | Yes  (No for Microsoft SQL Server 2000) | Yes | Yes  (No for SQL Server 2000) | Yes |
| Create a temporary table, and then SELECT from it, or INSERT INTO it | Yes  (No for SQL Server 2000) | Yes | Yes  (No for SQL Server 2000) | Yes |
| DBCC commands | No | No | No | No |
| BULK INSERT | No | Yes | Yes | Yes |
| Statements submitted by using the **sp\_executesql** stored procedure | No | No | No | No |

1 Includes the following Showplan SET option statements:

* SET SHOWPLAN\_XML ON
* SET SHOWPLAN\_ALL ON
* SET SHOWPLAN\_TEXT ON

2 Includes the following Statistics SET option statements:

* SET STATISTICS XML ON
* SET STATISTICS PROFILE ON

3 Includes the following SQL Server Profiler event classes:

* **Showplan XML For Query Compile**
* **Showplan All For Query Compile**

4 Includes the following SQL Server Profiler event classes:

* **Showplan XML**
* **Showplan All**
* **Showplan Text**
* **Showplan XML Statistics Profile**
* **Showplan Statistics Profile**

# Interpreting Execution Plans Containing Bitmap Filters

**SQL Server 2008 R2**

[Other Versions](javascript:;)



* [SQL Server 2008](http://msdn.microsoft.com/en-us/library/bb510541(d=printer,v=sql.100).aspx)

2 out of 2 rated this helpful - [Rate this topic](http://msdn.microsoft.com/en-us/library/bb510541(d=printer,v=sql.105).aspx#feedback)

Parallel query execution plans that use bitmap filtering have a [Bitmap](http://msdn.microsoft.com/en-us/library/ms190638(v=sql.105).aspx) operator in one or more operator subtrees. A bitmap filter uses a compact representation of a set of values from a table in one part of the operator tree to filter rows from a second table in another part of the tree. By removing unnecessary rows early in the query, subsequent operators have fewer rows to work with, and the overall performance of the query improves.

In SQL Server 2008, bitmap filtering can be introduced in the parallel query plan after optimization, as in SQL Server 2005, or introduced dynamically by the query optimizer during query plan generation. When the filter is introduced dynamically, it is referred to as an optimized bitmap filter. A query plan can contain both bitmap filters and optimized bitmap filters. The query optimizer determines when a bitmap filter or optimized bitmap filter is selective enough to be useful and in which operators the filter is applied. For more information, see [Optimizing Data Warehouse Query Performance Through Bitmap Filtering](http://msdn.microsoft.com/en-us/library/bb522541(v=sql.105).aspx).

When analyzing an execution plan containing bitmap filtering, it is important to understand how the data flows through the plan and where filtering is applied. The bitmap filter and optimized bitmap is created on the build input (the dimension table) side of a [hash join](http://msdn.microsoft.com/en-us/library/ms189582(v=sql.105).aspx); however, the actual filtering is typically done within the [Parallelism](http://msdn.microsoft.com/en-us/library/ms191528(v=sql.105).aspx) operator, which is on the probe input (the fact table) side of the hash join. However, when the bitmap filter is based on an integer column, the filter can be applied directly to the initial table or index scan operation rather than the Parallelism operator. This technique is called in-row optimization.

[Viewing Bitmap Filters in Showplans](javascript:void(0))

To view bitmap filters in the query plan, use the SET options SHOWPLAN\_ALL, SHOWPLAN\_TEXT, or SHOWPLAN\_XML, or click Include Actual Execution Plan in SQL Server Management Studio.

If an XML Showplan is produced, the physical and logical bitmap operators are listed in the following way:

<RelOp NodeId="2" PhysicalOp="Bitmap" LogicalOp="Bitmap Create" EstimateRows="88" EstimateIO="0" EstimateCPU="0.0718125" AvgRowSize="6893" EstimatedTotalSubtreeCost="0.229385" Parallel="1" EstimateRebinds="0" EstimateRewinds="0">

The operator in which a bitmap filter is applied contains the name of the bitmap in the Probe Column property.

The operator in which optimized bitmap filter is applied contains a bitmap predicate in the form of PROBE([Opt\_Bitmap1001], {[column\_name]} [, 'IN ROW']). The bitmap predicate reports on the following information:

* The bitmap name that corresponds to the name introduced in the Bitmap Create operator. The prefix 'Opt\_' indicates that an optimized bitmap filter is used.
* The column probed against. This is the point from which the filtered data flows through the tree.
* Whether the bitmap probe is performed in-row. When in-row optimization is used, the bitmap probe is invoked with the IN ROW parameter. Otherwise, this parameter is missing.

[Example](javascript:void(0))

The following example demonstrates how optimized bitmap filtering is used in an execution plan. The two dimension tables DimProduct and DimCustomer join to the fact table FactInternetSales using a primary-key-to-foreign-key join on a single integer column.

[Copy](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_696dc486-372b-4d47-bd67-9acfe036d381');)

USE AdventureWorksDW2008R2;

GO

SELECT \*

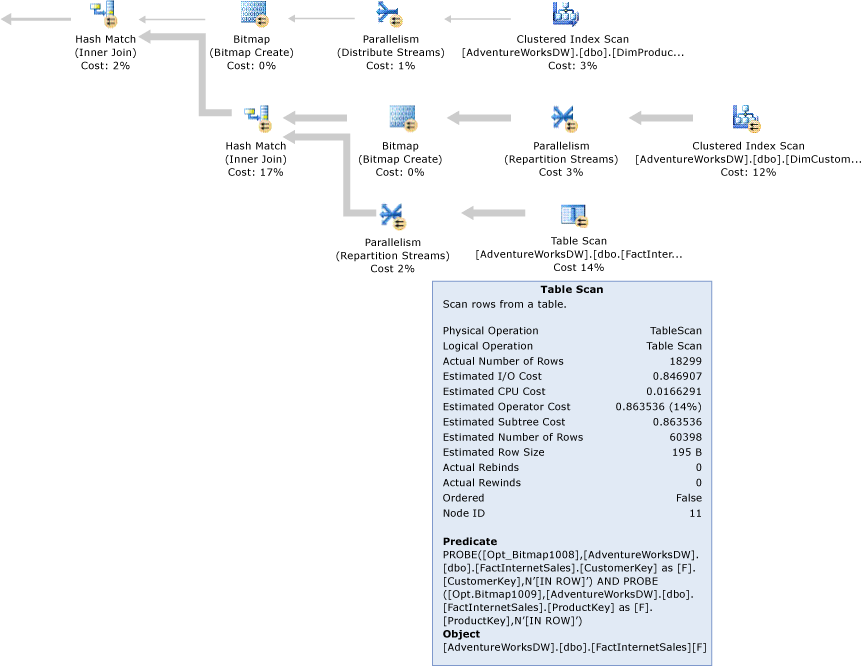
FROM dbo.FactInternetSales AS F

INNER JOIN dbo.DimProduct AS D1 ON F.ProductKey = D1.ProductKey

INNER JOIN dbo.DimCustomer AS D2 ON F.CustomerKey = D2.CustomerKey

WHERE D1.StandardCost <= 30 AND D2.YearlyIncome <= 50000;

The following illustration shows that an optimized bitmap filter is created in the operator subtree of both dimension tables after the dimension table has been scanned and the information needed to eliminate non-qualifying rows from the fact table is known. The filter is then applied to the fact table in the earliest possible location, the [Table Scan](http://msdn.microsoft.com/en-us/library/ms181129(v=sql.105).aspx) operator. The application of the filter is shown in the Predicate section of the Table Scan properties. The information shown in the Predicate indicates that both optimized bitmap filters Opt\_Bitmap1008 and Opt\_Bitmap1009 are used to limit the rows returned from the fact table. The columns probed against in the fact table are listed as [F].[CustomerKey] and [F].[ProductKey]. The IN ROW parameter is shown, indicating that in-row optimization is used in the process. If in-row optimization is not possible, the bitmap filtering is applied to the Parallelism operator.



Based on the illustration, the following conclusions can be made:

* Optimized bitmap filters are created in two subtrees.
* Both filters are applied dynamically to a single (Table Scan) operator.
* The optimized bitmap filter estimated to be the most selective is implemented first.
* The columns on which the dimension tables are joined to the fact table allows the use of in-row optimization. That is, the join is based on a single integer column.
* The filter is applied at the earliest possible point in the query, resulting in a reduced number of rows flowing from the Table Scan operation through the remaining operators in the tree.

# Logical and Physical Operators Reference

**SQL Server 2008 R2**

[Other Versions](javascript:;)



* [SQL Server 2008](http://msdn.microsoft.com/en-us/library/ms191158(d=printer,v=sql.100).aspx)
* [SQL Server 2005](http://msdn.microsoft.com/en-us/library/ms191158(d=printer,v=sql.90).aspx)
* [SQL Server 2012](http://msdn.microsoft.com/en-us/library/ms191158(d=printer,v=sql.110).aspx)

Operators describe how SQL Server executes a query or a Data Manipulation Language (DML) statement. The query optimizer uses operators to build a query plan to create the result specified in the query, or to perform the operation specified in the DML statement. The query plan is a tree consisting of physical operators.

Operators are classified as logical and physical operators. Logical operators describe a relational query processing operation on a conceptual level. Physical operators actually implement the operation defined by a logical operator using a concrete method or algorithm. For example, "join" is a logical operation, whereas "nested loops joins" is a physical operator.

**Logical Operators**

Logical operators describe the relational algebraic operation used to process a statement. In other words, logical operators describe conceptually what operation needs to be performed.

**Physical Operators**

Physical operators implement the operation described by logical operators. Each of the physical operators is an object or routine that performs an operation. For example, some physical operators access columns or rows from a table, index or view. Other physical operators perform other types of operations: The **Aggregate** operator calculates an expression containing MIN, MAX, SUM, COUNT or AVG, and the **Merge Join** operator performs different types of logical join operations.

The physical operators initialize, collect data, and close. Specifically, the physical operator can answer the following three method calls:

* **Init()**: The **Init()** method causes a physical operator to initialize itself and set up any required data structures. The physical operator may receive many **Init()** calls, though typically a physical operator receives only one.
* **GetNext()**: The **GetNext()** method causes a physical operator to get the first, or subsequent row of data. The physical operator may receive zero or many **GetNext()** calls.
* **Close()**: The **Close()** method causes a physical operator to perform some clean-up operations and shut itself down. A physical operator only receives one **Close()** call.

The **GetNext()** method returns one row of data, and the number of times it is called appears as **ActualRows** in the Showplan output that is produced by using SET STATISTICS PROFILE ON or SET STATISTICS XML ON. For more information about these SET options, see [SET STATISTICS PROFILE (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms188752(v=sql.105).aspx) and [SET STATISTICS XML (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms176107(v=sql.105).aspx).

The **ActualRebinds** and **ActualRewinds** counts that appear in Showplan output refer to the number of times that the **Init()** method is called. Unless an operator is on the inner side of a loop join, **ActualRebinds** equals one and **ActualRewinds** equals zero. If an operator is on the inner side of a loop join, the sum of the number of rebinds and rewinds should equal the number of rows processed on the outer side of the join. A rebind means that one or more of the correlated parameters of the join changed and the inner side must be reevaluated. A rewind means that none of the correlated parameters changed and the prior inner result set may be reused.

**ActualRebinds** and **ActualRewinds** are present in XML Showplan output produced by using SET STATISTICS XML ON. They are only populated for the **Nonclustered Index Spool**, **Remote Query**, **Row Count Spool**, **Sort**, **Table Spool**, and **Table-valued Function** operators. **ActualRebinds** and **ActualRewinds** may also be populated for the **Assert** and **Filter** operators when the **StartupExpression** attribute is set to TRUE.

When **ActualRebinds** and **ActualRewinds** are present in an XML Showplan, they are comparable to **EstimateRebinds** and **EstimateRewinds**. When they are absent, the estimated number of rows (**EstimateRows**) is comparable to the actual number of rows (**ActualRows**). Note that actual graphical Showplan output displays zeros for the actual rebinds and actual rewinds when they are absent. For more information about graphical Showplans, see [Displaying Graphical Execution Plans (SQL Server Management Studio)](http://msdn.microsoft.com/en-us/library/ms178071(v=sql.105).aspx).

A related counter, **ActualEndOfScans**, is available only when Showplan output is produced by using SET STATISTICS XML ON. Whenever a physical operator reaches the end of its data stream, this counter is incremented by one. A physical operator can reach the end of its data stream zero, one, or multiple times. As with rebinds and rewinds, the number of end of scans can be more than one only if the operator is on the inner side of a loop join. The number of end of scans should be less than or equal to the sum of the number of rebinds and rewinds.

Physical operators correspond to execution algorithms. Examples of physical operators include index scan/seek, nested loop join, merge join, hash join/aggregation, stream aggregation. Physical operators have costs associated with them. Each step in the execution of a query or DML statement involves a physical operator.

[Mapping Physical and Logical Operators](javascript:void(0))

The query optimizer creates a query plan as a tree consisting of logical operators. After the query optimizer creates the plan, the query optimizer chooses the most efficient physical operator for each logical operator. The query optimizer uses a cost-based approach to determine which physical operator will implement a logical operator.

Usually, a logical operation can be implemented by multiple physical operators. However, in rare cases, a physical operator can implement multiple logical operations as well.

[In This Section](javascript:void(0))

This section contains descriptions of the following logical and physical operators:

|  |  |
| --- | --- |
| [Aggregate](http://msdn.microsoft.com/en-us/library/ms190472(v=sql.105).aspx) | [Merge Interval](http://msdn.microsoft.com/en-us/library/ms187840(v=sql.105).aspx) |
| [Arithmetic Expression](http://msdn.microsoft.com/en-us/library/ms187899(v=sql.105).aspx) | [Merge Join](http://msdn.microsoft.com/en-us/library/ms189961(v=sql.105).aspx) |
| [Assert](http://msdn.microsoft.com/en-us/library/ms187603(v=sql.105).aspx) | [Nested Loops](http://msdn.microsoft.com/en-us/library/ms187871(v=sql.105).aspx) |
| [Assign](http://msdn.microsoft.com/en-us/library/ms191525(v=sql.105).aspx) | [Nonclustered Index Delete](http://msdn.microsoft.com/en-us/library/ms191319(v=sql.105).aspx) |
| [Async Concat](http://msdn.microsoft.com/en-us/library/ms190751(v=sql.105).aspx) | [Nonclustered Index Insert](http://msdn.microsoft.com/en-us/library/ms188227(v=sql.105).aspx) |
| [Bitmap](http://msdn.microsoft.com/en-us/library/ms190638(v=sql.105).aspx) | [Nonclustered Index Scan](http://msdn.microsoft.com/en-us/library/ms178038(v=sql.105).aspx) |
| [Bitmap Create](http://msdn.microsoft.com/en-us/library/ms189304(v=sql.105).aspx) | [Nonclustered Index Seek](http://msdn.microsoft.com/en-us/library/ms190376(v=sql.105).aspx) |
| [Bookmark Lookup](http://msdn.microsoft.com/en-us/library/ms180920(v=sql.105).aspx) | [Nonclustered Index Spool](http://msdn.microsoft.com/en-us/library/ms189611(v=sql.105).aspx) |
| [Branch Repartition](http://msdn.microsoft.com/en-us/library/ms191542(v=sql.105).aspx) | [Nonclustered Index Update](http://msdn.microsoft.com/en-us/library/ms177417(v=sql.105).aspx) |
| [Broadcast](http://msdn.microsoft.com/en-us/library/ms187497(v=sql.105).aspx) | [Online Index Insert](http://msdn.microsoft.com/en-us/library/ms188633(v=sql.105).aspx) |
| [Cache](http://msdn.microsoft.com/en-us/library/ms191249(v=sql.105).aspx) | [Parallelism](http://msdn.microsoft.com/en-us/library/ms191528(v=sql.105).aspx) |
| [Clustered Index Delete](http://msdn.microsoft.com/en-us/library/ms187951(v=sql.105).aspx) | [Parameter Table Scan](http://msdn.microsoft.com/en-us/library/ms189951(v=sql.105).aspx) |
| [Clustered Index Insert](http://msdn.microsoft.com/en-us/library/ms190962(v=sql.105).aspx) | [Partial Aggregate](http://msdn.microsoft.com/en-us/library/ms190683(v=sql.105).aspx) |
| [Clustered Index Merge](http://msdn.microsoft.com/en-us/library/bb522464(v=sql.105).aspx) | [Population Query](http://msdn.microsoft.com/en-us/library/ms189937(v=sql.105).aspx) |
| [Clustered Index Scan](http://msdn.microsoft.com/en-us/library/ms175184(v=sql.105).aspx) | [Refresh Query](http://msdn.microsoft.com/en-us/library/ms187084(v=sql.105).aspx) |
| [Clustered Index Seek](http://msdn.microsoft.com/en-us/library/ms190400(v=sql.105).aspx) | [Remote Delete](http://msdn.microsoft.com/en-us/library/ms175872(v=sql.105).aspx) |
| [Clustered Index Update](http://msdn.microsoft.com/en-us/library/ms177434(v=sql.105).aspx) | [Remote Index Scan](http://msdn.microsoft.com/en-us/library/cc280566(v=sql.105).aspx) |
| [Collapse](http://msdn.microsoft.com/en-us/library/ms191512(v=sql.105).aspx) | [Remote Index Seek](http://msdn.microsoft.com/en-us/library/cc280498(v=sql.105).aspx) |
| [Compute Scalar](http://msdn.microsoft.com/en-us/library/ms178082(v=sql.105).aspx) | [Remote Insert](http://msdn.microsoft.com/en-us/library/ms190932(v=sql.105).aspx) |
| [Concatenation](http://msdn.microsoft.com/en-us/library/ms187919(v=sql.105).aspx) | [Remote Query](http://msdn.microsoft.com/en-us/library/ms189297(v=sql.105).aspx) |
| [Constant Scan](http://msdn.microsoft.com/en-us/library/ms188318(v=sql.105).aspx) | [Remote Scan](http://msdn.microsoft.com/en-us/library/ms178103(v=sql.105).aspx) |
| [Convert](http://msdn.microsoft.com/en-us/library/ms190660(v=sql.105).aspx) | [Remote Update](http://msdn.microsoft.com/en-us/library/ms180908(v=sql.105).aspx) |
| [Cross Join](http://msdn.microsoft.com/en-us/library/ms187628(v=sql.105).aspx) | [Repartition Streams](http://msdn.microsoft.com/en-us/library/ms190783(v=sql.105).aspx) |
| [Cursor Catchall](http://msdn.microsoft.com/en-us/library/ms188324(v=sql.105).aspx) | [Result](http://msdn.microsoft.com/en-us/library/ms190966(v=sql.105).aspx) |
| [Declare](http://msdn.microsoft.com/en-us/library/ms189246(v=sql.105).aspx) | [RID Lookup](http://msdn.microsoft.com/en-us/library/ms190696(v=sql.105).aspx) |
| [Delete](http://msdn.microsoft.com/en-us/library/ms175494(v=sql.105).aspx) | [Right Anti Semi Join](http://msdn.microsoft.com/en-us/library/ms177472(v=sql.105).aspx) |
| [Deleted Scan](http://msdn.microsoft.com/en-us/library/ms188244(v=sql.105).aspx) | [Right Outer Join](http://msdn.microsoft.com/en-us/library/ms190390(v=sql.105).aspx) |
| [Distinct](http://msdn.microsoft.com/en-us/library/ms178098(v=sql.105).aspx) | [Right Semi Join](http://msdn.microsoft.com/en-us/library/ms187860(v=sql.105).aspx) |
| [Distinct Sort](http://msdn.microsoft.com/en-us/library/ms177486(v=sql.105).aspx) | [Row Count Spool](http://msdn.microsoft.com/en-us/library/ms190991(v=sql.105).aspx) |
| [Distribute Streams](http://msdn.microsoft.com/en-us/library/ms189914(v=sql.105).aspx) | [Segment](http://msdn.microsoft.com/en-us/library/ms180774(v=sql.105).aspx) |
| [Dynamic](http://msdn.microsoft.com/en-us/library/ms191312(v=sql.105).aspx) | [Segment Repartition](http://msdn.microsoft.com/en-us/library/ms190937(v=sql.105).aspx) |
| [Eager Spool](http://msdn.microsoft.com/en-us/library/ms190435(v=sql.105).aspx) | [Sequence](http://msdn.microsoft.com/en-us/library/ms190228(v=sql.105).aspx) |
| [Fetch Query](http://msdn.microsoft.com/en-us/library/ms186269(v=sql.105).aspx) | [SequenceProject](http://msdn.microsoft.com/en-us/library/ms187041(v=sql.105).aspx) |
| [Filter](http://msdn.microsoft.com/en-us/library/ms175020(v=sql.105).aspx) | [Snapshot](http://msdn.microsoft.com/en-us/library/ms175057(v=sql.105).aspx) |
| [Flow Distinct](http://msdn.microsoft.com/en-us/library/ms177473(v=sql.105).aspx) | [Sort](http://msdn.microsoft.com/en-us/library/ms186318(v=sql.105).aspx) |
| [Full Outer Join](http://msdn.microsoft.com/en-us/library/ms191494(v=sql.105).aspx) | [Split](http://msdn.microsoft.com/en-us/library/ms188309(v=sql.105).aspx) |
| [Gather Streams](http://msdn.microsoft.com/en-us/library/ms187002(v=sql.105).aspx) | [Spool](http://msdn.microsoft.com/en-us/library/ms188301(v=sql.105).aspx) |
| [Hash Match](http://msdn.microsoft.com/en-us/library/ms189582(v=sql.105).aspx) | [Stream Aggregate](http://msdn.microsoft.com/en-us/library/ms189907(v=sql.105).aspx) |
| [Hash Match Root](http://msdn.microsoft.com/en-us/library/ms187652(v=sql.105).aspx) | [Switch](http://msdn.microsoft.com/en-us/library/ms187852(v=sql.105).aspx) |
| [Hash Match Team](http://msdn.microsoft.com/en-us/library/ms178097(v=sql.105).aspx) | [Table Delete](http://msdn.microsoft.com/en-us/library/ms187573(v=sql.105).aspx) |
| [If](http://msdn.microsoft.com/en-us/library/ms191306(v=sql.105).aspx) | [Table Insert](http://msdn.microsoft.com/en-us/library/ms175086(v=sql.105).aspx) |
| [Inner Join](http://msdn.microsoft.com/en-us/library/ms187106(v=sql.105).aspx) | [Table Merge](http://msdn.microsoft.com/en-us/library/bb500170(v=sql.105).aspx) |
| [Insert](http://msdn.microsoft.com/en-us/library/ms188286(v=sql.105).aspx) | [Table Scan](http://msdn.microsoft.com/en-us/library/ms181129(v=sql.105).aspx) |
| [Inserted Scan](http://msdn.microsoft.com/en-us/library/ms189101(v=sql.105).aspx) | [Table Spool](http://msdn.microsoft.com/en-us/library/ms181032(v=sql.105).aspx) |
| [Intrinsic](http://msdn.microsoft.com/en-us/library/ms191298(v=sql.105).aspx) | [Table Update](http://msdn.microsoft.com/en-us/library/ms186368(v=sql.105).aspx) |
| [Iterator Catchall](http://msdn.microsoft.com/en-us/library/ms189972(v=sql.105).aspx) | [Table-Valued Function](http://msdn.microsoft.com/en-us/library/ms190769(v=sql.105).aspx) |
| [Key Lookup](http://msdn.microsoft.com/en-us/library/bb326635(v=sql.105).aspx) | [Top](http://msdn.microsoft.com/en-us/library/ms177432(v=sql.105).aspx) |
| [Keyset](http://msdn.microsoft.com/en-us/library/ms189589(v=sql.105).aspx) | [Top N Sort](http://msdn.microsoft.com/en-us/library/ms189054(v=sql.105).aspx) |
| [Language Element Catchall](http://msdn.microsoft.com/en-us/library/ms189868(v=sql.105).aspx) | [UDX](http://msdn.microsoft.com/en-us/library/ms190178(v=sql.105).aspx) |
| [Lazy Spool](http://msdn.microsoft.com/en-us/library/ms191221(v=sql.105).aspx) | [Union](http://msdn.microsoft.com/en-us/library/ms186962(v=sql.105).aspx) |
| [Left Anti Semi Join](http://msdn.microsoft.com/en-us/library/ms191171(v=sql.105).aspx) | [Update](http://msdn.microsoft.com/en-us/library/ms187026(v=sql.105).aspx) |
| [Left Outer Join](http://msdn.microsoft.com/en-us/library/ms191463(v=sql.105).aspx) | [While](http://msdn.microsoft.com/en-us/library/ms190797(v=sql.105).aspx) |
| [Left Semi Join](http://msdn.microsoft.com/en-us/library/ms191218(v=sql.105).aspx) | [Cursor Logical and Physical Operators](http://msdn.microsoft.com/en-us/library/ms181033(v=sql.105).aspx) |
| [Log Row Scan](http://msdn.microsoft.com/en-us/library/ms187600(v=sql.105).aspx) |  |

# Execution Plan Caching and Reuse

**SQL Server 2008 R2**

[Other Versions](javascript:;)



* [SQL Server 2008](http://msdn.microsoft.com/en-us/library/ms181055(d=printer,v=sql.100).aspx)
* [SQL Server 2005](http://msdn.microsoft.com/en-us/library/ms181055(d=printer,v=sql.90).aspx)

SQL Server has a pool of memory that is used to store both execution plans and data buffers. The percentage of the pool allocated to either execution plans or data buffers fluctuates dynamically, depending on the state of the system. The part of the memory pool that is used to store execution plans is referred to as the procedure cache.

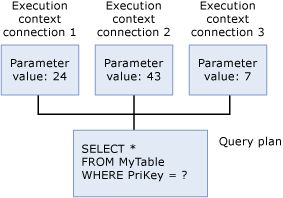
SQL Server execution plans have the following main components:

* Query Plan

The bulk of the execution plan is a re-entrant, read-only data structure used by any number of users. This is referred to as the query plan. No user context is stored in the query plan. There are never more than one or two copies of the query plan in memory: one copy for all serial executions and another for all parallel executions. The parallel copy covers all parallel executions, regardless of their degree of parallelism.

* Execution Context

Each user that is currently executing the query has a data structure that holds the data specific to their execution, such as parameter values. This data structure is referred to as the execution context. The execution context data structures are reused. If a user executes a query and one of the structures is not being used, it is reinitialized with the context for the new user.



When any SQL statement is executed in SQL Server, the relational engine first looks through the procedure cache to verify that an existing execution plan for the same SQL statement exists. SQL Server reuses any existing plan it finds, saving the overhead of recompiling the SQL statement. If no existing execution plan exists, SQL Server generates a new execution plan for the query.

SQL Server has an efficient algorithm to find any existing execution plans for any specific SQL statement. In most systems, the minimal resources that are used by this scan are less than the resources that are saved by being able to reuse existing plans instead of compiling every SQL statement.

The algorithms to match new SQL statements to existing, unused execution plans in the cache require that all object references be fully qualified. For example, the first of these SELECT statements is not matched with an existing plan, and the second is matched:

SELECT \* FROM Person;

SELECT \* FROM Person.Person;

[Removing Execution Plans from the Procedure Cache](javascript:void(0))

Execution plans remain in the procedure cache as long as there is enough memory to store them. When memory pressure exists, the Database Engine uses a cost-based approach to determine which execution plans to remove from the procedure cache. To make a cost-based decision, the Database Engine increases and decreases a current cost variable for each execution plan according to the following factors.

When a user process inserts an execution plan into the cache, the user process sets the current cost equal to the original query compile cost; for ad-hoc execution plans, the user process sets the current cost to zero. Thereafter, each time a user process references an execution plan, it resets the current cost to the original compile cost; for ad-hoc execution plans the user process increases the current cost. For all plans, the maximum value for the current cost is the original compile cost.

When memory pressure exists, the Database Engine responds by removing execution plans from the procedure cache. To determine which plans to remove, the Database Engine repeatedly examines the state of each execution plan and removes plans when their current cost is zero. An execution plan with zero current cost is not removed automatically when memory pressure exists; it is removed only when the Database Engine examines the plan and the current cost is zero. When examining an execution plan, the Database Engine pushes the current cost towards zero by decreasing the current cost if a query is not currently using the plan.

The Database Engine repeatedly examines the execution plans until enough have been removed to satisfy memory requirements. While memory pressure exists, an execution plan may have its cost increased and decreased more than once. When memory pressure no longer exists, the Database Engine stops decreasing the current cost of unused execution plans and all execution plans remain in the procedure cache, even if their cost is zero.

The Database Engine uses the resource monitor and user threads to free memory from the procedure cache in response to memory pressure. The resource monitor and user threads can examine plans run concurrently to decrease the current cost for each unused execution plan. The resource monitor removes execution plans from the procedure cache when global memory pressure exists. It frees memory to enforce policies for system memory, process memory, resource pool memory, and maximum size for all caches.

The maximum size for all caches is a function of the buffer pool size and cannot exceed the maximum server memory. For more information on configuring the maximum server memory, see the **max server memory** setting in [sp\_configure (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms188787(v=sql.105).aspx).

The user threads remove execution plans from the procedure cache when single cache memory pressure exists. They enforce policies for maximum single cache size and maximum single cache entries.

The following examples illustrate which execution plans get removed from the procedure cache:

* An execution plan is frequently referenced so that its cost never goes to zero. The plan remains in the procedure cache and is not removed unless there is memory pressure and the current cost is zero.
* An ad-hoc execution plan is inserted and is not referenced again before memory pressure exists. Since ad-hoc plans are initialized with a current cost of zero, when the database engine examines the execution plan, it will see the zero current cost and remove the plan from the procedure cache. The ad-hoc execution plan remains in the procedure cache with a zero current cost when memory pressure does not exist.

To manually remove a single plan or all plans from the cache, use [DBCC FREEPROCCACHE (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms174283(v=sql.105).aspx).

[Recompiling Execution Plans](javascript:void(0))

Certain changes in a database can cause an execution plan to be either inefficient or invalid, based on the new state of the database. SQL Server detects the changes that invalidate an execution plan and marks the plan as not valid. A new plan must then be recompiled for the next connection that executes the query. The conditions that invalidate a plan include the following:

* Changes made to a table or view referenced by the query (ALTER TABLE and ALTER VIEW).
* Changes made to a single procedure, which would drop all plans for that procedure from the cache (ALTER PROCEDURE).
* Changes to any indexes used by the execution plan.
* Updates on statistics used by the execution plan, generated either explicitly from a statement, such as UPDATE STATISTICS, or generated automatically.
* Dropping an index used by the execution plan.
* An explicit call to **sp\_recompile**.
* Large numbers of changes to keys (generated by INSERT or DELETE statements from other users that modify a table referenced by the query).
* For tables with triggers, if the number of rows in the **inserted** or **deleted** tables grows significantly.
* Executing a stored procedure using the WITH RECOMPILE option.

Most recompilations are required either for statement correctness or to obtain potentially faster query execution plans.

In SQL Server 2000, whenever a statement within a batch causes recompilation, the whole batch, whether submitted through a stored procedure, trigger, ad-hoc batch, or prepared statement, is recompiled. In SQL Server 2005 and later, only the statement inside the batch that causes recompilation is recompiled. Because of this difference, recompilation counts in SQL Server 2000 and later releases are not comparable. Also, there are more types of recompilations in SQL Server 2005 and later because of its expanded feature set.

Statement-level recompilation benefits performance because, in most cases, a small number of statements causes recompilations and their associated penalties, in terms of CPU time and locks. These penalties are therefore avoided for the other statements in the batch that do not have to be recompiled.

The SQL Server Profiler **SP:Recompile** trace event reports statement-level recompilations. This trace event reports only batch recompilations in SQL Server 2000. Further, the **TextData** column of this event is populated. Therefore, the SQL Server 2000 practice of having to trace **SP:StmtStarting** or **SP:StmtCompleted** to obtain the Transact-SQL text that caused recompilation is no longer required.

The trace event **SQL:StmtRecompile** reports statement-level recompilations. This trace event can be used to track and debug recompilations. Whereas **SP:Recompile** generates only for stored procedures and triggers, **SQL:StmtRecompile** generates for stored procedures, triggers, ad-hoc batches, batches that are executed by using **sp\_executesql**, prepared queries, and dynamic SQL.

The **EventSubClass** column of **SP:Recompile** and **SQL:StmtRecompile** contains an integer code that indicates the reason for the recompilation. The following table contains the meaning of each code number.

|  |  |
| --- | --- |
| **EventSubClass value** | **Description** |
| 1 | Schema changed. |
| 2 | Statistics changed. |
| 3 | Deferred compile. |
| 4 | SET option changed. |
| 5 | Temporary table changed. |
| 6 | Remote rowset changed. |
| 7 | FOR BROWSE permission changed. |
| 8 | Query notification environment changed. |
| 9 | Partitioned view changed. |
| 10 | Cursor options changed. |
| 11 | OPTION (RECOMPILE) requested. |
| **NoteNote** | |
| When the AUTO\_UPDATE\_STATISTICS database option is SET to ON, queries are recompiled when they target tables or indexed views whose statistics have been updated or whose cardinalities have changed significantly since the last execution. This behavior applies to standard user-defined tables, temporary tables, and the inserted and deleted tables created by DML triggers. If query performance is affected by excessive recompilations, consider changing this setting to OFF. When the AUTO\_UPDATE\_STATISTICS database option is SET to OFF, no recompilations occur based on statistics or cardinality changes, with the exception of the inserted and deleted tables that are created by DML INSTEAD OF triggers. Because these tables are created in **tempdb**, the recompilation of queries that access them depends on the setting of AUTO\_UPDATE\_STATISTICS in **tempdb**. Note that in SQL Server 2000, queries continue to recompile based on cardinality changes to the DML trigger inserted and deleted tables, even when this setting is OFF. For more information about disabling AUTO\_UPDATE\_STATISTICS, see [Using Statistics to Improve Query Performance](http://msdn.microsoft.com/en-us/library/ms190397(v=sql.105).aspx). | |

**Finding Missing Indexes**

**SQL Server 2008 R2**

[Other Versions](javascript:;)



* [SQL Server 2008](http://msdn.microsoft.com/en-us/library/ms345417(d=printer,v=sql.100).aspx)
* [SQL Server 2005](http://msdn.microsoft.com/en-us/library/ms345417(d=printer,v=sql.90).aspx)

The missing indexes feature is a lightweight, always-on way to identify indexes missing on database tables and indexed views that might enhance query performance if implemented.

[In This Section](javascript:void(0))

[About the Missing Indexes Feature](http://msdn.microsoft.com/en-us/library/ms345524(v=sql.105).aspx)

Describes the components of the missing indexes feature and how to enable or disable this feature.

[Using Missing Index Information to Write CREATE INDEX Statements](http://msdn.microsoft.com/en-us/library/ms345405(v=sql.105).aspx)

Provides guidelines for and examples of using the information returned by the missing index feature components to write CREATE INDEX DDL statements.

[Limitations of the Missing Indexes Feature](http://msdn.microsoft.com/en-us/library/ms345485(v=sql.105).aspx)

Describes limitations and restrictions for using the missing indexes feature.

[Related Query Tuning Features](http://msdn.microsoft.com/en-us/library/ms345577(v=sql.105).aspx)

Lists other SQL Server features that can be used with the missing indexes feature to tune query performance.

**Query Tuning Recommendations**

**SQL Server 2008 R2**

[Other Versions](javascript:;)



* [SQL Server 2008](http://msdn.microsoft.com/en-us/library/ms188722(d=printer,v=sql.100).aspx)
* [SQL Server 2005](http://msdn.microsoft.com/en-us/library/ms188722(d=printer,v=sql.90).aspx)

Some queries consume more resources than others. For example, queries that return large result sets and those that contain WHERE clauses that are not unique are always resource intensive. No degree of query optimizer intelligence can eliminate the resource cost of these constructs when compared to a less complex query. SQL Server uses the optimal access plan, but query optimization is limited by what is possible.

Nonetheless, to improve query performance, you can:

* Add more memory. This solution can be especially helpful if the server runs many complex queries and several of the queries execute slowly.
* Use more than one processor. Multiple processors allow the Database Engine to make use of parallel queries. For more information, see [Parallel Query Processing](http://msdn.microsoft.com/en-us/library/ms178065(v=sql.105).aspx).
* Rewrite the query. Consider the following issues:
  + If the query uses cursors, determine if the cursor query could be written using either a more efficient cursor type (such as fast forward-only) or a single query. Single queries typically outperform cursor operations. Because a set of cursor statements is typically an outer loop operation, in which each row in the outer loop is processed once using an inner statement, consider using either a GROUP BY or CASE statement or a subquery instead. For more information, see [Cursor Types (Database Engine)](http://msdn.microsoft.com/en-us/library/ms188644(v=sql.105).aspx) and [Query Fundamentals](http://msdn.microsoft.com/en-us/library/ms190659(v=sql.105).aspx).
  + If an application uses a loop, consider putting the loop inside the query. Often an application contains a loop that contains a parameterized query, which is executed many times and requires a network round trip between the computer running the application and SQL Server. Instead, create a single, more complex query using a temporary table. Only one network round trip is necessary, and the query optimizer can better optimize the single query. For more information, see [Procedural Transact-SQL](http://msdn.microsoft.com/en-us/library/ms189626(v=sql.105).aspx) and [Transact-SQL Variables](http://msdn.microsoft.com/en-us/library/ms187953(v=sql.105).aspx).
  + Do not use multiple aliases for a single table in the same query to simulate index intersection. This is no longer necessary because SQL Server automatically considers index intersection and can make use of multiple indexes on the same table in the same query. Consider the sample query:

[Copy](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_89cbd638-7739-48ba-ae12-19f8009584f7');" \o "Copy to clipboard.)

SELECT \* FROM lineitem

WHERE partkey BETWEEN 17000 AND 17100 AND

shipdate BETWEEN '1/1/1994' AND '1/31/1994'

SQL Server can exploit indexes on both the **partkey** and **shipdate** columns, and then perform a hash match between the two subsets to obtain the index intersection.

* + Use query parameterization to allow reuse of cached query execution plans. If a set of queries has the same query hash and query plan hash, you might improve performance by creating one parameterized query. Calling one query with parameters instead of multiple queries with literal values allows reuse of the cached query execution plan. For more information, see [Finding and Tuning Similar Queries by Using Query and Query Plan Hashes](http://msdn.microsoft.com/en-us/library/cc645887(v=sql.105).aspx) and [Execution Plan Caching and Reuse](http://msdn.microsoft.com/en-us/library/ms181055(v=sql.105).aspx).

If you can not modify the application, you can use template plan guides with forced parameterization to achieve a similar result. For more information, see [Specifying Query Parameterization Behavior by Using Plan Guides](http://msdn.microsoft.com/en-us/library/ms191275(v=sql.105).aspx).

* + Make use of query hints only if necessary. Queries using hints executed against earlier versions of SQL Server should be tested without the hints specified. The hints can prevent the query optimizer from choosing a better execution plan. For more information, see [SELECT (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms189499(v=sql.105).aspx).
* Use the query\_plan\_hash to capture, store, and compare the query execution plans for queries over time. For example, after changing the system configuration, you can compare query plan hash values for mission critical queries to their original query plan hash values. Differences in query plan hash values can tell you if the system configuration change resulted in updated query execution plans for important queries. You might also decide to stop execution for a current long-running query if its query plan hash in sys.dm\_exec\_requests differs from its baseline query plan hash, which is known to have good performance. For more information, see [Finding and Tuning Similar Queries by Using Query and Query Plan Hashes](http://msdn.microsoft.com/en-us/library/cc645887(v=sql.105).aspx).
* Make use of the **query governor** configuration option. The **query governor** configuration option can be used to prevent system resources from being consumed by long-running queries. By default, the option is set to allow all queries to execute, no matter how long they take. However, you can set the query governor to limit the maximum number of seconds that all queries are allowed to execute for all connections, or just the queries for a specific connection. Because the query governor is based on estimated query cost, rather than actual elapsed time, it does not have any run-time overhead. It also stops long-running queries before they start, rather than running them until some predefined limit is hit. For more information, see [query governor cost limit Option](http://msdn.microsoft.com/en-us/library/ms190419(v=sql.105).aspx) and [SET QUERY\_GOVERNOR\_COST\_LIMIT (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms176100(v=sql.105).aspx).
* Optimize reuse of query plans from the plan cache. The Database Engine caches query plans for possible reuse. If a query plan is not cached, it can never be reused. Instead, uncached query plans must be compiled each time they are executed, which results in poorer performance. The following Transact-SQL SET statement options prevent cached query plans from being reused. A Transact-SQL batch that contains these SET options turned ON cannot share its query plans with the same batch that was compiled with these SET options turned OFF:

|  |  |
| --- | --- |
| SET ANSI\_NULL\_DFLT\_OFF | SET ANSI\_NULL\_DFLT\_ON |
| SET ANSI\_NULLS | SET ANSI\_PADDING |
| SET ANSI\_WARNINGS | SET ARITHABORT |
| SET CONCAT\_NULL\_YIELDS\_NULL | SET DATEFIRST |
| SET DATEFORMAT | SET FORCEPLAN |
| SET LANGUAGE | SET NO\_BROWSETABLE |
| SET NUMERIC\_ROUNDABORT | SET QUOTED\_IDENTIFIER |
| SET TEXTSIZE |  |

* In addition, the SET ANSI\_DEFAULTS option affects the reuse of cached query plans because it can be used to change the ANSI\_NULLS, ANSI\_NULL\_DFLT\_ON, ANSI\_PADDING, ANSI\_WARNINGS, CURSOR\_CLOSE\_ON\_COMMIT, IMPLICIT\_TRANSACTIONS, and the QUOTED\_IDENTIFIER SET options. Note that most of the SET options that can be changed with SET ANSI\_DEFAULTS are listed as SET options that can affect the reuse of query plans.
* You can change some of these SET options with the following methods:
  + Use the **sp\_configure** stored procedure for server-wide changes. For more information, see [sp\_configure (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms188787(v=sql.105).aspx).
  + Use the SET clause of the ALTER DATABASE statement. For more information, see [ALTER DATABASE (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms174269(v=sql.105).aspx)
  + Change OLE DB and ODBC connection settings. For more information, see [Client Network Configuration](http://msdn.microsoft.com/en-us/library/ms190611(v=sql.105).aspx).

|  |
| --- |
| **NoteNote** |
| To avoid recompilations of query plans caused by SET options, establish SET options at connection time, and make sure that they do not change for the duration of the connection. Some SET options must be set to specific values to use indexed views or indexes on computed columns. For more information, see [SET Options That Affect Results](http://msdn.microsoft.com/en-us/library/ms175088(v=sql.105).aspx). |

**Advanced Query Tuning Concepts**

**SQL Server 2008 R2**

[Other Versions](javascript:;)



* [SQL Server 2008](http://msdn.microsoft.com/en-us/library/ms191426(d=printer,v=sql.100).aspx)
* [SQL Server 2005](http://msdn.microsoft.com/en-us/library/ms191426(d=printer,v=sql.90).aspx)

Microsoft SQL Server performs sort, intersect, union, and difference operations using in-memory sorting and hash join technology. Using this type of query plan, SQL Server supports vertical table partitioning, sometimes called columnar storage.

SQL Server employs three types of join operations:

* Nested loops joins
* Merge joins
* Hash joins

If one join input is small (fewer than 10 rows) and the other join input is fairly large and indexed on its join columns, an index nested loops join is the fastest join operation because they require the least I/O and the fewest comparisons. For more information about nested loops, see [Understanding Nested Loops Joins](http://msdn.microsoft.com/en-us/library/ms191318(v=sql.105).aspx).

If the two join inputs are not small but are sorted on their join column (for example, if they were obtained by scanning sorted indexes), a merge join is the fastest join operation. If both join inputs are large and the two inputs are of similar sizes, a merge join with prior sorting and a hash join offer similar performance. However, hash join operations are often much faster if the two input sizes differ significantly from each other. For more information, see [Understanding Merge Joins](http://msdn.microsoft.com/en-us/library/ms190967(v=sql.105).aspx).

Hash joins can efficiently process large, unsorted, nonindexed inputs. They are useful for intermediate results in complex queries because:

* Intermediate results are not indexed (unless explicitly saved to disk and then indexed) and often are not suitably sorted for the next operation in the query plan.
* Query optimizers estimate only intermediate result sizes. Because estimates can be very inaccurate for complex queries, algorithms to process intermediate results not only must be efficient, but also must degrade gracefully if an intermediate result turns out to be much larger than anticipated.

The hash join allows reductions in the use of denormalization. Denormalization is typically used to achieve better performance by reducing join operations, in spite of the dangers of redundancy, such as inconsistent updates. Hash joins reduce the need to denormalize. Hash joins allow vertical partitioning (representing groups of columns from a single table in separate files or indexes) to become a viable option for physical database design. For more information, see [Understanding Hash Joins](http://msdn.microsoft.com/en-us/library/ms189313(v=sql.105).aspx).