<http://technet.microsoft.com/en-us/library/cc917694.aspx>

# Forcing Query Plans

10 out of 17 rated this helpful - [Rate this topic](http://technet.microsoft.com/en-us/library/cc917694.aspx#feedback)

Published: November 1, 2005

**Author:** Burzin A. Patel

**Technical Reviewers:** Eric Hanson and Peter Scharlock

**Applies To:** Microsoft® SQL Server™ 2005

**Summary:** This paper explains the USE PLAN query hint and the Plan Guides feature that are introduced in Microsoft® SQL Server™ 2005. This paper also demonstrates how they can be used individually, or together, to force the selection of specific query execution plans for a variety of scenarios.

##### On This Page

[Introduction](http://technet.microsoft.com/en-us/library/cc917694.aspx#XSLTsection126121120120)   
[Target Audience](http://technet.microsoft.com/en-us/library/cc917694.aspx#XSLTsection127121120120)   
[SQL Server Query Optimization](http://technet.microsoft.com/en-us/library/cc917694.aspx#XSLTsection128121120120)   
[USE PLAN Query Hint](http://technet.microsoft.com/en-us/library/cc917694.aspx#XSLTsection129121120120)   
[Plan Guides](http://technet.microsoft.com/en-us/library/cc917694.aspx#XSLTsection130121120120)   
[Best Practices](http://technet.microsoft.com/en-us/library/cc917694.aspx#XSLTsection131121120120)   
[Supported Editions](http://technet.microsoft.com/en-us/library/cc917694.aspx#XSLTsection132121120120)   
[Conclusion](http://technet.microsoft.com/en-us/library/cc917694.aspx#XSLTsection133121120120)   
[Appendix](http://technet.microsoft.com/en-us/library/cc917694.aspx#XSLTsection134121120120)

### Introduction

Over the past few years, Microsoft® SQL Server™ has increased its presence in the industry and has reduced its TCO. This reduced TCO is a direct result, primarily, of the numerous self-tuning mechanisms built into Microsoft® SQL Server™. These mechanisms automatically perform tasks that would otherwise have to be performed by experienced database administrators. One such mechanism is the cost-based optimizer (CBO) that is used to dynamically generate query execution plans. The CBO probes several system-wide resource states and employs many complex, heuristical algorithms to generate the best possible plan for a given query and the underlying table and index structures. This mechanism works well for the vast majority of user queries, but there are times when experienced users need to force a particular query plan, based on some prior knowledge or insights into future uses.

Forcing query execution plans, to a limited extent, was possible in earlier versions of SQL Server by using various query hints, join hints, and index hints. However, the process was often based on trial and error and was also tedious. SQL Server 2005 introduces a new query hint, named USE PLAN, that guides the optimizer in creating a query plan based upon a specified XML query plan.

The Plan Guides feature, also new in SQL Server 2005, provides a method for injecting query hints into SQL statements in batches, stored procedures (SP), and so forth. However, it does not require any modification to the query itself. The mechanism uses a look-up mapping table and is very useful when the query for which the plan has to be influenced or forced originates in a non-modifiable application.

This paper explains the USE PLAN query hint and Plan Guides feature together with typical usage scenarios, restrictions, and recommended best practices. This paper presents only an overview of these features and is not intended to serve as a comprehensive reference document. For detailed information, see SQL Server 2005 Books Online. The Appendix section also contains a real-world, end-to-end scenario in which a query performance problem in a deployed application is resolved by using these features.

[Top Of Page](http://technet.microsoft.com/en-us/library/cc917694.aspx#mainSection)

### Target Audience

This paper is primarily intended for the following audience:

* Database administrators (DBAs) who want to tune query performance
* Application testers who want to easily test out different query plan options

[Top Of Page](http://technet.microsoft.com/en-us/library/cc917694.aspx#mainSection)

### SQL Server Query Optimization

SQL Server uses a complex, cost-based query optimization mechanism that considers numerous factors before formulating a query execution plan. After being compiled, query plans are cached by the SQL Server engine to avoid having to repeat the same task when the identical query is re-executed. Query plans are optimized for the specific data present in the underlying tables. Because of this, the SQL Server engine constantly monitors changes to the underlying tables and triggers a recompile of the query plan when it estimates that the data has changed significantly enough to justify a re-optimization.

This mechanism works well for the majority of queries. However, there are instances when a highly skewed data-set, stale statistics on large tables, an optimizer time-out when finding the best possible plan, and so forth, cause the optimizer to generate a less than optimal query execution plan. In these cases, the USE PLAN query hint and Plan Guides feature can be used very effectively to manually influence the optimizer behavior.

[Top Of Page](http://technet.microsoft.com/en-us/library/cc917694.aspx#mainSection)

### USE PLAN Query Hint

Microsoft® SQL Server™ 2005 introduces a new query hint, called USE PLAN, that can be used to guide the query optimizer into selecting a specified XML query plan. This powerful option provides users with full control over influencing the execution of a query.

Earlier versions of SQL Server supported several query hints, such as FORCE ORDER, LOOP JOIN, and KEEP PLAN, that could be used to help optimize a query in a particular way. However, none were powerful enough to influence the optimizer to consistently choose a particular query plan, especially when the referenced table row counts, statistics, indexes, and other attributes of the environment changed. The USE PLAN query hint fills this gap and provides the user with complete control over query plan stability.

The USE PLAN query hint is specified in an OPTION clause together with an XML showplan. The following is an example of the USE PLAN hint specified to influence the join type of a simple query that consists of a join between two tables.

Following is the original query:

SELECT count(\*) AS Total

FROM Sales.SalesOrderHeader h, Sales.SalesOrderDetail d

WHERE h.SalesOrderID = d.SalesOrderID

GO

Following is the same query with the USE PLAN query hint specified:

SELECT count(\*) AS Total

FROM Sales.SalesOrderHeader h, Sales.SalesOrderDetail d

WHERE h.SalesOrderID = d.SalesOrderID

OPTION (USE PLAN N'

<ShowPlanXML xmlns=

"http://schemas.microsoft.com/sqlserver/2004/07/showplan" Version="0.5"

Build="9.00.1187.07">

  <BatchSequence>

    <Batch>

      <Statements>

       ...

      </Statements>

    </Batch>

  </BatchSequence>

</ShowPlanXML>

')

GO

In this example, the USE PLAN hint and <xml showplan> are specified through the OPTION clause following the original SELECT query. For readability purposes, most of the 100-plus lines of the XML query plan have been replaced by the ellipses. Although not a significant example, the true power of this feature lies in being able to force the query plan for more complex queries that involve multiple table joins with multiple predicates and aggregate clauses.

#### Common Use Scenarios for USE PLAN Query Hint

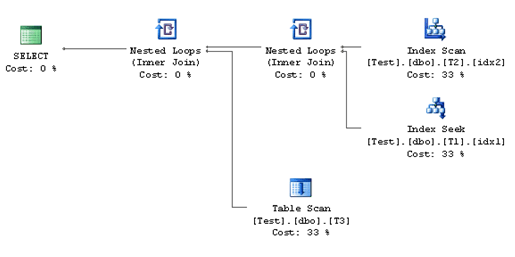
SQL Server generates optimal query plans for most queries. However, there are times, especially with more complex applications, when certain queries benefit from user intervention and some level of hand tuning. Following are typical examples of this:

* A complex query involving multiple tables where the compiled or recompiled query plan is occasionally not optimal. This behavior could be a result of out-of-date or missing statistics in any of the underlying tables. It could also be the result of complex constructs in the query that cause the optimizer to inaccurately estimate the size of the intermediate query results.
* Cases where a query plan chosen by the query optimizer in an earlier product version is preferred over the one chosen after the upgrade. The upgrade can be a Hotfix (QFE), a service pack, or a full version upgrade.

#### USE PLAN Query Hint Restrictions

USE PLAN query hints have the following restrictions:

* Only query plans for SELECT and SELECT INTO statements can be forced. These statements can reference tables or views. However, these tables or views may be indexed anyway and may be partitioned. Query plans for UPDATE, DELETE, or INSERT statements cannot be forced
* You can only force plans with USE PLAN that can be produced by the optimizer's normal search strategy. These plans are typically, though not always, left-deep binary trees where one child of each join is always a leaf (table scan, index scan, or index seek). Figure 1 illustrates an example of a plan that you can force.

[](http://technet.microsoft.com/en-us/library/Cc917694.foqupl01_big(l=en-us).gif)

**Figure 1: Example Query Execution Plan**

You cannot force arbitrary bushy trees that are join trees where at least one join node has join nodes for both children. You can produce plans with a bushy structure by using a parenthesized set of JOIN operations in the FROM clause of a query and the OPTION(FORCE ORDER) hint. However, trying to force such a plan will fail with an error message. Because of these limitations and the complexity of creating a valid plan, the most common and reliable way to force a plan is to capture a plan that is produced by the optimizer and then force it on the same query.

* Rewriting a query by using JOIN hints, query (OPTION clause) hints, and index hints is one way to get an improved plan for it to be produced automatically by SQL Server. You can then successfully force this plan on the original query in many cases. For more information about how to apply this technique, see the SQL Server 2005 Books Online topic, “Plan Forcing Scenario: Create a Plan Guide to Force a Plan Obtained from a Rewritten Query.”

Occasionally, the use of a JOIN hint, FORCE ORDER hint, or SET FORCEPLAN ON may cause the creation of a plan that cannot be matched when specified in a USE PLAN hint. In general, if you create a query with a FORCE ORDER hint and get the XML showplan for the query, and then try to force the same query by using USE PLAN with the FORCE ORDER hint removed, SQL Server may not find a plan for the query. This is because the FORCE ORDER hint overrides the typical search strategy of the optimizer. Similarly, it may not be possible to force a plan for a query if the plan was produced with the SET FORCEPLAN option enabled.

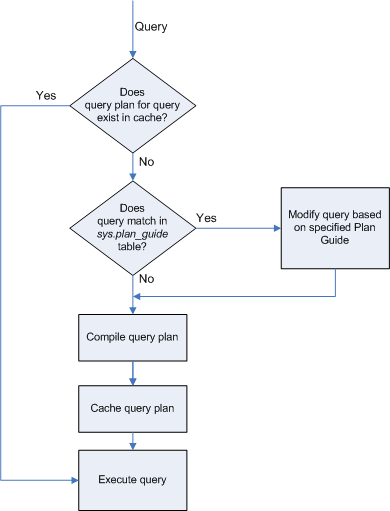
* Query plans for queries that contain an XML plan larger than 8 KB in a USE PLAN hint are not cacheable. To work around this, a plan guide should be used to specify the query hint.

[Top Of Page](http://technet.microsoft.com/en-us/library/cc917694.aspx#mainSection)

### Plan Guides

There are times when application performance tuning requires that the query plans for a particular query, or a small set of queries, be influenced by using one or more query hints. Although this can be easy to do when users have access to the application code, they often do not and the particular queries to be modified are embedded deep within a third-party application. This makes alteration virtually impossible.

The Plan Guides feature provides an ideal solution for such scenarios and offers users a mechanism for injecting hints into the original query. The mechanism uses an internal look-up system table, based on information in the sys.plan\_guides catalog view, to map the original query to a substitute query or template. The flow chart in Figure 2 illustrates the flow of operations involved in the query mapping process.



**Figure 2: Plan Guide Matching Flow Chart**

Every SQL query statement or batch is first compared against the optimizer’s cached plan store to check for a match. If one exists, the cached query plan is used to execute the query. If not, the query or batch is checked against the set of existing plan guides in the current database for a match. If an active plan guide exists for the statement and its context (batch, SP, or other module), the original matching statement is substituted with the one from the plan guide. After this is done, the query plan is compiled and cached and the query executed.

The Plan Guides feature can be used to specify any of the following query hints individually, or together with others, when applicable:

{HASH | ORDER} GROUP

{CONCAT | HASH | MERGE} UNION

{LOOP | MERGE | HASH} JOIN

FAST number\_rows

FORCE ORDER

MAXDOP number\_of\_processors

OPTIMIZE FOR ( @variable\_name = literal\_constant ) [ ,...n ]

RECOMPILE

ROBUST PLAN

KEEP PLAN

KEEPFIXED PLAN

EXPAND VIEWS

MAXRECURSION number

USE PLAN <xmlplan>

The Plan Guides feature essentially consists of two stored procedures to create, drop, enable, and disable plan guides, and also a new metadata view that describes the stored plan guides. The following sections describe the procedures involved to create plan guides and also the various scenarios in which they are useful.

#### Creating and Administering Plan Guides

Plan guides are created and administered by using the following two system stored procedures:

* sp\_create\_plan\_guide
* sp\_control\_plan\_guide

##### sp\_create\_plan\_guide

The sp\_create\_plan\_guide stored procedure is used to create a plan guide. Following is the format for this command:

sp\_create\_plan\_guide [ @name = ] N'plan\_guide\_name'

  , [ @stmt = ] N'statement\_text'

  , [ @type = ] N' { OBJECT | SQL | TEMPLATE }'

  , [ @module\_or\_batch = ]

      {   N'[ schema\_name.]object\_name'

        | N'batch\_text'

        | NULL

      }

  , [ @params = ] { N'@parameter\_name data\_type [,...n ]' | NULL }

  , [ @hints = ] { N'OPTION ( query\_hint [,...n ] )' | NULL }

where:

**@name** is of type navrchar(128) and specifies the name of the plan guide. Plan guide names are database-wide and scoped to the current database.

**@stmt** is of type navrchar(max) and contains a Transact-SQL statement or batch.

**@type** is of type nvarchar(60) and specifies the type of entity against which this plan guide will be matched. Following are the valid values for @type:

* OBJECT: Used to indicate that the statement\_text appears in the context of a Transact-SQL stored procedure, scalar function, multi-statement table-valued function, or Transact-SQL DML trigger.
* SQL: Used to indicate that the given statement\_text appears in the context of a stand-alone statement or batch that can be submitted to SQL Server through any mechanism.
* TEMPLATE: Used to indicate that the plan guide applies to any query that parameterizes to the form indicated in statement\_text.

**@module\_or\_batch** is of type nvarchar(max) and specifies the module name or batch text. If @module\_or\_batch is NULL (default value) and @type = 'SQL', @module\_or\_batch is set to @stmt.

**@params** is of type nvarchar(max) and represents a string that contains the definitions of all parameters for a T-SQL batch to be matched by this plan guide. For @type='SQL', the values of @module\_or\_batch and @params, taken together, uniquely identify an incoming T-SQL batch. The parameters provided in @params must exactly match those provided with the SQL batch submitted through sp\_executesql in a character-for-character comparison. You should provide a value for @params only when you define a plan guide for parameterized dynamic SQL. The value of @type must be 'SQL' or ‘TEMPLATE’ if you specify a non-null value for @params. Each parameter definition consists of a parameter name and a data type. The n is a placeholder and indicates additional parameter definitions. If the Transact-SQL statement or batch in @module\_or\_batch does not contain parameters, @params must be NULL. This is the default.

**@hints** is of type nvarchar(max) and specifies the OPTION clause text to attach to an incoming query that matches @stmt. It must be syntactically the same as an OPTION clause on a SELECT statement, or be a NULL value, which represents an empty OPTION clause. It can contain any legal sequence of query hints.

All arguments passed to sp\_create\_plan\_guide must be either constants of the designated type, or variables that can be implicitly converted to the designated type, and follow standard calling conventions for stored procedures. Constant strings that are passed as arguments must be Unicode strings and be specified by using N'...' notation. All arguments must either be specified as ‘@name=value” or just “value”, because mixing the two may result in an error when sp\_create\_plan\_guide is executed.

The following is an example of a plan guide created for a simple SQL statement:

sp\_create\_plan\_guide

@name = N'PlanGuide1',

@stmt = N'SELECT COUNT(\*) AS Total

FROM Sales.SalesOrderHeader h, Sales.SalesOrderDetail d

WHERE h.SalesOrderID = d.SalesOrderID and h.OrderDate

BETWEEN ''1/1/2000'' AND ''1/1/2005''

',

@type = N'SQL',

@module\_or\_batch = NULL,

@params = NULL,

@hints = N'OPTION (MERGE JOIN)'

GO

When creating plan guides, care should be taken to specify the query in the @stmt parameter and the parameter names and values in the @params parameter exactly as they are received from the application. This is best achieved by capturing the batch or statement text from SQL Profiler as described in SQL Server Books Online. Single-quoted literal values, such as ‘1/1/2000’, should be delimited with single quotes escaped by additional single quotes, as shown in the example.

##### sp\_control\_plan\_guide

The sp\_control\_plan\_guide stored procedure is used to enable, disable, or drop a plan guide.Following is the format for this command:

sp\_control\_plan\_guide [ @operation = ] N'<control\_option>' [ , [ @name = ]

N'plan\_guide\_name' ]

The <control option> can be:

**DROP** – Used to drop the plan guide specified by plan\_guide\_name.

**DROP ALL** – Used to drop all plan guides in the current database.

**DISABLE** - Used to disable the plan guide specified by plan\_guide\_name.

**DISABLE ALL** – Used to disable all plan guides in the current database.

**ENABLE** - Used to enable the plan guide specified by plan\_guide\_name.

**ENABLE ALL** – Used to enable all plan guides in the current database.

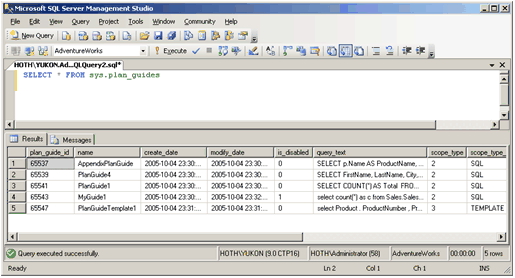
For example:

sp\_control\_plan\_guide N'DROP', N'PlanGuide1'

To execute sp\_control\_plan\_guide on a plan guide of type OBJECT (created by specifying @type = 'OBJECT' ), you must, at a minimum, have ALTER permission on the object that is referenced by the plan guide. For all other plan guides, you must have ALTER DATABASE permissions. Trying to drop or alter a function, stored procedure, or DML trigger referenced by a plan guide, whether enabled or disabled, results in an error.

#### sys.plan\_guides Catalog View

All plan guides are stored in the sys.plan\_guides database system catalog view. A sample listing of all plan guides in a test database is shown in Figure 3.

[](http://technet.microsoft.com/en-us/library/Cc917694.foqupl03_big(l=en-us).gif)

**Figure 3: List of Plan Guides in a Test Database**

This system view can be queried to access details about a plan guide such as the date it was created, the date modified, whether it is enabled or disabled, and the query text.

#### Common Usage Scenarios for Plan Guides

This section presents some common use scenarios in which plan guides can be used to influence the query execution plan generated by the optimizer.

* **Parameterized Queries**

Parameterizing T-SQL queries are a well-known, database programming, best practice. It allows query plan reuse and eliminates the need of recompilation for multiple invocations of the same query that simply has different parameter values. However, there are times when parameterized queries might perform poorly, because they use cached query plans that are optimized for some non-representative set of parameter values.

In these cases, the OPTIMIZE FOR or RECOMPILE query hints can generally be used to address this problem. OPTIMIZE FOR instructs SQL Server to optimize the query plan for the particular parameter value specified in the OPTIMIZE FOR clause. This option is very useful in cases where there is just one optimal value for a parameter and it is easily identifiable and known to the DBAs. RECOMPILE instructs the server to discard a query plan after execution and forces the query optimizer to recompile a new query plan for successive executions of the same query1. The RECOMPILE hint is useful when the query is executed infrequently with vastly differing parameter values.

Following is an example of a plan guide created for a stored procedure to force the optimization of the resulting query plan for value @Country=’US’:

sp\_create\_plan\_guide N'PlanGuide2',

N'SELECT \*

FROM Sales.SalesOrderHeader h, Sales.Customer c, Sales.SalesTerritory t

WHERE h.CustomerID = c.CustomerID AND c.TerritoryID = t.TerritoryID

AND CountryRegionCode = @Country',

N'OBJECT',

N'Sales.GetSalesOrderByCountry',

NULL,

N'OPTION (OPTIMIZE FOR(@Country = N''US''))'

* **Forcing Query Plans**

Another common hint for use with plan guides is the USE PLAN query hint previously discussed. This query hint applies when you know of a better-performing query plan that can be substituted for the one chosen by the optimizer for a particular query. USE PLAN forces SQL Server to use a particular query plan specified explicitly in the hint syntax when executing the query if it is one of the query plans that the optimizer would consider in its selection process. What this implies is that any arbitrarily formulated or manually altered query plan may not be forced by the USE PLAN query hint.

* **Eliminating or Replacing Existing Query Hints**

Although it is not a common or recommended practice, there are times when application queries directly specify query hints. The following example query uses one such query hint to force a nested loop join:

SELECT t1.a, t2.b FROM Tab1 t1, Tab1 t2 WHERE t1.a = t2.a

OPTION(LOOP JOIN)

GO

In cases where these query hints cause problems and need to be eliminated, plan guides can be created with a NULL specified for the @hint parameter. The following plan guide effectively removes the loop join hint embedded in the previous application T-SQL:

sp\_create\_plan\_guide

@name = N'PlanGuide1',

@stmt = N'SELECT t1.a, t2.b FROM Tab1 t1, Tab1 t2 WHERE t1.a = t2.a

OPTION(LOOP JOIN)

',

@type = N'SQL',

@module\_or\_batch = NULL,

@params = NULL,

@hints = NULL

GO

To substitute the application-provided, loop join hint with another hint, the required join type can be specified in the @hint option. For example, to force a merge join, the plan guide can be created with @hint = N’OPTION (MERGE JOIN)’.

* **Forcing A Non-parallel Execution Plan**

When operating on multi-processor systems, SQL Server determines at run time whether a query should be executed through a parallel or non-parallel query plan, based on predetermined criteria. This is the default behavior for SQL Server and it works well for speeding up query execution for complex queries through parallelization. Occasionally, a DBA would like to consistently force a particular query to execute with a non-parallel query plan. This can be done by creating a plan guide for the particular query using the MAXDOP=1 query hint, as shown in the following:

sp\_create\_plan\_guide

@name = N'PlanGuide3',

@stmt = N'SELECT TOP 1 \* FROM Sales.SalesOrderHeader h ORDER BY

OrderDate DESC

',

@type = N'SQL',

@module\_or\_batch = NULL,

@params = NULL,

@hints = N'OPTION (MAXDOP 1)'

* **Forcing Join Types**

Plan guides can also be used to force the join type used in a particular query. This is shown in the following plan guide example that forces a merge join between the two tables:

sp\_create\_plan\_guide

@name = N'PlanGuide4',

@stmt = N'SELECT FirstName, LastName, City, VacationHours

FROM HumanResources.Employee e, Person.Contact c, Person.Address a,

HumanResources.EmployeeAddress ea

WHERE e.EmployeeID = ea.EmployeeID

AND ea.AddressID = a.AddressID

AND e.ContactID = c.ContactID

',

@type = N'SQL',

@module\_or\_batch = NULL,

@params = NULL,

@hints = N'OPTION (HASH JOIN)'

GO

This will force the optimizer to use a hash match type join for all the join operators2.

* **Controlling Parameterization**

SQL Server 2005 introduces two new options for forcing parameterization of queries: FORCED PARAMETERIZATION and SIMPLE PARAMETERIZATION. The FORCED PARAMETERIZATION option forces parameterization of all queries. (There are some exceptions and these are listed in SQL Server 2005 Books Online.) On the other hand, the SIMPLE PARAMTERIZATION option lets the SQL Server query optimizer decide whether to parameterize queries. Both of these are database-wide options and can be enabled by using the ALTER DATABASE command.

In some cases, it may be beneficial to specify either of these hints for a specific query to alter its behavior. For example, a database may have the default parameterization set to SIMPLE, and a DBA may only want the following query to be force parameterized:

SELECT Product.ProductNumber, Product.ReorderPoint,

SUM(ProductInventory.Quantity) AS Quantity

FROM Production.Product, Production.ProductInventory

WHERE Product.ProductID = ProductInventory.ProductID

AND Product.ProductNumber = N'BE-2908'

AND Product.ReorderPoint > 0

GROUP BY Product.ProductID, Product.ProductNumber, Product.ReorderPoint

This can be achieved by creating a plan guide template for the query. A template plan guide can be created by first extracting the template text and parameter definitions using the sp\_get\_query\_template stored procedure, and then using these to create the plan guide template by using the sp\_create\_plan\_guide stored procedure. For example:

DECLARE @stmt nvarchar(max)

DECLARE @params nvarchar(max)

EXEC sp\_get\_query\_template N'SELECT Product.ProductNumber,

Product.ReorderPoint, SUM(ProductInventory.Quantity) AS Quantity

FROM Production.Product, Production.ProductInventory

WHERE Product.ProductID = ProductInventory.ProductID

AND Product.ProductNumber = N''BE-2908''

AND Product.ReorderPoint > 0

GROUP BY Product.ProductID, Product.ProductNumber,

Product.ReorderPoint',

@stmt OUTPUT,

@params OUTPUT

EXEC sp\_create\_plan\_guide N'PlanGuideTemplate1',

@stmt,

N'TEMPLATE',

NULL,

@params,

N'OPTION(PARAMETERIZATION FORCED)'

After being created, this plan guide template will match all query executions with this format. It will do this regardless of the literal values compared with ProductNumber and ReorderPoint, and thereby facilitate query plan reuse and a reduced number of query compilations. The plan guide template matching can be verified by either observing that the two variables were in fact parameterized in the query execution plan, or by searching for ‘TemplatePlanGuideDB’ and ‘TemplatePlanGuideName’ in the showplan\_xml output listing.

#### Plan Guide Restrictions

This section lists some of the restrictions enforced with the use of plan guides.

* **Modifying and Deleting Underlying Objects**

After a plan guide has been created on a function, stored procedure, or DML trigger that is referenced by a plan guide, whether enabled or disabled, an error will occur if you try to modify or delete that object.

* **Database Scope**

All plan guides have a database-wide scope. This implies that the plan guide name has to be unique within a database. However, different databases can have a plan guide with the same name.

* **Supported DML**

Plan guides may be applied to any statement that can accept query hints, as an OPTION clause. This includes SELECT, UPDATE, DELETE, and INSERT...SELECT statements.

* **Query Text Matching**

In order for plan guides that specify @type = 'SQL' or 'TEMPLATE' to match a query successfully, the values for batch\_text and @parameter\_name data\_type [,...n ] must be provided in exactly the same format as their counterparts submitted by the application. Specifically, it has to be character for character, including comments and white spaces.

* **Encrypted Objects**

Plan guides cannot be created against stored procedures, functions, or DML triggers that specify the WITH ENCRYPTION clause.

* **DDL Triggers**

Plan guides cannot be specified for DDL triggers. Only DML triggers are supported.

[Top Of Page](http://technet.microsoft.com/en-us/library/cc917694.aspx#mainSection)

### Best Practices

Following are some of the recommended best practices for using the USE PLAN query hint and the Plan Guides feature.

* The USE PLAN query hint and plan guides should only be used when other standard query tuning options, such as index tuning and ensuring current statistics, have been extensively tried and have failed to produce the necessary results.
* Only experienced database administrators who understand all the implications and long-term ramifications of forcing the query plans should use these options. After a query plan is forced by using either the USE PLAN query hint or a plan guide, it gets locked down and prevents the optimizer from adapting to changing data distributions, new indexes, improved query execution algorithms in successive SQL Server releases, service packs (SPs), and HotFixes (engineering fixes).
* You should only try to force a small fraction of the workload. If you are forcing more than a few dozen queries, check if there are other issues with the configuration that could be limiting performance. These can include insufficient system resources, incorrect database configuration settings, missing indexes, poorly written queries, and other factors.
* It is not advisable to code by hand or modify the XML showplan that is specified in the USE PLAN query hint. The XML showplan is a lengthy and complex listing. Any change that would prevent it from identically matching one of the query optimizer generated plans would result in the USE PLAN hint being ignored. You should capture and use a plan produced by SQL Server in a USE PLAN hint for the most reliable plan forcing results.
* The USE PLAN query hint should not be directly embedded into the application code, because this would make the maintenance of the application across query plan and SQL Server version changes difficult to manage. Embedding USE PLAN directly into the query also generally makes the plan for the query not cacheable. The USE PLAN hint is designed primarily for ad-hoc performance tuning and test purposes, and for use with the Plan Guides feature
* The plan guides created for an application should be well documented and regularly backed up, because they constitute an integral part of the application’s performance tuning. You should also retain the scripts that you used to create plan guides and treat them as you would other source code for an application.
* After being created, a plan guide should be tested to make sure that it is being applied to the intended queries. You can easily do this by verifying that the Showplan XML listing produced by the Profiler Showplan XML event, or by using SET SHOWPLAN\_XML ON, does contain the PlanGuideDB and PlanGuideName attributes for the plan guide that you expect should match the query.

[Top Of Page](http://technet.microsoft.com/en-us/library/cc917694.aspx#mainSection)

### Supported Editions

The following table illustrates the USE PLAN and plan guide functionality supported in the different editions of SQL Server 2005.

**Table 1**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Function** | **SQL Server 2005 (Developer Edition)** | **SQL Server 2005 (Express Edition)** | **SQL Server 2005 (Workgroup Edition)** | **SQL Server 2005 (Standard Edition)** | **SQL Server 2005  (Enterprise Edition)** |
| USE PLAN query hint | Y | Y | Y | Y | Y |
| Plan guides use | Y | N | N | Y | Y |
| Plan guides create/enable/disable | Y | N | N | Y | Y |
| Plan guides drop | Y | Y | Y | Y | Y |

[Top Of Page](http://technet.microsoft.com/en-us/library/cc917694.aspx#mainSection)

### Conclusion

With the introduction of the USE PLAN query hint and the Plan Guides features, SQL Server 2005 significantly enhances the ability of users to control the behavior of the optimizer. You can use these features independently or together to fine-tune the query execution plans by using a wide variety of query hints, and maintain query plan stability.

**For More Information**

<http://www.microsoft.com/technet/prodtechnol/sql/default.mspx>

[Top Of Page](http://technet.microsoft.com/en-us/library/cc917694.aspx#mainSection)

### Appendix

This appendix provides an example scenario of performance tuning in the application, AdventureWorksApplication. This application operates against the SQL Server 2005 AdventureWorks sample database. The following sections present the problem scenario, the analysis performed, and the resolutions. The queries presented in this appendix are for example purposes only and do not truly exhibit any performance issues.

**Scenario**

Users of AdventureWorksApplication have indicated that their product information catalog search requests, at times, took an excessive amount of time to return the results. Yet at other times, the response times were very fast.

**Analysis**

Because the intermittent nature of the problem was not something the DBAs had seen before, it made the analysis challenging. To do a thorough investigation of the problem, they employed performance tuning best practices by obtaining a holistic view of the system performance. To do this, they monitored the database server by using Windows perfmon and SQL Profiler. For Windows perfmon, they logged all the SQL Server objects and key system objects, such as network, memory, processor, and disks, to a perfmon log file. They also set up SQL Profiler with a Duration column filter set to capture all queries that took longer than 600 milliseconds to execute. Next, they monitored the database server for 24 hours and made sure that the application users encountered the problem at least a couple of times during that period.

After analyzing the captured perfmon log, they found that there were no serious system bottlenecks and the SQL Server counter values were stable and well within standard limits. Confident that there was no major resource bottleneck, the DBAs turned their attention to the SQL Profiler data.

During the 24-hour period, the SQL Profiler had captured approximately 280 instances of queries that took more than 600 milliseconds to execute. After some comprehensive comparison and analysis, the DBAs found that all 280 queries were different invocations of just three distinct queries.

Of the three queries, they found that one was missing an optimal index and, thus, was performing a full index scan. The second query involved a join between two large tables and resulted in a poor query execution plan by virtue of the optimizer using sub-optimal indexes, even though the optimal indexes were available. Through further analysis, they determined that the Automatic update statistics option on one of the large tables had been turned off, most likely accidentally during an experiment they had conducted in the past.

The third query, submitted through an API server cursor3 and captured in the SQL Profiler trace, as shown in the following, appeared to have all the correct indexes to operate from and also had relatively up-to-date statistics on all the tables. Moreover, it completed execution in less than 50 milliseconds when executed directly through SQL Server Management Studio.

declare @P1 int

set @P1=-1

declare @P2 int

set @P2=0

declare @P3 int

set @P3=28688

declare @P4 int

set @P4=8193

declare @P5 int

set @P5=2560

exec sp\_cursorprepexec @P1 output, @P2 output, N'@P1 char',

N'SELECT p.Name AS ProductName, v.Name AS VendorName, p.ProductLine

FROM Production.Product p, Purchasing.ProductVendor pv,

Purchasing.Vendor v

WHERE p.ProductID = pv.ProductID

AND pv.VendorID = v.VendorID

AND p.ProductLine = @P1

ORDER BY p.Name, v.Name', @P3 output, @P4 output,

@P5 output, 'Z'

This was initially baffling. After some experimentation and analysis, however, the DBAs determined that this query resulted in an efficient execution plan that operated in less than 50 milliseconds when the query plan was compiled with the common values of ‘M’, ‘R’, or ‘T’ supplied for the @P1 parameter. However, it was a drastically different execution plan when supplied with any other value. They explained this behavior as being related to the specific value that the optimizer optimized the query for.

In summary, they determined that supplying anything other than the common values when a compile or recompile was performed resulted in the optimizer compiling and caching the query plan for that particular value. Yet, when that query plan was reused for subsequent executions of the same query for the common values (‘M’, ‘R’, or ‘T’), it resulted in sub-optimal performance. This sub-optimal performance problem existed until the query was recompiled. At that point, based on the @P1 parameter value supplied, the query might or might not have a performance problem.

**Resolution**

The DBAs resolved the first query problem by creating the missing index on the table and validating that the index was actually selected by the optimizer.

For the second query, they enabled the Automatic update statistics option that had accidentally been turned off and then manually updated the statistics of all indexes on that table one time to ensure that the statistics were current. This resulted in the optimizer selecting the correct index and also resulted in satisfactory query performance.

From the analysis, the DBAs knew that to guarantee a good execution plan for the third query they had to ensure that the query was always compiled with any one of the common values (‘M’, ‘R’, or ‘T’) for the @P1 parameter. To do this, they knew they could use the OPTIMIZE FOR query hint. However, because the query originated in the AdventureWorksApplication, directly modifying the query to add this hint was not possible.

To work around this, they resorted to creating a plan guide with the OPTIMIZE FOR query hint. The following procedures explain the process that they used to do this.

**Collect the query plan**

1. Start a SQL Server Profiler trace.
2. Select the RPC:Starting event from the Stored Procedures group.
3. Cause AdventureWorksApplication to execute the query.
4. Locate the RPC:Starting event generated for the query.
5. Collect the query by right-clicking the RPC:Starting event that contains the query and then select the Extract Event Data option.
6. Save the event data in a file, CursorQuery.sql, on your desktop.
7. Open CursorQuery.sql in an editor window in SQL Server Management Studio.
8. Replace each single quote (') in the plan with two single quotes ('') by selecting Find and Replace and then selecting Quick Replace.
9. Copy the query part of the sp\_cursorprepexec call into the system buffer.
10. Save CursorQuery.sql.

**Create the plan guide**

* Create a plan guide by executing the following sp\_create\_plan\_guide statement with the specified OPTIME FOR query hint as shown.
* sp\_create\_plan\_guide
* @name = N'AppendixPlanGuide',
* @stmt = N'SELECT p.Name AS ProductName, v.Name AS VendorName,
* p.ProductLine
* FROM Production.Product p, Purchasing.ProductVendor pv,
* Purchasing.Vendor v
* WHERE p.ProductID = pv.ProductID
* AND pv.VendorID = v.VendorID
* AND p.ProductLine = @P1
* ORDER BY p.Name, v.Name',
* @type = N'SQL',
* @module\_or\_batch = NULL,
* @params= N'@P1 char',
* @hint = N'OPTION (OPTIMIZE FOR(@P1 = N''M''))'

The text supplied for the @stmt parameter is directly pasted from the system buffer loaded in step 9 in the **Collect the query plan** procedure.

**Execute the query and verify that the plan guide is applied**

1. Cause AdventureWorksApplication to execute the query again and gather its XML execution plan by selecting the Showplan XML Statistics Profile event under the Performance group in SQL Server Profiler.
2. Collect the query plan by right-clicking the Showplan XML Statistics Profile event that corresponds to the query and then select the Extract Event Data option.
3. Save the event data in a file, Showplan.SQLPlan, on your desktop.
4. Change the name of Showplan.SQLPlan file to Showplan.xml.
5. Open the Showplan.xml file in any viewer, such as Internet Explorer. You should see that the XML showplan output contains the ‘PlanGuideDB’ and ‘PlanGuideName’ tags as shown in bold in the following:
6. <ShowPlanXML xmlns=
7. "http://schemas.microsoft.com/sqlserver/2004/07/showplan"
8. Version="1.0" Build="9.00.1282.00">
9. <BatchSequence>
10. <Batch>
11. <Statements>
12. <StmtSimple PlanGuideDB="AdventureWorks"
13. PlanGuideName="AppendixPlanGuide"> ...
14. ...
15. </StmtSimple>
16. </Statements>
17. </Batch>
18. </BatchSequence>
19. </ShowPlanXML>

[Top Of Page](http://technet.microsoft.com/en-us/library/cc917694.aspx#mainSection)

|  |  |
| --- | --- |
| 1 | The RECOMPILE hint should be used very selectively, because it can consume valuable processor resources. This is especially true for queries that are frequently executed. |
| 2 | It is not possible in SQL Server to selectively force the join types of the joins involved in the query by using a query hint. |
| 3 | Refer to the SQL Server Books Online article, API Server Cursors, for additional details on server cursor APIs. |