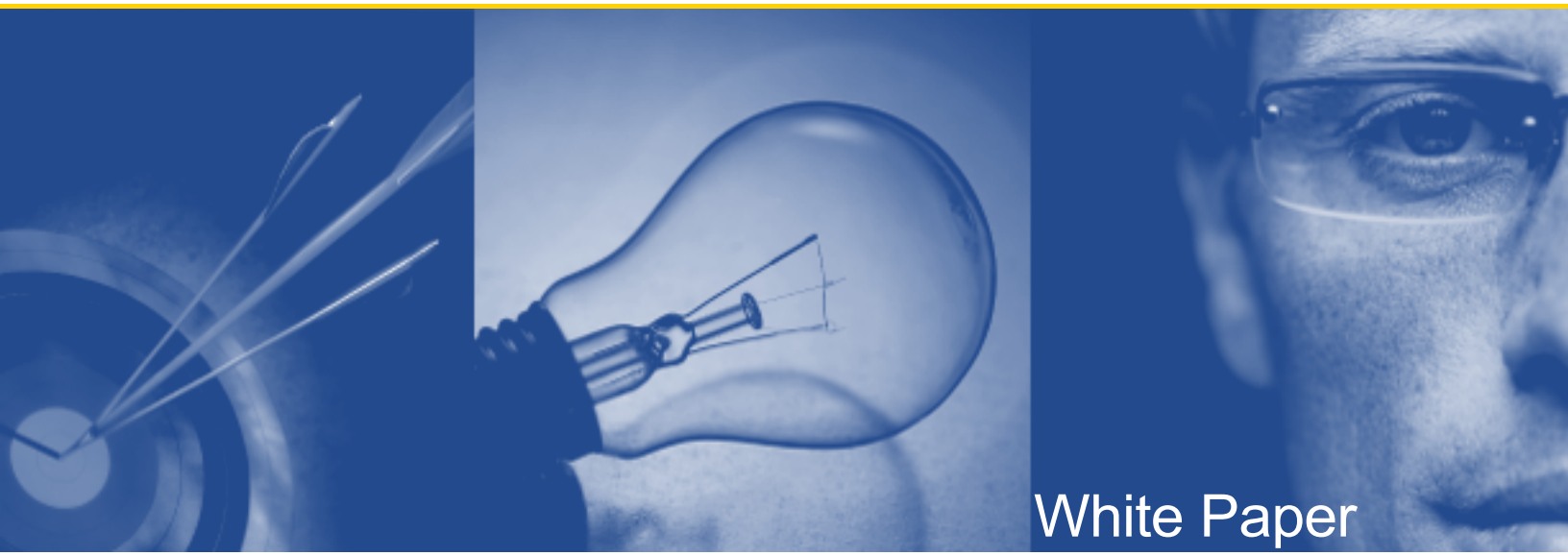
QUEST SOFTWARE®

# Query Tuning Strategies for Microsoft® SQL Server®

*Written by*
*Kevin Kline, Microsoft MVP since 2004*
*Technical Strategy Manager, Quest Software*



## WARRANTY

The information contained in this document is subject to change without notice. Quest Software makes no warranty of any kind with respect to this information. QUEST SOFTWARE SPECIFICALLY DISCLAIMS THE IMPLIED WARRANTY OF THE MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Quest Software shall not be liable for any direct, indirect, incidental, consequential, or other damage alleged in connection with the furnishing or use of this information.

## TRADEMARKS



World Headquarters
5 Polaris Way
Aliso Viejo, CA 92656
www.quest.com
e-mail: info@quest.com

Please refer to our Web site (www.quest.com) for regional and international office information.

Updated—November 2009

# CONTENTS

# OVERVIEW

This white paper offers useful techniques for improving queries in Microsoft SQL Server 2008. There are always a large number of tips and techniques applicable in narrow classes of programming tasks, each one offering a small improvement in performance. Knowing as many of these tuning tricks and techniques as possible expands your options when tuning for performance. In addition, knowing an effective process for analyzing query performance and behavior is an essential skill for any SQL Server professional.

The white paper introduces several basic elements that I've used with some success for tuning queries. In addition, it describes a handful of scenarios where poor performance is common, and provides recommendations for improvement.

The basic elements of query tuning that are covered include:

a) SET commands that show you what a query is doing

b) DBCC commands that help construct a useful "test jig" for query tuning

c) Key elements of an execution plan to consider

Most examples are based on either the venerable PUBS database, the NORTHWIND database, or on standard system tables. I have greatly expanded the size of the tables used in the PUBS database, adding tens of thousands of rows to many tables. You can find the PUBS database at http://codeplex.com/SqlServerSamples.

# WHAT'S MY QUERY DOING? AND WHY IS IT TAKING SO LONG?

There's a lot of instrumentation in SQL Server 2008 that helps you see what your query is doing behind the scenes to retrieve a given result set. You can use trace files, queries against Dynamic Management Views (DMVs) and Dynamic Management Functions (DMFs), and the many graphic features of SQL Server Management Studio (SSMS) to better illustrate the behaviors of a given SELECT statement.

As a long-time SQL Server tuner, I find that many of the old (dare I say "antiquated") methods of assessing a query are in fact the easiest and most effective. And by effective, I mean that they return the most actionable information in the least amount of time and with the least amount of personal, human intervention. Yes, the graphic tools can provide more information than the old scripted SET statements available in Transact-SQL. However, the graphic tools require that you put your hand on the mouse and click—a lot. That means the information is neither immediately actionable (because you have to point and click a lot to get the information) nor is it something that you can easily script to run in the off-hours when you're not at your desk.

The commands I like to use are:

SET STATISTICS I/O

> Shows the overall I/O of the query, including the number of scans performed, the number of logical reads performed (reads from cache), the number of physical reads performed (reads from disk), and the number of read-aheads performed (the number of pages placed in cache in anticipation of future reads). Since I/O is often one of the biggest bottlenecks for a query, it's important to know its overall I/O utilization and to compare the I/O utilization of two (or more) alternative queries.

SET STATISTICS I/O can return inaccurate I/O counts on queries that involve LOBs.

SET STATISTICS TIME

> Shows the total elapsed time (i.e. the round-trip time) of the query, as well as the CPU time consumed to parse, compile and execute the query. The round-trip time is dependent upon the total activity on the server, while the CPU time is independent of the total activity on the server. (Note—SET STATISTICS TIME may return inaccurate results for queries on servers running in fibre mode.)

SET SHOWPLAN_ALL

> Shows the estimated (not actual) execution plan chosen for a given query in hierarchical format that is representative of the steps taken by the query engine to process the query. The pipe marks in the output indicate the general level of the statement, with more of the first actions of the query appearing at the bottom of the output and working their way upward. You can use SET SHOWPLAN_TEXT for a subset of output returned by SET SHOWPLAN_ALL, which is useful when performing query tuning via a scripted method, such as the OSQL utility. Conversely, you can use the SET SHOWPLAN_XML statement to get even more data about the query than that provided by SET SHOWPLAN_ALL. It's up to you as to which you might like to use.

It's important to remember that, as with any SET statement, the statement remains in effect until explicitly disabled with the OFF subclause once it's been enabled with the ON subclause. For example, the following Transact-SQL code will show the I/O, time and execution plan of only the single query:

```
SET STATISTICS IO ON
SET STATISTICS TIME ON
SET SHOWPLAN_ALL ON
GO

SELECT st.stor_name AS 'Store',
       ISNULL((SELECT SUM(bs.qty)
          FROM sales AS bs
          WHERE bs.stor_id = st.stor_id), 0)
```

```
       AS 'Books Sold'
FROM   stores AS st
WHERE  st.stor_id IN
   (SELECT DISTINCT stor_id
    FROM sales)
GO

SET STATISTICS IO OFF
SET STATISTICS TIME OFF
SET SHOWPLAN_ALL OFF
GO
```

In the preceding query, if the final SET…OFF statements were not included, all subsequent statements would also return the I/O, time and execution plans of those statements. Note that the results are displayed in the MESSAGES tab of SSMS and not in the RESULTS tab.

Remember that the SET SHOWPLAN_ALL ON statement only displays the estimated execution plan. If you wish to see the actual execution plan of a query, use the SET STATISTICS PROFILE statement.

# SET STATISTICS I/O

The command SET STATISTICS IO ON forces SQL Server to report actual I/O activity on executed transactions. Once the option is enabled, every query thereafter produces additional output that contains I/O statistics. To disable the option, execute SET STATISTICS IO OFF.

For example, the following script obtains I/O statistics for a simple query counting rows of the "Employees" table in the NORTHWIND database:

```
SET STATISTICS IO ON
GO
SELECT COUNT(*) FROM employees
GO
SET STATISTICS IO OFF
GO
```

Results:

```
-----------
2977

Table 'Employees'. Scan count 1, logical reads 53, physical reads 0, read-ahead reads 0.
```

The scan count tells us the number of scans performed. Logical reads show the number of pages read from the cache. Physical reads show the number of pages read from the disk. Read-ahead reads indicate the number of pages placed in the cache in anticipation of future reads.

Additionally, you would execute a system stored procedure to obtain table size information for your analysis:

```
sp_spaceused employees
```

Results:

```
name       rows reserved  data    index_size  unused
---------- ---- --------- ------- ----------- -------
Employees  2977 2008 KB   1504 KB 448 KB      56 KB
```

What can you tell by looking at this information?

The query did not have to scan the whole table. The volume of data in the table is more than 1.5 megabytes, yet it took only 53 logical I/O operations to obtain the result. This indicates that the query has found an index that could be used to compute the result, and scanning the index took less I/O than it would take to scan all data pages.

Index pages were mostly found in the data cache since the physical reads value is zero. This is because the query was executed shortly after other queries on the Employees table, and the table and its index were already cached. Your mileage may vary.

SQL Server has reported no read-ahead activity. In this case, data and index pages were already cached. A table scan on a large table read-ahead would probably kick in and cache necessary pages before your query requested them. Read-ahead turns on automatically when SQL Server determines that your transaction is reading database pages sequentially. A separate SQL Server connection runs ahead of your process and caches data pages for it. Configuration and tuning of read-ahead parameters is beyond the scope of this paper.

In this example, the query was executed as efficiently as possible. No further tuning is required.

# SET STATISTICS TIME

Elapsed time of a transaction is a volatile measurement, since it depends on activity of other users on the server. However, it provides some real measurement, compared to the number of data pages, which doesn't mean anything to your users. They are concerned about seconds and minutes they spend waiting for a query to come back, not about data caches and read-ahead efficiency. The SET STATISTICS TIME ON command reports the actual elapsed time and CPU utilization for every query that follows. Executing SET STATISTICS TIME OFF suppresses the option.

```
SET STATISTICS TIME ON
GO
SELECT COUNT(*) FROM titleauthors
GO
SET STATISTICS TIME OFF
GO
```

Results:

```
SQL Server Execution Times:
   cpu time = 0 ms. elapsed time = 8672 ms.
SQL Server Parse and Compile Time:
   cpu time = 10 ms.

-----------
25

(1 row(s) affected)

SQL Server Execution Times:
   cpu time = 0 ms. elapsed time = 10 ms.
SQL Server Parse and Compile Time:
   cpu time = 0 ms.
```

The first message reports a somewhat confusing elapsed time value of 8,672 milliseconds. This number is not related to the script and indicates the amount of time that has passed since the previous command execution. You may disregard this first message. It took SQL Server only 10 milliseconds to parse and compile the query. It took 0 milliseconds to execute it (shown after the result of the query). What this really means is that the duration of the query was too short to measure. The last message that reports parse and compile time of 0 ms, refers to the SET STATISTICS TIME OFF command (that's the time it took to compile it). You may disregard this message since the most important messages in the output are highlighted.

Note that elapsed and CPU time are shown in milliseconds. The numbers may vary on your computer between runs of the query because the time values are dependent on total server load. In other words, every time you execute this script you may get slightly different statistics depending on what else your SQL Server was processing at the same time.

If you need to measure elapsed duration of a set of queries or a stored procedure, it may be more practical to implement it programmatically (shown below). The reason is that the STATISICS TIME reports duration of every single query and you have to add things up manually when you run multiple commands. Imagine the size of the output and the amount of manual work in cases when you time a script that executes a set of queries thousands of times in a loop!

Instead consider the following script to capture time before and after the transaction and report the total duration in seconds (you may use milliseconds if you prefer):

```
DECLARE @start_time DATETIME
SELECT  @start_time = GETDATE()

< any query or a script that you want to time, without a GO >

SELECT  'Elapsed Time, sec' = DATEDIFF( second, @start_time, GETDATE() )
GO
```

If your script consists of several steps separated by GO, you can't use a local variable to save the start time. A variable is destroyed at the end of the step, defined by the GO command, where it was created. But you can preserve start time in a temporary table like this:

```
CREATE TABLE #save_time ( start_time DATETIME NOT NULL )
INSERT #save_time VALUES ( GETDATE() )
GO
< any script that you want to time (may include GO) >
GO
SELECT  'Elapsed Time, sec' = DATEDIFF( second, start_time, GETDATE() )
FROM    #save_time
DROP TABLE #save_time
GO
```

Remember that SQL Server's DATETIME datatype stores time values in three millisecond increments. It is impossible to get more granular time values than those using the DATETIME datatype. In SQL Server 2008, you can opt for using the DATETIME2 datatype if you need greater granularity with the time measurement.

If you'd like to discover more about how to read and interpret execution plans, here are a few articles and posts:

- http://sqlserverpedia.com/wiki/Examining_Query_Execution_Plans
- http://sqlserverpedia.com/blog/sql-server-2005/three-kinds-of-execution-plans/
- http://sqlserverpedia.com/blog/sql-server-bloggers/understanding-statistics-io-2/

## What's a Test Jig? I Don't Like Dancing!

No, a test jig is not a kind of dance. Although the word jig commonly refers to certain Irish and English dances from Jane Austin's day, this usage in software engineering comes from the old industrial days when a jig referred to a special kind of box. The box served as a framework to hold work firmly in place so that an artisan could mill, drill, or otherwise perform precision work with both hands. And so, you're doing something similar with your Transact-SQL code. People also refer to this sort of code as a test harness.

Typically, you'll use a test jig to ensure that all of the types of cache, both buffer cache for data and procedure cache for objects, are uniformly clean. This enables you to ensure that an improvement in query performance is due to the change you made in the code and not due to unexpectedly cached objects or data. There are three commands that can help you control buffer and procedure caching while you test the SQL query:

DBCC FREEPROCCACHE [ ( { plan_handle | sql_handle | pool_name } ) ] [ WITH NO_INFOMSGS ]

> Clears the entire procedure cache of the SQL Server 2008 instance if you provide no parameters. Otherwise, it clears the procedure cache of the single execution plan from the cache (using the supplied plan handle or SQL handle) or all workload groups for the named resource pool on SQL Server 2008 Enterprise Edition using resource governor. On SQL Server 2005, the statement works only to clear the entire cache.

DBCC FLUSHPROCINDB(<DBID>)

Clears all execution plans from the procedure cache of the SQL Server instance for the specified database ID.

DBCC DROPCLEANBUFFERS [ WITH NO_INFOMSGS ]

Clears the buffer cache (i.e., data) of all clean buffers. To ensure that the buffer cache contains only clean buffers, first execute the CHECKPOINT statement to force all dirty pages to disk. After a CHECKPOINT, the DROPCLEANBUFFERS statement removes all data from the buffer cache.

The subclause WITH NO_INFOMSGS simply suppresses informational messages returned by the DBCC commands.

Do not run these commands on a production system. Although cleaning the procedure and buffer cache are very important for testing the development of a query, you should not execute these commands hastily since they will literally clear the caches of the current instance of SQL Server. If the system is used in any production capacity, then queries may experience a precipitous decline in performance until the applicable data and/or objects are reloaded into the cache. In general, you will only perform these statements within your local development environment.

So building on the example at the end of the previous section, your test jig should look like this:

```
DBCC DROPCLEANBUFFERS
DBCC FREEPROCCACHE
GO

SET STATISTICS IO ON
SET STATISTICS TIME ON
GO

SET SHOWPLAN_ALL ON
GO

SELECT st.stor_name AS 'Store',
       ISNULL((SELECT SUM(bs.qty)
          FROM sales AS bs
          WHERE bs.stor_id = st.stor_id), 0)
       AS 'Books Sold'
FROM   stores AS st
WHERE  st.stor_id IN
   (SELECT DISTINCT stor_id
    FROM sales)
GO

SET STATISTICS IO OFF
SET STATISTICS TIME OFF
GO

SET SHOWPLAN_ALL OFF
GO
```

Notice that the SET SHOWPLAN_ALL statement is separated by its own GO delimiter. That's because the SET SHOWPLAN_ALL statement must be executed alone in its own Transact-SQL batch.

TIP FOR SAN USERS – These DBCC commands only purge the cache of SQL Server, not of the storage subsystems. When using a Storage Area Network (SAN), the SAN controller may keep the data cached as well, thereby making repeated runs of the same query faster. Simply running one "before" query and one "after" query may look like the "after" query is faster—when in fact, the data is just being pulled from the SAN's cache. To eliminate this variable, perform repeated runs of the same query multiple times and average the results together.

There are a number of ways that you can get additional information about what is happening with your queries on an instance of Microsoft SQL Server 2008. While a full discussion of these DMVs and DMFs are beyond the scope of this white paper, you can find the query PLAN_HANDLE using the following four DMVs: sys.dm_exec_cached plans, sys.dm_exec_requests, sys.dm_exec_query_memory_grants and sys.dm_exec_query_stats. You can find the query SQL_HANDLE in these five DMVs: sys.dm_exec_query_stats, sys.dm_exec_requests, sys.dm_exec_cursors, sys.dm_exec_xml_handles and sys.dm_exec_query_memory_grants. Resource governor POOL_NAME values are found in the DMV sys.dm_resource_governor_resource_pools.

Remember, as with any DMV, the data retrieved by a query against the DMV represents the current conditions of the SQL Server instance. Therefore, a query against a DMV may not return information about a particular query because it has aged out of the cache or the cache has been cleaned.

### CONTEXT Isn't Just a Prison Story

There are a lot of good programming practices that are commonly used; for example, including a header comment block at the beginning of your Transact-SQL program stating who wrote the program, when it was written, and why. Later, other programmers will update the header block with their own comments describing changes and updates to the original program.

Here's one tip that you might want to incorporate into your standardized coding practices. One element of good programming that isn't very commonly used is to set the CONTEXT_INFO value for a query or a Transact-SQL program. Context information is optional trace information which is created for a session using the SET CONTEXT_INFO statement, thereby associating up to 128 bytes of binary data with the current session or connection. The context information data, which cannot be null, is then stored in the CONTEXT_INFO columns of sys.dm_exec_requests, sys.dm_exec_sessions and sys.sysprocesses. (In SQL Server 2000, the same data is stored in master.dbo.sysprocesses.)

```
A basic implementation of CONTEXT_INFO might appear as follows:
DECLARE @Ctxt varbinary(128)
SET @Ctxt = CONVERT(varbinary(128),'Mary''s session')
SET CONTEXT_INFO @Ctxt
GO
```

You can then see the context information when you check on SQL Server's activity in SSMS under Management >> Activity Monitor >> View Processes. For your production implementation of the code, you might put the Windows or SQL Server login ID into the context information so that you can immediately tell what user is responsible for a given process on the SQL Server instance.

# SET SHOWPLAN

The SET SHOWPLAN statement is an excellent way to reveal the execution plan of a query or transaction. You can also use the graphic showplan option in SSMS to see a query's execution plan, which definitely has its advantages in certain situations. The graphic showplan option in SSMS is especially useful when doing side-by-side comparisons of two queries, as you'll soon see in some examples below. These tools will show you exactly how much processing, as a percentage, was consumed by each step of a batch. The graphic showplan tells you which alternatives are more, or less, costly to the query engine. You can also run two (or more) queries in a single batch and see which one performed the best.

If you just want quick and dirty execution plan details, then use the SET SHOWPLAN_TEXT variant of the statement. The SET SHOWPLAN_ALL variant shows a great deal more information about execution plans. And to get the motherload of information about a given query's execution plan, use the SET SHOWPLAN_XML variant. In this white paper, the examples will primarily use SET SHOWPLAN_TEXT.

Whichever method you use, there are a few operations that usually indicate inordinate resources consumption or an ineffective plan. Some operations to watch out for include the following:

LOOKUP

PARALLELISM

SCANS

SPOOLS

Remember, these operations are not necessarily bad. Their presence in a query plan may simply indicate that the given operation is the only available means of answering the query. To use an analogy, sports cars are fast, but you'll need a minivan to transport eight people at once. In the same way, there are sometimes faster operations available to the optimizer (such as SEEK compared to SCAN), but sometimes the slower operation is the only method available to accomplish what you want. Nevertheless, when there are multiple ways that the query can be coded, you might opt for strategies that minimize the presence of slower-running operations in favor of other, better-performing alternatives.

## An Execution Plan Does Not Require an Executioner

For some reason, the term "execution plan" invokes images of a burly medieval man with a big ax, clad in a tunic and a dark hood concealing all facial features except for a pair of glowering, pupilless eyes. Fortunately, we're not talking about that sort of execution plan. In SQL Server, an execution plan (also known as a query plan or optimizer plan or—for those who can't shake their Oracle background—the explain plan) is the set of operations that the SQL Server query optimizer must perform to return the result set requested by a given query. You can discover all of a given query's behind-the-scenes operations by using the SET SHOWPLAN_ALL ON statement presented earlier.

In general, when you examine an execution plan for tuning opportunities, you're looking for a short-list of red flags, such as:

- The query isn't using useful indexes, often because a useful index is missing or the query optimizer somehow overlooks a useful index
- The query isn't using good statistics for an index, because the statistics are out of date (also called "stale statistics") or aren't representative of the distribution of values in the index (that is, the index's cardinality)
- The query isn't using partitions properly
- The transaction is dealing with I/O issues, often the result of a data or index hot spot
- And when we compare two queries against each other, we're often looking for an indication that one version of a query is raising a red flag where another version of the query, which returns the same result set, is not

## Yes, Sir! SARG, Sir!

The single most important construct in a query affecting whether you'll see any of the above-mentioned red flags within the execution plan comes in the form of a search argument, better known as a SARG. A SARG itself come in two forms: as an element of the query's WHERE clause and as an element of the query's JOIN clause. A SARG is a filter. It restricts the result set of a query so that it effectively answers your question. A SARG has a major impact on one of the two ways that the query optimizer is able to figure out the expense of a given query:

1. The number of rows processed at each level of the query, also known as cardinality, and used as an input to the next item (below);
2. The cost of the algorithm according to the kind of operators used in the query.

We know from database design recommendations that we should create indexes on tables to increase the speed with which we can access the data. When we build indexes, we are implicitly improving the cardinality of the parent table. Although there are enough rules and best practices about the creation and use of indexes to fill a separate white paper, you will generally want to create indexes on the columns of a table that are frequently referenced in WHERE clauses and JOIN clauses. As you'll see in later examples within this document, a well-formed SARG (that is, a WHERE clause or JOIN clause) can make or break the performance of a query. Alternately, a bad SARG can cause a query to use a less than optimal index or, in some cases, to completely ignore an existing index.

Currently, the list of standard search argument operators is composed of: =, >, <, =>, <=, BETWEEN and LIKE.

## Which Is Better? Comparing Two Variants as Illustrated by SEEK or SCAN Operations

You'll use execution plan information to compare the relative effectiveness of two separate queries. For example, you might want to see if one query, compared to another, adds extra layers of overhead or chooses a different and non-optimal indexing strategy.

In the previous sections, you were shown a few very important techniques that will help you determine what the overall workload of a query might be. Now, let's take a look at a couple simple variations on a theme for a single query.

One of the first things you'll need to distinguish in an execution plan is the difference between a SEEK and a SCAN operation. A simple but useful rule of thumb is that a SEEK operation performs the best, while a SCAN operation is less than optimal, if not overtly bad. A SEEK goes quickly, via an index, to the affected records, while a SCAN reads the entire object, whether it is a table, a clustered index (which is essentially the entire table) or a non-clustered index. Thus, a SCAN usually consumes a lot more resources than a SEEK. If your execution plan shows SCANs only, then you should consider tuning the query.

In the following examples, we compare two queries, both of which are attempting to return all sales records from the big_sales table, where the store's ID number is in the 6,300s. The first variant uses the LIKE operator, while the second variant checks for a value of a substring using the SUBSTRING function:

```
SELECT *
FROM big_sales
WHERE stor_id LIKE '63%'
```

The execution plan and statistics I/O returned by the previous query looks like this:

```
|--Clustered Index Seek(OBJECT:([big_sales].[UPKCL_big_sales]),
   SEEK:([ big_sales].[stor_id] >= '62þ' AND [big_sales].[stor_id] < '64'),
   WHERE:([ big_sales].[stor_id] like '63%') ORDERED FORWARD)

Table 'big_sales'. Scan count 1, logical reads 10, physical reads 0
```

The query above is able to perform a SEEK for specific values against the clustered index, consuming a very light number of scans and logical reads. The SHOWPLAN describes exactly what the seek operation is based upon—the unique, primary key clustered on the value of the stor_id column. It also reveals that the results are "like '63%'" and ORDERED according to how they are currently stored in the index mentioned. Since SQL Server supports forward and backward scrolling through indexes, with equally good performance for both, you might see either ORDERED FORWARD or ORDERED BACKWARD in the execution plan. This merely tells you which direction the table or index was read. You can even manipulate this behavior by using the ASC and DESC keywords in an ORDER BY clause, if you've used one.

Compare this to a query that returns the same result set using a SUBSTRING function:

```
SELECT *
FROM big_sales
WHERE SUBSTRING(stor_id,1,2) = '63'
```

The execution plan and statistics I/O returned by the previous query looks like this:

```
|--Clustered Index Scan(OBJECT:([big_sales].[UPKCL_big_sales]),
   WHERE:(substring([big_sales].[stor_id],(1),(2))='63'))

Table 'big_sales'. Scan count 1, logical reads 79, physical reads 0
```

Even though both queries retrieved the same result set, the first query with its simple CLUSTERED INDEX SEEK operation is a better execution plan than the second query with its CLUSTERED INDEX SCAN. In this case, the statistics I/O count for the query, using the SUBSTRING function, shows that it consumed **almost eight times as many logical reads** to return the same result set!

> A SEEK operation is invoked by either a WHERE clause or a JOIN clause. A SCAN operation, while certainly able to retrieve data, is less efficient.

In the case of the earlier examples using LIKE and SUBSTRING, the predicate tells you what records the WHERE clause filters from the result set. Because this is done as a component of the SEEK or SCAN operation, the WHERE predicate often neither hurts nor helps performance of the operation itself. What the WHERE clause does do is help the query optimizer find the best possible indexes to apply to the query. In the case of the query using SUBSTRING, the function applied to the column stor_id causes SQL Server to disregard any indexes existing on that column and forces a scan.

Another useful way to compare the queries is to execute them both in a single query window of SSMS while displaying the estimated execution plan (either by choosing Query >> Display Estimated Execution Plan or by right-clicking in the query window and then choosing Display Estimated Execution Plan). When two or more query execution plans are displayed, SQL Server shows the relative cost of each batch as the first line of the report for each query. Thus, by looking at Figure 1 below, we can see that the variant of the query that uses the LIKE comparison operator uses only 12 percent of the total resources consumed by the batch, while the variant of the query that used the SUBSTRING function consumed the lion's share of the resources used at 88 percent.

If you'd like to discover more about how indexes make queries perform faster, here are a few articles and posts:

- http://sqlserverpedia.com/wiki/Index_Selectivity_and_Column_Order
- http://sqlserverpedia.com/blog/sql-server-bloggers/when-is-a-seek-actually-a-scan/
- http://sqlserverpedia.com/blog/sql-server-bloggers/catch-all-queries/

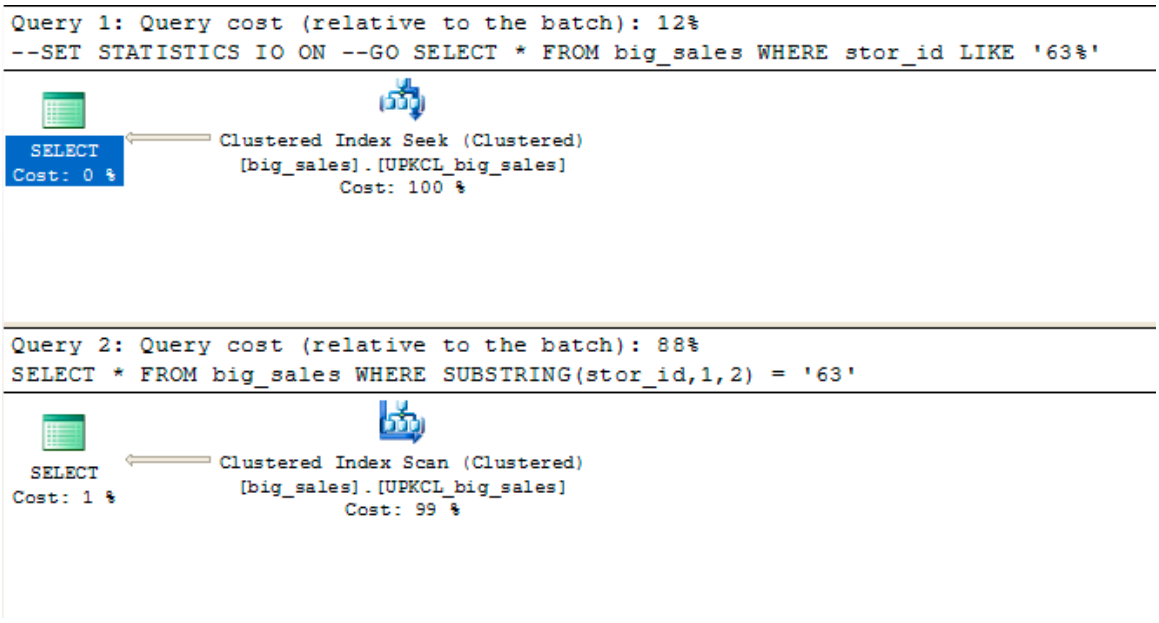



**Figure 1: Examining Two Execution Plans Side-By-Side in SSMS**

# SPECIAL CASE SCENARIOS FOR QUERY TUNING

The first part of this white paper showed you a simple, textual method for analyzing the resources consumed by a query, and how that query is processed by the optimizer to return a particular result set. Now, in the following section, we will explore several common query scenarios in which better-performing alternatives are available.

## Functions and Expressions That Suppress Indexes

In many cases, the optimizer cannot use indexes on the columns of a SARG when built-in functions or expressions are applied to an indexed column. The example immediately before this section, in which a SUBSTRING function was applied to an indexed column, illustrates this principle in action. To avoid this problem, try to rewrite these conditions so that index keys are not involved in the expression or built-in function.

In effect, you must help SQL Server by removing any expressions around columns that are indexed. Otherwise, SQL Server may not be able to utilize the index. In some cases, when you apply an expression or function to the column, the index is suppressed. But once you move the expression or function from the column to the value that the SARG is evaluated against, the index is usable once again.

The following queries select a row from a table using a SARG based upon a column which also serves as the unique clustered index. In each case, the column referenced in the WHERE clause is an indexed column. The reason that the index is suppressed is shown as a comment in the code.

| QUERY WITH SUPPRESSED INDEX | OPTIMIZED QUERY USING INDEX |
|---|---|
| SELECT *<br>FROM big_sales<br>WHERE SUBSTRING(stor_id,1,2) = '63'<br>-- function on the indexed column | SELECT *<br>FROM big_sales<br>WHERE stor_id LIKE '63%' |
| SELECT *<br>FROM big_sales<br>WHERE (stor_id-146) = '7896'<br>-- implicit conversion and expression<br>-- on the indexed column | SELECT *<br>FROM big_sales<br>WHERE stor_id = '8042' |
| SELECT *<br>FROM jobs<br>WHERE (job_id + 7) = 14<br>-- mathematic expression on the<br>-- indexed column | SELECT *<br>FROM jobs<br>WHERE job_id = 7 |
| DECLARE @job_id VARCHAR(5)<br>SELECT @job_id = '2'<br>SELECT *<br>FROM jobs | DECLARE @job_id VARCHAR(5)<br>SELECT @job_id = '2'<br>SELECT *<br>FROM jobs |

| QUERY WITH SUPPRESSED INDEX | OPTIMIZED QUERY USING INDEX |
|---|---|
| `WHERE CAST(job_id AS VARCHAR(5)) = @job_id`<br>`-- explicit conversion on the indexed`<br>`-- column` | `WHERE job_id = CAST(@job_id AS SMALLINT)` |
| `CREATE INDEX employee_hire_date`<br>`ON employee ( hire_date )`<br>`GO`<br>`-- Get all employees hired`<br>`-- in the 1st quarter of 1990`<br><br>`SELECT hire_date`<br>`FROM employee`<br>`WHERE DATEPART( year, hire_date )`<br>`= 1990`<br>`AND DATEPART( quarter, hire_date )`<br>`= 1`<br><br>`-- Function on the indexed column` | `CREATE INDEX employee_hire_date`<br>`ON employee ( hire_date )`<br>`GO`<br>`-- Get all employees hired`<br>`-- in the 1st quarter of 1990`<br><br>`SELECT hire_date`<br>`FROM employee`<br>`WHERE hire_date >= '1/1/1990'`<br>`AND hire_date < '4/1/1990'` |

The quick and dirty way to remember this issue when writing a query is that the SARG should place any functions or mathematical expressions not upon a column in the search argument, but upon the value to which it is compared. However, there is a bit more to it than that. The following list details all the situations where SQL Server has difficulty accurately calculating the cardinality of a given query:

1. Queries where the SARG compares values between different columns of the same table
2. Queries where the SARG uses operators with any of these characteristics:
   a) There are no statistics on the columns involved on either side of the operators
   b) The query should be a highly selective value set, but it is actually not uniformly distributed, especially if the operator is anything other than an equal sign (=) (for example, an indexed ZIP CODE column that contains millions of records but only a handful are not "90120")
   c) The SARG uses the not equal to (!=) comparison operator or the NOT logical operator, especially if the column allows NULL
3. Queries that use any of the SQL Server built-in functions or a scalar-valued, user-defined function whose argument is not a constant value (as shown above)
4. Queries that involve joining columns through arithmetic or string concatenation operators (as shown above)
5. Queries that compare variables whose values are not known when the query execution plan is built [i.e. when it is compiled and optimized, for example WHERE CONVERT(INT, my_column) = @my_val].

# Head Fakes to the Query Optimizer

Sometimes, when turning a query, you want to force SQL Server to explore other ways to build the execution plan without resorting to a query hint. A common way to achieve this result is to use a function call, such as COALESCE, to force a new query plan. (You could similarly use other functions, such as ISNULL or NULLIF to get the same behavior.)

Normally, COALESCE returns the first nonnull expression from its arguments. But when you enter two arguments, both of which are identical, COALESCE will not affect the result set but will indeed force a different execution plan. For example:

| QUERY WITHOUT COALESCE | QUERY WITH COALESCE |
|---|---|
| `SELECT s1.stor_id`<br>`FROM sales S1, stores S2`<br>`WHERE s1.stor_id = s2.stor_id` | `SELECT s1.stor_id`<br>`FROM sales S1, stores S2`<br>`WHERE s1.stor_id =`<br>`  COALESCE(s2.stor_id, s2.stor_id)` |

The COALESCE transformation usually has one of the following effects on an execution plan: either to rearrange the joining path or to disable internal database transformations.

As described earlier in the section "Functions and Expressions that Suppress Indexes," using the COALESCE function call will stop the index search on s2.stor_id in our example above.

A similar situation can arise when addressing the transitive property of evaluations. You might recall the transitive property from high school algebra in which we can say "If A = B, and B = C, then A = C". Well, you'd think that the query optimizer would always be able to apply the transitive property to our queries and thus transform this query:

```
SELECT s1.stor_id
FROM sales S1, stores S2
WHERE s1.stor_id = s2.stor_id
  AND s1.stor_id = 6380
```

Into this query:

```
SELECT s1.stor_id
FROM sales S1, stores S2
WHERE s1.stor_id = s2.stor_id
  AND s1.stor_id = 6380
  AND s2.stor_id = 6380
```

But transitive conversions don't always happen automatically. Because it's rather unpredictable as to when the transitive property will be applied behind the scenes by the query optimizer, if it's applied at all, it's usually advisable to hard-code the transitive conversions yourself.

Note: using the COALESCE function call or transitive SARGs does not always guarantee better performance. You'll have to check the performance of each variation of the query explicitly to see which performs best. So, while head faking the query optimizer is not guaranteed to always improve your performance, it will usually alter the execution plan of a query and allow you to try new alternatives.

# Subqueries Optimization

As a good rule of thumb, try to replace subqueries with joins where possible. The optimizer may sometimes automatically flatten out subqueries and replace them with regular or outer joins. But it doesn't always do a good job at that. Explicit joins give the optimizer more options to choose the order of tables and find the best possible plan. When you optimize a particular query, investigate if getting rid of subqueries makes a difference.

## Example

The following queries select the names of all user tables in the PUBS database and the clustered index name for each table if one exists. If there is no clustered index, then table name still appears in the list with a dash in the clustered index column. Both queries return the same result set, but the first one uses a subquery, while the second employs an outer join. Compare the execution plans produced by Microsoft SQL Server.

| SUBQUERY SOLUTION | JOIN SOLUTION |
|---|---|
| SELECT st.stor_name AS 'Store',<br>ISNULL((SELECT SUM(bs.qty)<br>FROM sales AS bs<br>WHERE bs.stor_id = st.stor_id), 0)<br>AS 'Books Sold'<br>FROM stores AS st<br>WHERE st.stor_id IN<br>(SELECT DISTINCT stor_id<br>FROM sales) | SELECT st.stor_name AS 'Store',<br>SUM(bs.qty) AS 'Books Sold'<br>FROM stores AS st<br>JOIN sales AS bs<br>ON bs.stor_id = st.stor_id<br>GROUP BY st.stor_name |
| \|--Compute Scalar<br>(DEFINE:([Expr1009]=isnull<br>([Expr1007],(0))))<br>\|--Nested Loops(Left Outer Join,<br>OUTER REFERENCES:([st].[stor_id]))<br>\|--Nested Loops(Left Semi Join,<br>WHERE:([ stores].[stor_id] as<br>[st].[stor_id]=[sales].[stor_id]))<br>\| \|--Clustered Index Scan (OBJECT:<br>([ stores].[UPK_storeid] AS [st]))<br>\| \|--Clustered Index Scan<br>(OBJECT:([ sales].[UPKCL_sales]))<br>\|--Stream Aggregate<br>(DEFINE:([Expr1007]=<br>SUM([sales].[qty] as [bs].[qty])))<br>\|--Clustered Index Seek (OBJECT: | \|--Stream Aggregate<br>(GROUP BY:([st].[stor_name]) DEFINE:<br>([Expr1004]=SUM([sales].[qty] as<br>[bs].[qty])))<br>\|--Nested Loops(Inner Join, OUTER<br>REFERENCES:([st].[stor_id]))<br>\|--Sort(ORDER BY:([st].[stor_name] ASC))<br>\| \|--Clustered Index Scan (OBJECT:<br>([ stores].[UPK_storeid] AS<br>[st]))<br>\|--Clustered Index Seek (OBJECT:<br>([ sales].[UPKCL_sales] AS [bs]),<br>SEEK:([bs].[stor_id]=[<br>stores].[stor_id]<br>as [st].[stor_id])) |

| SUBQUERY SOLUTION | JOIN SOLUTION |
|---|---|
| ([sales]. [UPKCL_sales] AS [bs]), SEEK:([bs].[stor_id]= [stores].[stor_id] as [st].[stor_id]) ) | |
| Table 'sales'. Scan count 7, logical reads 24, physical reads 0, read-ahead reads 0.<br><br>Table 'stores'. Scan count 1, logical reads 2, physical reads 0, read-ahead reads 0. | Table 'sales'. Scan count 6, logical reads 12, physical reads 0, read-ahead reads 0.<br><br>Table 'stores'. Scan count 1, logical reads 2, physical reads 0, read-ahead reads 0. |

Without probing deeper, we see that the join variation of the query required one fewer scan and half as many total logical reads as the subquery solution.

Incidentally, the result sets are the same in both cases, though the sort orders are different because the join query has an implicit ORDER BY clause due to its GROUP BY clause:

```
Store                                    Books Sold
---------------------------------------- -----------
Barnum's                                 154125
Bookbeat                                 518080
Doc-U-Mat: Quality Laundry and Books     581130
Eric the Read Books                      76931
Fricative Bookshop                       259060
News & Brews                             161090

(6 row(s) affected)

Store                                    Books Sold
---------------------------------------- -----------
Eric the Read Books                      76931
Barnum's                                 154125
News & Brews                             161090
Doc-U-Mat: Quality Laundry and Books     581130
Fricative Bookshop                       259060
Bookbeat                                 518080

(6 row(s) affected)
```

Notice that the execution plan of both queries contains two stream aggregate operations, but the placements of the operations are very different. When an expensive operation is nested within a query so that it must be performed on every iteration of a looping operation, the execution costs can add up quickly. In the examples above, the subquery variant of the query must perform a SUM for each line of the result set that’s retrieved. In comparison, the join variant of the query performs the SUM operation as the final, culminating operation of the query.

# UNION Vs. UNION ALL

The UNION ALL variant has a number of performance benefits over the more commonly used UNION statement. The difference is that UNION has a side effect of eliminating all duplicate rows and sorting results, which UNION ALL doesn't do.

Selecting a distinct result requires building a temporary worktable, storing all rows in it and sorting before producing the output. (Displaying the showplan on a SELECT DISTINCT query will reveal a stream aggregation is taking place, consuming as much as 30 percent of the resources used to process the query.) In some cases, when that's exactly what you need to do, UNION is your friend. But if you don't expect any duplicate rows in the result set or you don't care about the existence of duplicate records, then use UNION ALL. It simply selects from one query and then mashes subsequent result sets to the bottom of the first result set, as shown by the concatenation operation. UNION ALL requires no work table and no sorting (unless other conditions unrelated to the UNION ALL operator cause the creation of a worktable). One more potential problem with UNION is the danger of flooding tempdb database with a huge worktable. It may happen if you expect a large result set from a UNION query.

The following queries select the ID for all stores in the sales table, which ships as is with the PUBS database, and the ID for all stores in the big_sales table, a version of the sales table that was populated with more than 10,000 rows. The only difference between the two solutions is the use of UNION versus UNION ALL. But the addition of the ALL keyword makes a big difference in the execution plan.

The first solution requires stream aggregation and sorting the results before they are returned to the client. The second query is much more efficient, especially for large tables.

| **UNION SOLUTION** | **UNION ALL SOLUTION** |
|---|---|
| `SELECT stor_id FROM sales`<br>`UNION`<br>`SELECT stor_id FROM big_sales` | `SELECT stor_id FROM sales`<br>`UNION ALL`<br>`SELECT stor_id FROM big_sales` |
| `\|--Merge Join(Union)`<br>`  \|--Stream Aggregate(GROUP BY:([sales].`<br>`  [stor_id]))`<br>`  \|  \|--Clustered Index Scan (OBJECT:`<br>`  ([sales].[UPKCL_sales]))`<br>`  \|--Stream Aggregate(GROUP BY:`<br>`  ([big_sales]. [stor_id]))`<br>`  \|--Clustered Index Scan(OBJECT:`<br>`  ([big_sales].[UPKCL_big_sales]))` | `\|--Concatenation`<br>`  \|--Index Scan(OBJECT:([sales].`<br>`  [titleidind]))`<br>`  \|--Index Scan(OBJECT:([big_sales].`<br>`  [titleidind]))` |
| `(6 row(s) affected)`<br><br>`Table 'big_sales'. Scan count 1, logical reads 79, physical reads 0, read-ahead reads 0.`<br><br>`Table 'sales'. Scan count 1, logical reads 2, physical reads 0, read-ahead reads 0.` | `(10041 row(s) affected)`<br><br>`Table 'big_sales'. Scan count 1, logical reads 32, physical reads 0, read-ahead reads 0.`<br><br>`Table 'sales'. Scan count 1, logical reads 2, physical reads 0, read-ahead reads 0.` |

Although the result sets in this example are largely interchangeable, you can see that the UNION ALL statement consumed less than half of the resources that the UNION statement consumed. So be sure to anticipate your result sets, and in those that are already distinct, use the UNION ALL clause.

To learn more about UNION and UNION ALL, here's a related post from SQLServerPedia:

http://sqlserverpedia.com/blog/sql-server-bloggers/all-any-and-some-the-three-stooges/

# UPDATE…FROM and DELETE…FROM

T-SQL offers an extension to ANSI-SQL syntax for UPDATE and DELETE commands that may be very efficient in many cases. It allows you to specify a FROM clause and join several tables in an UPDATE or DELETE command. It's much easier to read an UPDATE or DELETE statement with a FROM clause. This is because you can easily distinguish the elements of the transaction that filter the UPDATE or DELETE operation upon the affected table (that is, the true WHERE clause) from those elements of the transaction that simply relate the records of the affected table to related tables (that is, the JOIN clause).

## Examples

These principles are illustrated using a few variations of an UPDATE statement. However, these principles apply equally to DELETE statements. To update the titleauthor table, the ANSI SQL solution below executes two correlated subqueries, while the UPDATE…FROM command shown later replaces the subqueries with more explicit joins.

```
UPDATE titleauthor
SET    royaltyper = 90
WHERE  au_id = (SELECT  au_id FROM  authors
                WHERE au_lname = 'Ringer' AND au_fname = 'Albert')
  AND  title_id = (SELECT  title_id FROM  titles
                   WHERE title = 'Life Without Fear')
```

This yields a rather complex execution plan (edited for brevity) shown here:

```
|--Clustered Index Update(OBJECT:([titleauthor].[UPKCL_taind]),
   SET:([titleauthor].[royaltyper] = 90))
     |--Compute Scalar(DEFINE:([Expr1011]=(90)))
          |--Nested Loops(Inner Join, WHERE:([titleauthor].[title_id]=[Expr1019]))
               |--Assert(WHERE:(CASE WHEN [Expr1018]>(1) THEN (0) ELSE NULL END))
               |    |--Stream Aggregate(DEFINE:([Expr1018]=Count(*),
                         [Expr1019]=ANY([titles].[title_id])))
               |       |--Index Seek(OBJECT:([titles].[titleind]),
                            SEEK:([titles].[title]='Life Without Fear') ORDERED FORWARD)
               |--Nested Loops(Inner Join, OUTER REFERENCES:([Expr1017]))
                    |--Assert(WHERE:(CASE WHEN [Expr1016]>(1) THEN (0) ELSE NULL END))
                    |    |--Stream Aggregate(DEFINE:([Expr1016]=Count(*),
                              [Expr1017]=ANY([authors].[au_id])))
                    |       |--Index Seek(OBJECT:([authors].[aunmind]),
                              SEEK:([authors].[au_lname]='Ringer'
                              AND [authors].[au_fname]='Albert') ORDERED FORWARD)
                    |--Index Seek(OBJECT:([titleauthor].[auidind]),
                       SEEK:([titleauthor].[au_id]=[Expr1017]) ORDERED FORWARD)
```

On the other hand, we can exploit the Transact-SQL extension, allowing the FROM clause and JOIN subclause in the UPDATE statement:

```
UPDATE  titleauthor
SET     royaltyper = 90
FROM    titleauthor AS ta
   JOIN authors AS a ON ta.au_id = a.au_id
   JOIN titles  AS t ON ta.title_id = t.title_id
WHERE (a.au_lname = 'Ringer' AND a.au_fname = 'Albert')
  AND (t.title = 'Life Without Fear')
```

This yields a simpler execution plan:

```
|--Clustered Index Update(OBJECT:([titleauthor].[UPKCL_taind]),
   SET:([titleauthor].[royaltyper] = [Expr1006]))
     |--Compute Scalar(DEFINE:([Expr1006]=(90)))
          |--Top(ROWCOUNT est 0)
               |--Nested Loops(Inner Join, OUTER REFERENCES:([ta].[title_id]))
                    |--Nested Loops(Inner Join, OUTER REFERENCES:([a].[au_id]))
                    |    |--Index Seek(OBJECT:([authors].[aunmind] AS [a]),
                           SEEK:([a].[au_lname]='Ringer'
                           AND [a].[au_fname]='Albert') ORDERED FORWARD)
                    |    |--Index Seek(OBJECT:([titleauthor].[auidind] AS [ta]),
                            SEEK:([ta].[au_id]=[authors].[au_id] as [a].[au_id])
                            ORDERED FORWARD)
                  |--Index Seek(OBJECT:([titles].[titleind] AS [t]),
                     SEEK:([t].[title]='Life Without Fear' AND
                     [t].[title_id]=[titleauthor].[title_id] as [ta].[title_id])
                     ORDERED FORWARD)
```

In the next example, a row is updated in the titles table that has a specific order recorded in the sales table. Note that the ANSI SQL solution has to execute essentially the same subquery twice, because column title_id is needed for the WHERE clause and the column qty is used in the SET clause.

ANSI SQL:

```
UPDATE  titles
SET     ytd_sales = ytd_sales + (
    SELECT  qty
    FROM    sales s
    WHERE   s.stor_id = '7131'
        AND s.ord_num = 'N914014' )
WHERE   title_id = (
    SELECT  title_id
    FROM    sales s
    WHERE   s.stor_id = '7131'
        AND s.ord_num = 'N914014' )
```

The execution plan for the ANSI SQL query follows:

```
|--Clustered Index Update(OBJECT:([titles].[UPKCL_titleidind]),
SET:([titles].[ytd_sales] = [Expr1009]))
     |--Compute
Scalar(DEFINE:([Expr1009]=[titles].[ytd_sales]+CONVERT_IMPLICIT(int,[Expr1015],0)))
          |--Nested Loops(Left Outer Join)
               |--Nested Loops(Inner Join, OUTER REFERENCES:([Expr1017]))
               |    |--Assert(WHERE:(CASE WHEN [Expr1016]>(1) THEN (0) ELSE NULL END))
               |    |    |--Stream Aggregate(DEFINE:([Expr1016]=Count(*),
                             [Expr1017]=ANY([sales].[title_id] as [s].[title_id])))
               |    |         |--Clustered Index Seek(OBJECT:([sales].[UPKCL_sales] AS
[s]),
                                  SEEK:([s].[stor_id]='7131' AND
[s].[ord_num]='N914014') ORDERED FORWARD)
               |    |--Clustered Index Seek(OBJECT:([titles].[UPKCL_titleidind]),
                        SEEK:([titles].[title_id]=[Expr1017]) ORDERED FORWARD)
               |--Assert(WHERE:(CASE WHEN [Expr1014]>(1) THEN (0) ELSE NULL END))
                    |--Stream Aggregate(DEFINE:([Expr1014]=Count(*),
                       [Expr1015]=ANY([sales].[qty] as [s].[qty])))
                         |--Clustered Index Seek(OBJECT:([sales].[UPKCL_sales] AS
[s]),
                            SEEK:([s].[stor_id]='7131' AND [s].[ord_num]='N914014')
ORDERED FORWARD)
```

Now compare the expansive ANSI SQL update operation shown above and the resultant execution plan with the SQL Server Transact-SQL extension:

```
UPDATE  titles
SET     ytd_sales = ytd_sales + s.qty
FROM titles AS t
   JOIN sales AS s ON t.title_id = s.title_id
WHERE s.stor_id = '7131'
    AND s.ord_num = 'N914014'
```

This produces an execution plan with only seven major operations (in comparison, the ANSI SQL plan had eleven):

```
|--Clustered Index Update(OBJECT:([pubs].[dbo].[titles].[UPKCL_titleidind]),
   SET:([pubs].[dbo].[titles].[ytd_sales] = [Expr1004]))
  |--Compute Scalar(DEFINE:([Expr1004]=[titles].[ytd_sales] as
[t].[ytd_sales]+[Expr1010]))
      |--Top(ROWCOUNT est 0)
           |--Nested Loops(Inner Join, OUTER REFERENCES:([s].[title_id]))
                |--Compute
Scalar(DEFINE:([Expr1010]=CONVERT_IMPLICIT(int,[sales].[qty] as [s].[qty],0)))
                |    |--Clustered Index Seek(OBJECT:([sales].[UPKCL_sales] AS [s]),
                        SEEK:([s].[stor_id]='7131' AND [s].[ord_num]='N914014')
ORDERED FORWARD)
                |--Clustered Index Seek(OBJECT:([titles].[UPKCL_titleidind] AS [t]),
                   SEEK:([t].[title_id]=[sales].[title_id] as [s].[title_id]) ORDERED
FORWARD)
```

As you might expect, the ANSI SQL query, with its two distinct subqueries, has a larger amount of read activity than that generated by the query containing the FROM…JOIN clause.

# TOP

The TOP clause of the SELECT statement limits the number of rows returned by a single transaction, whether it be a SELECT, INSERT, UPDATE or DELETE statement. This optional clause provides great efficiencies in numerous programming tasks.

Some practical tasks are much more efficient to program with TOP than with standard SQL commands. Let's demonstrate it using several examples. One of the most popular queries in almost any database is a request for the first (that is, the TOP n) items from a long list of selected rows. You could certainly make use of this feature, when returning result sets to a client, to ensure that the application grabs blocks of, say, 30 records at a time. In case of the PUBS database, we could search for the top five best-selling titles. Compare the two solutions—with TOP and using ANSI SQL.

Pure ANSI SQL:

```
SELECT  title, ytd_sales
FROM    titles a
WHERE   ( SELECT  COUNT(*)
          FROM    titles b
          WHERE   b.ytd_sales >
                  a.ytd_sales
        ) < 5
ORDER BY ytd_sales DESC
```

The textual execution plan for the ANSI SQL query looks like this:

```
StmtText
-------------------------------------------------------------------------------------
  |--Sort(ORDER BY:([a].[ytd_sales] DESC))
       |--Filter(WHERE:([Expr1004]<(5)))
            |--Nested Loops(Inner Join, OUTER REFERENCES:([a].[ytd_sales]))
                 |--Clustered Index
Scan(OBJECT:([pubs].[dbo].[titles].[UPKCL_titleidind] AS [a]))
                 |--Compute
Scalar(DEFINE:([Expr1004]=CONVERT_IMPLICIT(int,[Expr1008],0)))
                      |--Stream Aggregate(DEFINE:([Expr1008]=Count(*)))
                           |--Clustered Index
Scan(OBJECT:([pubs].[dbo].[titles].[UPKCL_titleidind] AS
                               [b]), WHERE:([pubs].[dbo].[titles].[ytd_sales] as
                               [b].[ytd_sales]>[pubs].[dbo].[titles].[ytd_sales] as
[a].[ytd_sales]))
```

And the graphic execution plan for the ANSI SQL query is shown below in Figure 2:

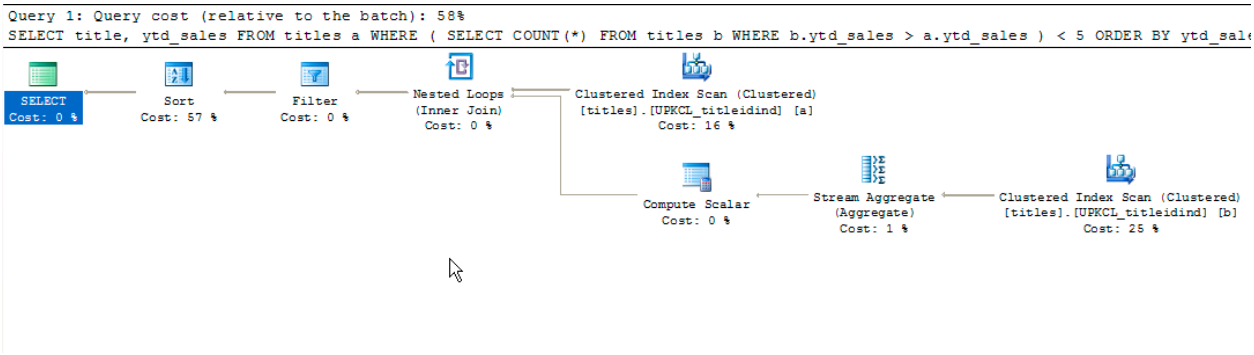



**Figure 2: A Graphic Execution Plan in SSMS**

The I/O statistics show that a significant number of reads is needed to complete this query:

```
Table 'titles'. Scan count 19, logical reads 38, physical reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
```

The pure ANSI SQL solution executes a correlated subquery which may be inefficient, especially in this case. That's because there is no index on ytd_sales to support the subquery since the query engine will have to traverse these values one or more times. Additionally, the pure ANSI SQL command does not filter out NULL values in ytd_sales, nor does it discriminate in the case of a tie between multiple titles.

Using TOP n:

```
SELECT TOP 5 title, ytd_sales
FROM     titles
ORDER BY ytd_sales DESC
```

The very simple textual execution plan for the TOP query looks like this:

```
StmtText
-----------------------------------------------------------------------------
  |--Sort(TOP 5, ORDER BY:([pubs].[dbo].[titles].[ytd_sales] DESC))
       |--Clustered Index Scan(OBJECT:([pubs].[dbo].[titles].[UPKCL_titleidind]))
```

And the graphic execution plan for the TOP query is shown in Figure 3 below:

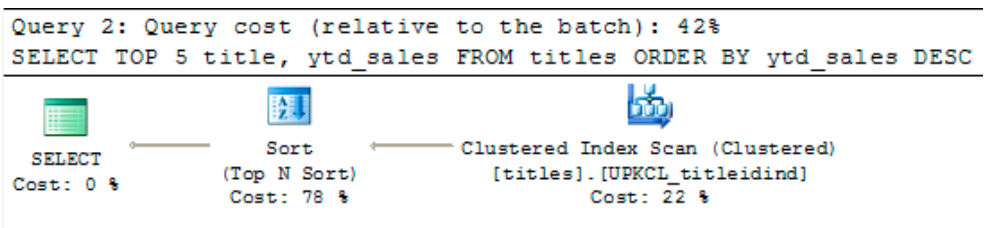



**Figure 3: A Graphic Execution Plan for a SELECT…TOP Statement**

The I/O statistics show that a very light load of reads, especially when compared to the earlier query, are needed to complete this one:

```
Table 'titles'. Scan count 1, logical reads 2, physical reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
```

The second solution, using TOP n to terminate the result set after it has found the first five rows, produces a dramatically simpler and more efficient execution plan. In this case, we also have an ORDER BY clause that forces sorting of the whole table before results may be retrieved. Both queries have almost identical execution plans.

To gain an even greater performance advantage on a large table, we would create an index on ytd_sales to avoid sorting. The query would then use the index SEEK to find the first five rows and stop, rather than the currently used clustered index SCAN looking at all rows in the table. In addition, compare the query plan of the second (TOP) query to the first solution (the ANSI query), which scans the whole table and executes a correlated subquery for every row retrieved in the outer query. The difference in performance is negligible on a small table. But on a large table, it might amount to hours of processing time for the first solution versus seconds for the second solution.

When determining the needs of your query, consider whether you only need to review a few of the rows retrieved. If that is the case, the TOP clause will be a valuable time saver.

# Let's All JOIN Hands and Sing: Understanding the Impact of Joins

If you read through the different query steps earlier in this paper, you saw how a large number of the operations are dedicated to explaining what happens with joins in SQL Server. Every join strategy has its strengths as well as its weaknesses. However, there are certain rare circumstances where the query engine chooses a less efficient join, usually using a hash or merge strategy when a simple nested loop offers better performance.

SQL Server uses three join strategies. They are listed here from the least to the most complex:

## Nested Loop

This is the best strategy for small tables with simple inner joins. It works best where one table has relatively few records compared to a second table with a fairly large number of records, and they are both indexed on the joined columns. Nested loop joins require the least I/O and the fewest comparisons.

A nested loop iterates through each record in the outer table once, and then searches the inner table for matches each time to produce the output. There are a lot of names for specific nested loop strategies. For example, a naïve nested loop join occurs when an entire table or index is searched. Other examples include an index nested loop join or a temporary index nested loop join when a regular index or temporary index is used.

## Merge

This is the best strategy for large, similarly-sized tables with sorted join columns. Merge operations sort and then cycle through all of the data to produce the output. Good merge join performance is based on having indexes on the proper set of columns, almost always the columns mentioned in the equality clause of the JOIN predicate.

Merge joins take advantage of the pre-existing sorts by taking a row from each input and performing a direct comparison. For example, inner joins return records where the join predicates are equal. If they aren't equal, the record with the lower value is discarded and the next record is picked up and compared. This process continues until all records have been examined. Sometimes merge joins are used to compare tables in many-to-many relationships. When that happens, SQL Server uses a temporary table to store rows.

If a WHERE clause also exists on a query using a merge join, then the merge join predicate is evaluated first. Then, any records that make it past the merge join predicate are then evaluated by the other predicates in the WHERE clause. Microsoft calls this a residual predicate.

## Hash

The best strategies for large, dissimilarly sized tables, or for complex join requirements where the join columns are not indexed or sorted is a hashing join. Hashing is used for UNION, INTERSECT, INNER, LEFT, RIGHT, and FULL OUTER JOIN, as well as set matching and difference operations. Hashing is also used for joining tables where no useful indexes exist. Hash operations build a temporary hashing table and then cycle through all of the data to produce the output.

A hash uses a build input (always the smaller table) and a probe input. The hash key (that is, the columns in the join predicate or sometimes in the GROUP BY list) is what the query uses to process the join. A residual predicate is any evaluation in the WHERE clause that does not apply to the join itself. Residual predicates are evaluated after the join predicates. There are several different options that SQL Server may choose when constructing a hash join, in order of precedence:

- **In-memory hash:** This join builds a temporary hash table in memory by first scanning the entire build input into memory. Each record is inserted into a hash bucket based on the hash value computed for the hash key. Next, the probe input is scanned record by record. Each probe input is compared to the corresponding hash bucket and, where a match is found, returned in the result set.
- **Grace hash:** The grace hash option is used when the hash join is too large to be processed in memory. In that case, the whole build input and probe input are read in. They are then pumped out into multiple, temporary work tables in a step called partitioning fan-out. The hash function on the hash keys ensures that all joining records are in the same pair of partitioned worktables. Partition fan-out basically chops two long steps into many small steps that can be processed concurrently. The hash join is then applied to each pair of work tables and any matches are returned in the result set.

- **Hybrid hash:** If the hash is only slightly larger than available memory, SQL Server may combine aspects of the in-memory hash join with the grace hash join in what is called a hybrid hash join.
- **Recursive Hash:** Sometimes the partitioned fan-out tables produced by the grace hash are still so large that they require further re-partitioning. This is called a recursive hash.

Remember that hash and merge joins process through each table just once. Therefore, they might have deceptively low I/O metrics, as does our example in the second query, if you use SET STATISTICS IO ON with queries of this type. However, the low I/O does not mean these join strategies are inherently faster than nested loop joins because of their enormous computational requirements and the fact that they must materialize a worktable in tempDB to complete the processing of the query.

Hash joins, in particular, are computationally expensive. If you find certain queries in a production application consistently using hash joins, this is your clue to tune the query or add indexes to the underlying tables.

In the following example, both a standard nested loop (using the default query plan) and hash and merge joins (forced through the use of hints) are shown:

```
SELECT a.au_fname, a.au_lname, t.title
FROM authors AS a
INNER JOIN titleauthor ta
   ON a.au_id = ta.au_id
INNER JOIN titles t
   ON t.title_id = ta.title_id
ORDER BY au_lname ASC, au_fname ASC

StmtText
------------------------------------------------------------------------------
  |--Nested Loops(Inner Join, OUTER REFERENCES:([ta].[title_id]))
       |--Nested Loops(Inner Join, OUTER REFERENCES:([a].[au_id]))
       |    |--Index Scan(OBJECT:([pubs].[dbo].[authors].[aunmind] AS [a]), ORDERED
FORWARD)
       |    |--Index Seek(OBJECT:([pubs].[dbo].[titleauthor].[auidind] AS [ta]),
            SEEK:([ta].[au_id]=[a].[au_id]) ORDERED FORWARD)
       |--Clustered Index Seek(OBJECT:([pubs].[dbo].[titles].[UPKCL_titleidind] AS
[t]),
          SEEK:([t].[title_id]=[ta].[title_id]) ORDERED FORWARD)
```

When examining the I/O of this query, we see that a nominal number of reads are performed and that no work table is created to process the result set:

```
Table 'titles'. Scan count 0, logical reads 50, physical reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
Table 'titleauthor'. Scan count 23, logical reads 46, physical reads 0, read-ahead
reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
Table 'authors'. Scan count 1, logical reads 2, physical reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
```

The visual plan of this query, which consumes the smaller amount of resources at 30 percent compared to the following query, is shown below in Figure 4:

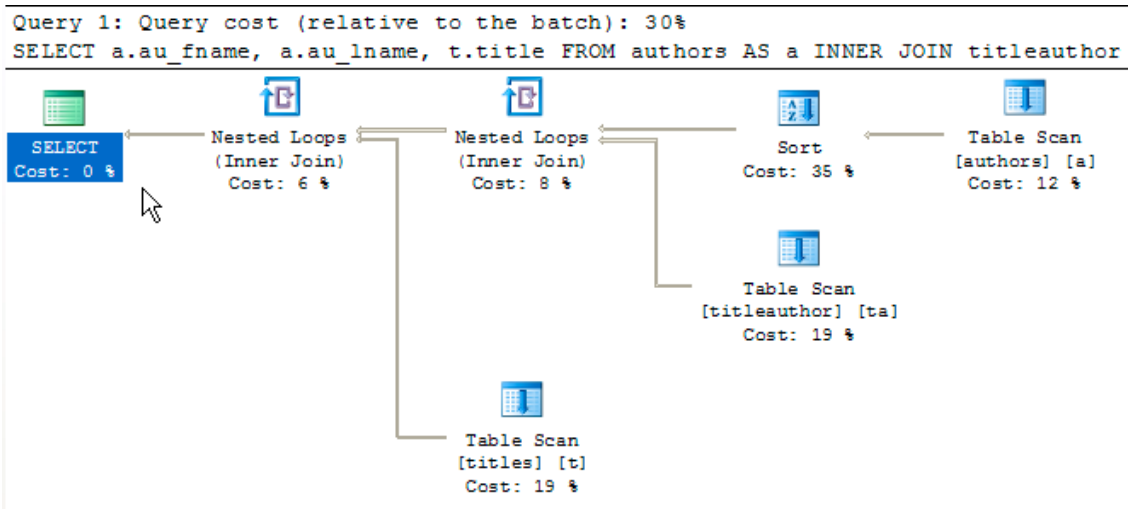



**Figure 4: The Graphic Execution Plan of a Query Using a Nested Loop Join Algorithm**

The showplan displayed above is the standard query plan produced by SQL Server. It can force SQL Server to show you how it would handle these as merge and hash joins using hints:

```
SELECT a.au_fname, a.au_lname, t.title
FROM authors AS a
INNER MERGE JOIN titleauthor ta
   ON a.au_id = ta.au_id
INNER HASH JOIN titles t
   ON t.title_id = ta.title_id
ORDER BY au_lname ASC, au_fname ASC

Warning: The join order has been enforced because a local join hint is used.

StmtText
-------------------------------------------------------------------------------
  |--Sort(ORDER BY:([a].[au_lname] ASC, [a].[au_fname] ASC))
       |--Hash Match(Inner Join, HASH:([ta].[title_id])=([t].[title_id]),
          RESIDUAL:([ta].[title_id]=[t].[title_id]))
            |--Merge Join(Inner Join, MERGE:([a].[au_id])=([ta].[au_id]),
               RESIDUAL:([ta].[au_id]=[a].[au_id]))
            |    |--Clustered Index
Scan(OBJECT:([pubs].[dbo].[authors].[UPKCL_auidind]
                  AS [a]), ORDERED FORWARD)
            |    |--Index Scan(OBJECT:([pubs].[dbo].[titleauthor].[auidind] AS [ta]),
                  ORDERED FORWARD)
            |--Index Scan(OBJECT:([pubs].[dbo].[titles].[titleind] AS [t]))
```

Due to the way that hash and merge joins process date, they often exhibit deceptively low I/O:

```
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
Table 'titles'. Scan count 1, logical reads 2, physical reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
Table 'titleauthor'. Scan count 1, logical reads 2, physical reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
Table 'authors'. Scan count 1, logical reads 2, physical reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
```

It's important to note that while the merge and hash joins have lower total scan counts and logical reads than the first query, they also require a worktable in TempDB to process the query. Since the first example—using nested loop joins—does not require a work table, it is able to execute the query with significantly less load.

The visual plan (which shows the hash/merge query, consuming 70 percent of the overall resources as compared to the earlier query's 30 percent), is shown below in Figure 5:

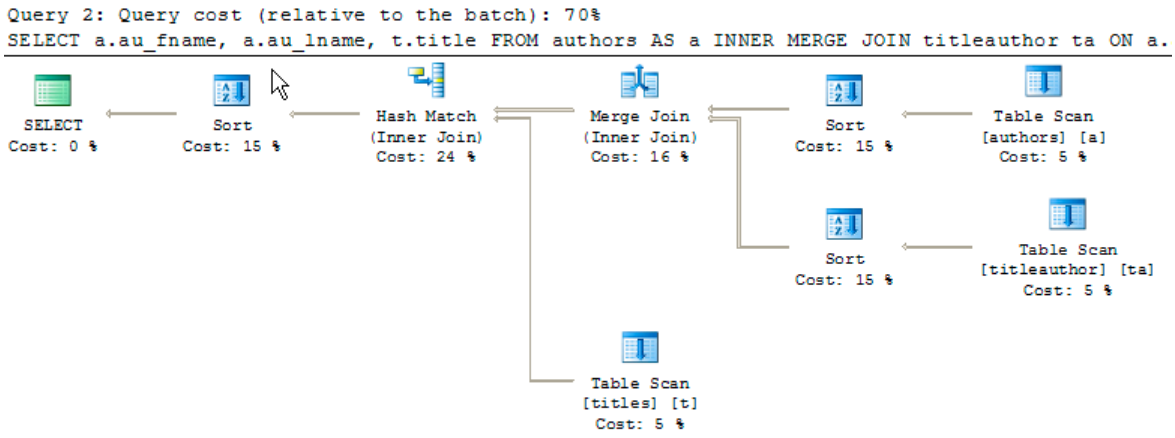



**Figure 5: A Graphic Execution Plan of a Query Using a Hash/Merge Join Algorithm**

In this example, you can clearly see that each join considers the join predicate of the other join to be a residual predicate. (You'll also note that the use of a hint caused SQL Server to issue a warning.) The second query was also forced to use a SORT operation to support the hash and merge joins.

If you'd like to discover more about how to optimize joins, here are a few articles and posts:

- http://sqlserverpedia.com/blog/sql-server-2005/no-join-predicate/
- http://sqlserverpedia.com/blog/sql-server-bloggers/does-order-matter-in-a-join-clause/
- http://sqlserverpedia.com/wiki/Optimizer_Hints

# SET NOCOUNT ON

You may have already noticed that, under normal circumstances, successful queries return a system message about the number of rows that they affect. In many cases you don't need this information, especially in procedure code (such as triggers, user-defined functions and stored procedures) that only returns information to the end user via PRINT and RAISERROR statements.

Command SET NOCOUNT ON allows you to suppress the message for all subsequent transactions in your session, until you issue the SET NOCOUNT OFF command. (Yes, it's a double negative, but T-SQL was not created by English majors after all.)

This option has more than a cosmetic effect on the output generated by your script. It reduces the amount of information passed from the server to the client by suppressing the DONE_IN_PROC background chatter. Therefore, it helps to lower network traffic and improves the overall response time of your transactions and procedural code. Time to pass a single message may be negligible, but think about a script that executes some queries in a loop and sends kilobytes of useless information to a user.

As an example, consider the following pseudocode to insert 9,999 rows into the sales table:

1. Create some variables
2. Create a loop
3. Assign new variables to an INSERT statement
4. Insert the data
5. Go to step two if loop counter is less than 10,000
6. Print the final running time of the procedural code

When run with SET NOCOUNT OFF, the elapsed time for the example code was 5,176 milliseconds. When run with SET NOCOUNT ON, the elapsed time for the procedural code was 1,620 milliseconds!

Note that the more Transact-SQL commands and/or iterations through the procedural code, the greater the benefit of the SET NOCOUNT ON statement. Consider adding SET NOCOUNT ON at the beginning of every stored procedure and script that doesn't require the "n ROWS AFFECTED" message in the output.

# QUERYING AGAINST COMPOSITE KEYS

Composite keys are problematic for SQL Server. Composite indexes are composed of several columns of a table. The problem is that composite indexes are used from the left-most column to right.

The following examples show that SQL Server 2008 now handles poorly ordered WHERE clauses much better than earlier versions of the product. In earlier versions of the product, SQL Server might've ignored indexes when all the columns of an index were addressed in the WHERE clause, solely because the columns were not referenced in the same order as they appeared in the index. This is no longer a problem in SQL Server 2008. However, the problem still impacts how SQL Server chooses execution plans and may cause the optimizer to choose less than optimal plans.

Consider this composite index that contains three columns:

```
CREATE INDEX my_ndx ON new_sales ([stor_id], [ord_num], [title_id] )
```

Depending on your WHERE clause conditions, SQL Server may use all or fewer columns of the index, or not use the index at all, as shown below:

**Table 1. Usage of Composite Key Columns**

| WHERE CLAUSE CONDITIONS | INDEX USE |
|---|---|
| WHERE stor_id = @a<br>AND ord_num = @b<br>AND title_id = @c | my_ndx SEEK |
| WHERE stor_id = @a<br>AND ord_num = @b | my_ndx SEEK |
| WHERE ord_num = @b<br>AND stor_id = @a | my_ndx SEEK, using the same execution plan as the query above |
| WHERE stor_id = @a | my_ndx SEEK |
| WHERE stor_id = @a<br>AND title_id = @c | my_ndx SEEK, using the same execution plan as the query above. This query is not able to use the third column of the index. |
| WHERE ord_num = @b | my_ndx SCAN |
| WHERE ord_num = @b<br>AND title_id = @c | my_ndx SCAN. |
| WHERE title_id = @c | my_ndx SCAN. |

The key point to remember is that you should know the order of columns appearing within a composite index. Once you know the order of the columns, you should always structure your WHERE clause to analyze columns starting with the left-most column in the composite index and working toward the right. If you build a WHERE clause that does not use leftmost column(s), the index will typically be ignored, resulting in a SCAN operation rather than a better-performing SEEK operation.

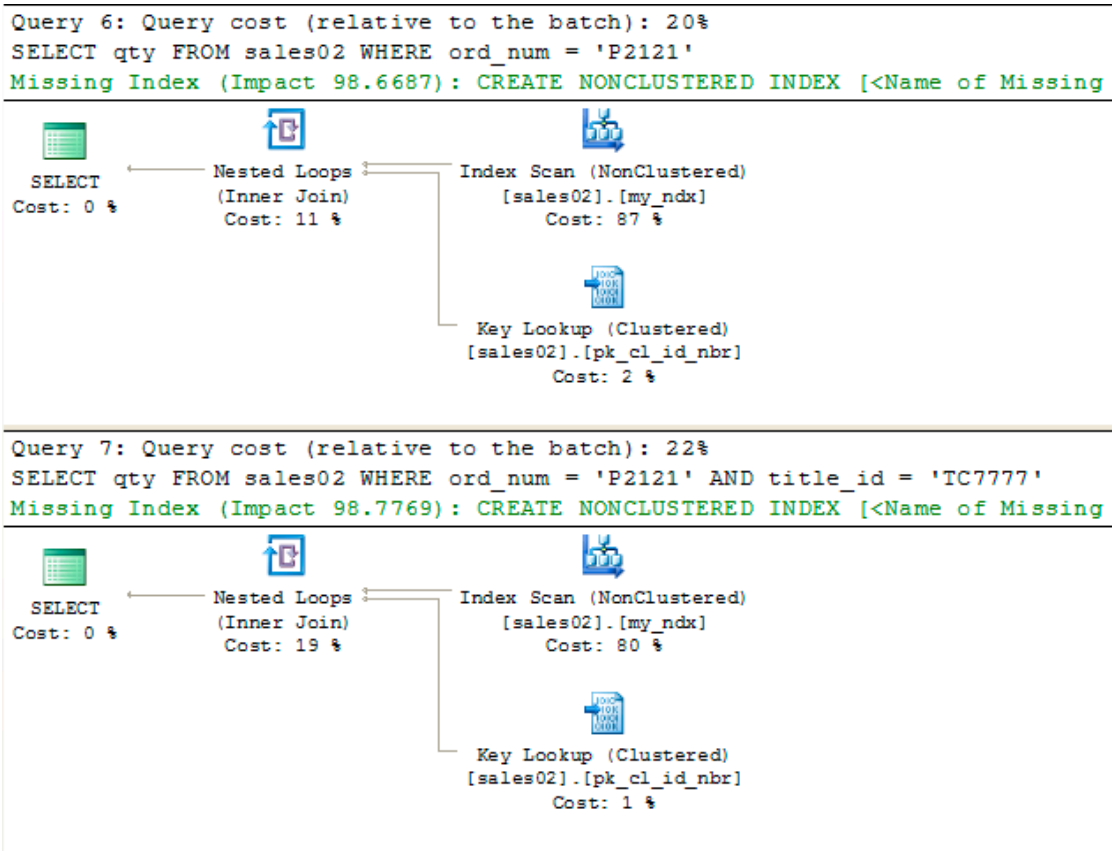



**Figure 6: Comparing Two Query Execution Plans, the Second Going Against a Compound Index**

As shown above, in Figure 6, when using SQL Server 2008 and the graphic execution plan features of SSMS, you get the added bonus of seeing recommended new indexes that could improve the performance of the query.

# SUMMARY

This white paper has presented a collection of tips and tricks to help you get the most out of your queries on a SQL Server 2008 database. Some ideas presented in the white paper include:

- How to use the SET STATISTICS and SET SHOWPLAN commands to see the resource consumption and execution plan of a query
- How to use the DBCC DROPCLEANBUFFERS and DBCC FREEPROCCACHE commands to clear the development server's buffer and procedure cache, using them as elements of your SQL testing harness
- Understanding the basics of reading execution plans, as illustrated by the difference between SEEK and SCAN operations, to determine which variant of a query performs the best
- A variety of scenarios that offer opportunities for performance improvement, such as
  - Functions and expressions that suppress indexes
  - Subquery optimizations versus joins
  - UNION versus UNION ALL
  - The advantages of UPDATE…FROM and DELETE…FROM over ANSI standard syntax
  - TOP and SET ROWCOUNT
  - Alternate join strategies and how they can impact query performance
  - What's so special about SET NOCOUNT ON and why you'd want to use it with procedural code
  - The impact of querying against concatenated keys

Add these tips and techniques to your SQL Server coding toolkit. Be sure to check on the I/O generated by your queries using SET STATISTICS I/O and to read the query execution plans with either SET SHOWPLAN or the graphic execution plans within SQL Server Management Studio. The bottom line is that you need to test, test and retest!

# About the Author

**Kevin Kline** is the technical strategy manager for SQL Server Solutions at Quest Software. A Microsoft SQL Server MVP since 2004, Kevin is a founding board member and past president of the international Professional Association for SQL Server (PASS). He has written or co-written several books, including *SQL in a Nutshell, Pro SQL Server 2008 Relational Database Design and Implementation*, and *Database Benchmarking: Practical Methods for Oracle & SQL Server*. He is a top rated speaker at conferences worldwide such as Microsoft TechEd, the PASS Community Summit, Microsoft IT Forum, DevTeach, and SQL Connections, and has been active in the IT industry since 1986. Kevin contributes to *SQL Server Magazine* and *Database Trends & Applications* and blogs SQLBlog.com and SQLMag.com.

# About Quest Software, Inc.

Now more than ever, organizations need to work smart and improve efficiency. Quest Software creates and supports smart systems management products—helping our customers solve everyday IT challenges faster and easier. Visit www.quest.com for more information.

## Contacting Quest Software

| | |
|---|---|
| Phone: | 949.754.8000 (United States and Canada) |
| Email: | info@quest.com |
| Mail: | Quest Software, Inc.<br>World Headquarters<br>5 Polaris Way<br>Aliso Viejo, CA 92656<br>USA |
| Web site: | www.quest.com |

Please refer to our Web site for regional and international office information.

## Contacting Quest Support

Quest Support is available to customers who have a trial version of a Quest product or who have purchased a commercial version and have a valid maintenance contract. Quest Support provides around the clock coverage with SupportLink, our web self-service. Visit SupportLink at http://support.quest.com

From SupportLink, you can do the following:

- Quickly find thousands of solutions (Knowledgebase articles/documents).
- Download patches and upgrades.
- Seek help from a Support engineer.
- Log and update your case, and check its status.

View the ***Global Support Guide*** for a detailed explanation of support programs, online services, contact information, and policy and procedures. The guide is available at: http://support.quest.com/pdfs/Global Support Guide.pdf