The White Papers

Microsoft T-SQL Performance Tuning

Part 1: Analyzing and Optimizing T-SQL Query Performance on Microsoft SQL Server using SET and DBCC

By Kevin Kline

QUEST SOFTWARE

Contents

QUEST SOFTWARE

Microsoft T-SQL Performance Tuning

Part 1: Analyzing and Optimizing T-SQL Query Performance on Microsoft SQL Server using SET and DBCC

By Kevin Kline

Introduction

This article is the first in a series that describes a variety of performance tuning techniques that you can apply to your Microsoft SQL Server Transact-SQL programs. In many cases, you can use the graphical user interface provided in Microsoft SQL Enterprise Manager or Microsoft SQL Query Analyzer to achieve the same or similar results. However, this series focuses on using Transact-SQL as the basis for our solutions. All examples and syntax are verified for Microsoft SQL Server 2000. Other articles in this series cover topics like:

1. Datatype tuning
2. Tuning through database and table partitioning
3. Indexing strategies
4. Query optimizer strategies
5. SHOWPLAN output and analysis
6. Optimizer hints and Join techniques
7. Query tuning tips and tricks

SQL Server provides you with capabilities to benchmark transactions by sampling I/O activity and elapsed execution time using certain *SET* and *DBCC* commands. In addition, some *DBCC* commands may be used to obtain a very detailed explanation of any index statistic, estimate the cost of every possible execution plan, and boost performance.

SET STATISTIC IO

The command *SET STATISTICS IO ON* forces SQL Server to report actual I/O activity on executed transactions. It cannot be paired with *SET NOEXEC ON* option, because it only makes sense to monitor I/O activity on commands that actually execute. Once the option is enabled every query produces additional output that contains I/O statistics. In order to disable the option, execute *SET STATISTICS IO OFF*. These commands also work on Sybase Adaptive Server, though some results sets may look somewhat different.

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, we execute a system stored procedure to obtain table size statistics for our analysis:

```
sp_spaceused syscomments
```

Results:

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

What can we tell by looking at this information?

- The query did not have to scan the whole table. The number of data in the table is more than 1.5 megabytes, yet it took only 53 logical I/O operations to obtain the result. It indicates that the query has found an index that could be used to compute the result, and scanning the index took fewer I/O than it would take to scan all data pages.
- Index pages were mostly found in data cache since the physical reads value is zero. This is because we executed the query shortly after other queries on *employees* and the table and its index were already cached. Your mileage may vary.
- Microsoft has reported no read-ahead activity. In this case data and index pages were already cached. For 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 and believes that it can predict which pages you'll need next. A separate SQL Server connection virtually runs ahead of your process and caches data pages for it. Configuration and tuning of read-ahead parameters is beyond the scope of this book.

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 that 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 our 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 what 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 (but don't try to compare your machine's performance to our notebook PC's, because this is not a representative benchmark). Moreover, 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 3 millisecond increments. It is impossible to get more granular time values than that using the *DATETIME* datatype.

SET NOCOUNT ON

This simple command is perhaps one of the single biggest performance boosts that you can add on to any code. At its most obvious level, *SET NOCOUNT ON* turns of the *N rows affected* verbiage that appears at the end of every query. More importantly, though, it also eliminates the *DONE_IN_PROC* internal messaging sent from the server to the client for each step in a stored procedure. Since many stored procedures need only return the messages that are explicitly put there by the programmer, this can provide an enormous performance boost when placed as the first command in a stored procedure, trigger or function. The syntax is simple:

```
SET NOCOUNT ON
```

Its behavior is disabled with the command *SET NOCOUNT OFF*.

DBCC

DBCC stands for *Database Consistency Check* and has many useful options. It is most famous (or infamous) for its use by DBAs to check consistency of a database and look for database corruption using commands like *DBCC CHECKDB, DBCC NEWALLOC, DBCC CHECKCATALOG*, and others. But *DBCC* has many additional options, and some of them may be useful in your T-SQL programming work.

DBCC SHOW_STATISTICS

This command is very useful in analyzing index effectiveness. You can use this command to tell whether the query optimizer can effectively use an index or not. It shows index statistics on a specified index of a particular table. The syntax is:

```
DBCC SHOW_STATISTICS ( table_name, index_name )
```

The output contains information about index density, which defines how many rows potentially have the same key. For composite keys, the output contains information about partial keys consisting of the first few columns of the index, as well as information for the whole key. The lower the value of the Density reading the better, since this indicates higher selectivity.

You can say that index density is the opposite of selectivity. Selective indexes have low density. Indexes with high density are not very selective and are unlikely to be used by the optimizer in query plans. Example:

```
USE northwind
GO
DBCC SHOW_STATISTICS ( [Order Details], OrderID )
GO
```

Results:

```
Statistics for INDEX 'OrderID'.
Updated              Rows  Rows Sampled Steps  Density      Average key length
-------------------- ----- ------------ ------ ------------ ------------------
Mar 26 2002  8:46AM  2155  2155         187    1.118621E-3  8.0
```

```
(1 row(s) affected)

All density               Average Length           Columns
------------------------- ------------------------ --------------------
1.2048193E-3              4.0                      OrderID
1.2048193E-3              8.0                      OrderID, ProductID

(2 row(s) affected)

RANGE_HI_KEY RANGE_ROWS EQ_ROWS  DISTINCT_RANGE_ROWS  AVG_RANGE_ROWS
------------ ---------- -------- -------------------- --------------
10248        0.0        3.0      0                    0.0
10253        11.0       3.0      4                    2.75
10256        7.0        2.0      2                    3.5
…
11070        10.0       4.0      5                    2.0
11075        9.0        3.0      4                    2.25
11076        0.0        3.0      0                    0.0
11077        0.0        25.0     0                    0.0

(187 row(s) affected)

DBCC execution completed. If DBCC printed error messages, contact your system administrator.
```

If we multiply the number shown in the *All density* column above by the total number of rows, we can determine how many rows have the same full key or partial key. In this example, a key consisting of values for columns *OrderID* and *ProductID* provides an excellent selectivity: 0.00120483 * 2155 = 2.5963855915. Selectivity of one means that any key defined by these columns is associated with only one row in the table – perfect selectivity. Other values returned by the *DBCC* command are useful as well:

Value Returned	**Description**
Updated	The date and time the index statistics were last updated.
Rows	The total number of rows in the table.
Rows Sampled	The number of rows sampled for index statistics information.
Steps	The number of distribution steps.
Density	The selectivity of the first index column prefix.
Average key length	The average length of the first index column prefix.
All density	The selectivity of a set of index column prefixes.
Average length	The average length of a set of index column prefixes.
Columns	The names of index column prefixes for which All density and Average length are displayed.
RANGE_HI_KEY	The upper bound value of a histogram step.
RANGE_ROWS	The number of rows from the sample that fall within a histogram step, not counting the upper bound.
EQ_ROWS	The number of rows from the sample that are equal in value to the upper bound of the histogram step.
DISTINCT_RANGE_ROWS	The number of distinct values within a histogram step, not counting

	the upper bound.
AVG_RANGE_ROWS	The average number of duplicate values within a histogram step, not counting the upper bound (where RANGE_ROWS / DISTINCT_RANGE_ROWS for DISTINCT_RANGE_ROWS > 0).

TABLE AND INDEX FRAGMENTATION

There are several commands available that can help you get a handle on table and index fragmentation: *DBCC SHOWCONTIG*, *DBCC INDEXDEFRAG*, *DBCC DBREINDEX*, and *CREATE/DROP INDEX*.

Table fragmentation is similar to hard disk fragmentation that occurs on any computer after weeks of creating, dropping, and modifying files. Database tables, just as disks, need defragmentation every so often in order to stay efficient. The most efficient allocation is when all pages occupy a contiguous area in the database, but after weeks of use, a table may become scattered across the disk drive. The more pieces it is broken into, the less efficient the table becomes.

The *DBCC SHOWCONTIG* command helps you decide when to recreate a clustered index. It provides information about table fragmentation and can also help you determine the right time to recreate clustered indexes, thereby reducing table fragmentation.

The syntax of this DBCC command is

```
DBCC SHOWCONTIG ( table [, index] )
```

You may use either table name and index name, or table ID and index ID numbers. For example:

```
USE northwind
GO
DBCC SHOWCONTIG ( [Order Details], OrderID )
GO
```

Results:

```
DBCC SHOWCONTIG scanning 'Order Details' table...
Table: 'Order Details' (325576198); index ID: 2, database ID: 6
LEAF level scan performed.
- Pages Scanned................................: 5
- Extents Scanned..............................: 2
- Extent Switches..............................: 1
- Avg. Pages per Extent........................: 2.5
- Scan Density [Best Count:Actual Count].......: 50.00% [1:2]
- Logical Scan Fragmentation ..................: 0.00%
- Extent Scan Fragmentation ...................: 50.00%
- Avg. Bytes Free per Page.....................: 2062.0
- Avg. Page Density (full).....................: 74.52%
DBCC execution completed. If DBCC printed error messages, contact your system administrator.
```

Statistics returned by the *DBCC* command are explained in the following table.

Statistics	Description
Pages Scanned	Number of database pages used by the table (when you specify indid of 1 or 0) or a non-clustered index (when you specify indid > 1).
Extent Switches	All pages of a table or an index are linked into a chain. Access to the table or index is more efficient when all pages of each extent are

	linked together into a segment of this chain. *DBCC* command scans the chain of pages and counts the number of times it has to switch between extents. If the number of extent switches exceeds the number of pages divided by 8, then there is a room for optimization.
Avg. Pages per Extent	Space for each table is reserved in extents of 8 pages. Some pages are unused, because the table has never grown to use them or because rows have been deleted from a page. The closer this number is to 8 – the better. A lower number indicates that there are many unused pages that decrease the performance of table access.
Scan Density [Best Count: Actual Count]	Scan Density shows how contiguous the table is. The closer the number is to 100% – the better. Anything less than 100% indicates fragmentation. Best Count shows the ideal number of extent switches that could be achieved on this table. Actual Count shows the actual number of extent switches.
Logical Scan Fragmentation	The Percentage of out-of-order pages returned from scanning the leaf pages of an index. This reading is not relevant for heaps (tables without indexes of any kind) and text indexes. A page is considered out of order when the next page in the Index Allocation Map (IAM) is different than the page indicated by the next page pointer in the leaf page.
Extent Scan Fragmentation	Percentage of out-of-order extents in scanning the leaf pages of an index, excluding heaps. An extent is considered out-of-order when the extent containing the current index page is not physically next after the extent holding the previous index page.
Avg. Bytes free per page	The average number of free bytes per page used by the table or index. The lower the number – the better. High numbers indicate inefficient space usage. The highest possible number of free space is 2014 – the size of a database page minus overhead. This or a close a number close to this will be displayed for empty tables. For tables with large rows this number may be relatively high even after optimization. For example, if row size is 1005 bytes, then only one row will fit per page. DBCC will report average free space also as 1005 bytes, but don't expect another row to fit into the same page. In order to fit a row of 1005 bytes you'd also need additional room for row system overhead.
Avg. Page density (full)	How full is an average page. Numbers close to 100% are better. This number is tied to the previous one and depends on the row size as well as on clustered index fill-factor. Transactions performed on table rows change this number because they delete, insert or move rows around by updating keys.

In order to defragment a table, drop and re-create a clustered index on it. Dropping and recreating the clustered index, will also recreate all the non-clustered indexes on a table. Another method available to defragment a table is found with the *DBCC INDEXDEFRAG* command. This command will reorder the leaf level pages of the index in a logical order just as dropping and recreating the clustered index will. However, *DBCC INDEXDEFRAG* offers some advantages. It is an online operation, keeping the index

and table available to other users while the command is running. It can also sustain an interruption without loss of the work already completed. Its disadvantage is that it does not do as good a job of reorganizing the data as a clustered index drop/re-create operation. The syntax for this command is:

```
DBCC INDEXDEFRAG
    ( { database | 0 } ,{ table | 'view' } ,{ index }
    )     [ WITH NO_INFOMSGS ]
```

When defining which database, table, view, or index you would like to defragment, you may use either the name of the object or its object ID. (When using a zero instead of the database name or database ID, the current database is assumed.) For example:

```
DBCC INDEXDEFRAG (Pubs, Authors, Aunmind)
GO
```

Results:

```
Pages Scanned Pages Moved Pages Removed
------------- ----------- -------------
359           346         8

(1 row(s) affected)

DBCC execution completed. If DBCC printed error messages, contact your system administrator.
```

Another command available to defragment tables and indexes is the *DBCC DBREINDEX*. Unlike *DBCC INDEXDEFRAG*, this command locks the affected table and renders it unavailable to all other users while the index is rebuilding. The syntax is:

```
DBCC DBREINDEX
    ( ['database.owner.table_name' [,index_name [,fillfactor] ] ]
    )     [ WITH NO_INFOMSGS ]
```

For example:

```
DBCC DBREINDEX ('pubs.dbo.authors')
```

If statistics are far from perfect, you can ask your DBA to rebuild clustered index on the table in order to freshen it up. Even on tables that do not have a clustered index can benefit by creating a dummy one and then dropping it.

Note that the operation may be very time-consuming on large tables and usually requires free space in the database equal to 1.25 times the total size of data pages used by the table. Unfortunately, it may not be feasible to rebuild indexes on large tables in critical production systems. You may often have to live with fragmented tables or other methods to defragment them.

DBCC SQLPERF

DBCC SQLPERF can be used to obtain general SQL Server performance statistics. In SQL Server 2000, only *DBCC SQLPERF LOGSPACE* is officially supported and maintained in the documentation. However, additional keywords allowed in earlier versions of SQL Server are still working. These same parameters can also be tracked in real time using the SQL Server Performance Monitor. The syntax of the command follows:

```
DBCC SQLPERF ( {IOSTATS | LRUSTATS | NETSTATS | RASTATS [, CLEAR]} | {THREADS} | {LOGSPACE} )
```

SQLPERF options and their results are explained below:

IOSTATS
Reports I/O usage since the server was started or since these statistics were cleared. The closer these values are to zero, the better. Example results:

```
Statistic           Value
------------------- ------
Reads Outstanding   0.0
Writes Outstanding  1.0
```

LRUSTATS
Reports cache usage since the server was started or since these statistics were cleared. LRU is Least Recently Used. Cache Hit Ratio is the single most important performance value in this group and indicates better results the closer it is to 100. (See *DBCC PROCCACHE* below for a similar command.) Example results:

```
Statistic                       Value
------------------------------- ------------------------
Cache Hit Ratio                 99.875603
Cache Flushes                   0.0
Free Page Scan (Avg)            0.0
Free Page Scan (Max)            0.0
Min Free Buffers                331.0
Cache Size                      4362.0
Free Buffers                    22.0
```

NETSTATS
Reports network usage. Example results:

```
Statistic                       Value
------------------------------- ------------------------
Network Reads                   243.0
Network Writes                  244.0
Network Bytes Read              38328.0
Network Bytes Written           88446.0
Command Queue Length            0.0
Max Command Queue Length        0.0
Worker Threads                  0.0
Max Worker Threads              0.0
Network Threads                 0.0
Max Network Threads             0.0
```

RASTATS
Reports Read Ahead usage. Example results:

```
Statistic                       Value
------------------------------- ------------------------
RA Pages Found in Cache         0.0
RA Pages Placed in Cache        0.0
RA Physical IO                  0.0
Used Slots                      0.0
```

CLEAR
This option is used in conjunction with one of the four discussed above. Clears the specified statistics and restarts generation of statistics. This option generates no output.

THREADS
Maps the Windows NT system thread ID to a SQL Server *spid*. The output is very similar to that of the system stored procedure *SP_WHO* and contains the login name, physical and logical (reported as CPU) I/O activity, and memory usage statistics. Example results:

```
Spid   Thread ID   Status     LoginName  IO                   CPU         MemUsage
------ ----------- ---------- ---------- -------------------- ----------- -----------
1      NULL        background NULL       0                    0           0
2      NULL        sleeping   NULL       0                    0           0
3      NULL        background NULL       0                    0           5
4      NULL        background NULL       0                    0           -6
5      0           background sa         6                    0           2
6      0           background sa         0                    0           2
7      NULL        sleeping   NULL       0                    0           0
8      0           background sa         0                    0           2
9      0           background sa         0                    0           2
10     0           background sa         0                    0           2
11     0           background sa         0                    0           2
12     0           background sa         0                    0           2
51     0           sleeping   CORPORATE\ 116                  40          124
52     2828        runnable   CORPORATE\ 276                  220         68
53     0           sleeping   CORPORATE\ 48                   200         102
```

LOGSPACE
Reports the percentage of transaction log space used. This option can only be used if transaction log is located on its own database segment. Example results:

```
Database Name  Log Size (MB) Log Space Used (%) Status
-------------- ------------- ------------------ -----------
master         0.4921875     64.186508          0
tempdb         0.4921875     51.578388          0
model          0.4921875     45.436508          0
msdb           2.2421875     31.685539          0
pubs           1.7421875     43.049328          0
Northwind      0.9921875     41.235928          0
```

In most cases, we recommend that you use SQL Performance Monitor to collect this information. It is easier to log-on to a file and provides visual comparisons of performance over time. SQL Performance Monitor also provides an explanation for each performance monitor result. However, if you are interested in a snapshot of a particular performance issue, then *DBCC SQLPERF* may come handy.

DBCC PROCCACHE

In earlier versions of SQL Server, the data cache and procedure cache could be tuned separately. The data cache was that area of the memory cache where data pages were stored for quick access while the procedure cache was used to hold query plans and compiled stored procedures execution plans. In SQL Server 2000, you cannot tune these areas of the memory cache separately since SQL Server now handles all aspects of this automatically. However, you can check up on the state of affairs in the procedure cache using the command *DBCC PROCCACHE*.

The syntax is:

```
DBCC PROCCACHE
```

The results are returned in a long string with values explained in the table below:

Column Name	Description
num proc buffs	The number of stored procedures that could possibly be in the procedure cache.
num proc buffs used	The number of slots in the cache holding stored procedures.
num proc buffs active	The number of slots in the cache holding stored procedures that are executing.
proc cache size	The total size of the procedure cache.
proc cache used	The amount of the procedure cache holding stored procedures.
proc cache active	The amount of the procedure cache holding stored procedures that are executing.

HINT: If you are anxious to find every last detail of server performance, SQL Server 2000 still returns valid data for the command *DBCC MEMUSAGE* even though it claims not to support the command any more. This command tells you the top 20 space consuming objects in the memory cache. Of course, it is highly advisable not to code any solutions around *DBCC MEMUSAGE* since it might not be around tomorrow, but you might have fun playing with the command.

DBCC PINTABLE

Retrieving data from cache significantly increases performance rather than retrieving data from disk. The speed of memory cache I/O is based on the lightning fast speed of electrons while the speed of disk is limited to the actual movement of gears and electronic motors. You have the option of forcing SQL Server to pin a table in memory, ensuring that its pages are not flushed from memory. Syntax:

```
DBCC PINTABLE(database_id, table_id)
```

This command requires that you use the ID of the database and table. If you don't happen to know the database and/or table ID (which I seldom do), you can use the *DB_ID* function and *OBJECT_ID* function, respectively, in a query to find out the values. The example below shows this usage:

```
DECLARE @db INT, @obj INT
SELECT @db  = DB_ID('northwind'),
       @obj = OBJECT_ID('northwind..employees')
DBCC PINTABLE (@db, @obj)
```

This command should not be used lightly though, as the results point out:

```
Warning: Pinning tables should be carefully considered. If a pinned table is larger, or grows
larger, than the available data cache, the server may need to be restarted and the table
unpinned.

DBCC execution completed. If DBCC printed error messages, contact your system administrator.
```

The command does not actually read a table into the memory cache. Instead, it ensures that pages from the table are retained in cache once read. (You could couple this command immediately with a *SELECT* statement to read the table directly into memory.) This command has a dark side – it does not flush the pages of a pinned table, even when it needs the space in the memory cache very badly. This can be especially bad if the pinned table is very large. Consequently, this command should be used only on small, frequently used tables such as lookup tables. Use the command *DBCC UNPINTABLE* to release the pages from memory cache.

Summary

In this article, we discussed various T-SQL discovery, optimization and tuning techniques using the *SET* and *DBCC* commands. SQL Server performance is a composite of many factors. And while T-SQL programming is only one of aspect, it is a very important one. In this article, we focused on the following commands:

- SET STATISTICS IO
- SET STATISTICS TIME
- SET NOCOUNT ON
- DBCC SHOW_STATISTICS
- DBCC SHOWCONTIG
- DBCC INDEXDEFRAG
- DBCC DBREINDEX
- DBCC SQLPERF
- DBCC PROCCACHE
- DBCC PINTABLE

Future articles in this series will discuss database design tips, indexing strategies, analyzing the query plan of queries, optimizer hints, and cool programming tips and tricks that provide the added oomph to T-SQL programs.

About the Author

Kevin Kline serves as Senior Product Architect for SQL Server at Quest Software designing products for DBAs and database developers. Kevin is author of four books, including the very popular "SQL in a Nutshell" and "Transact-SQL Programming" both published by O'Reilly & Associates (www.oreilly.com), and numerous magazine and on-line articles. Kevin is also active in the SQL Server community, serving as Executive Vice President of the Professional Association for SQL Server (www.sqlpass.org). When he's not spending time on database technology, Kevin enjoys spending time with his wife & four children, practicing classical guitar (very badly), and gardening.

About Quest Software

Quest Software, Inc. is a leading provider of application management solutions. Our products increase the performance and uptime of business-critical applications while driving down the total cost associated with running those applications. By focusing on both the people and technology that make applications run, Quest Software enables IT professionals to achieve more with fewer resources. Quest Software is based in Irvine, California and has offices around the globe. For more information, visit www.quest.com.

Material adapted from "Transact-SQL Programming" (O'Reilly & Associates, ISBN: 1565924010) by Kevin Kline, Lee Gould, and Andrew Zanevsky, http://www.oreilly.com/catalog/wintrnssql/.