Microsoft T-SQL Performance Tuning
Part 2: Index Tuning Strategies

Adapted from "Transact-SQL Programming" By Kevin Kline, Andrew Zanevsky, and Lee Gould. Published by O'Reilly & Associates. ISBN: 1565924010

By Kevin Kline, Senior Product Architect for SQL Server

QUEST SOFTWARE

Contents

QUEST SOFTWARE

Microsoft T-SQL Performance Tuning
Part 2: Index Tuning Strategies

By Kevin Kline

Introduction

This article is the second 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 could use the graphic user interface provided in Microsoft SQL Enterprise Manager or Microsoft SQL Query Analyzer to achieve the same or similar results to those described here. 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 & tricks

Overview

Creating proper indexes is probably the single most critical aspect of T-SQL performance. Adding a new index can very often reduce response time of a query from hours to seconds. This article is all about choosing the best indexes and then analyzing their effectiveness. The SQL Server Optimizer can analyze and choose indexes for you, but it can only choose from existing ones. It is difficult to overstate how important good indexes are to query performance. Spend extra time analyzing and creating best indexes for any query that you expect to show good performance. There is another side of this coin. Poorly chosen indexes result in poor response time and immense aggravation of the same managers and users.

> In some companies, creation of indexes is a DBA's prerogative. But programmers are the ones who write SQL code that uses them. Very often they have a better idea of what index they need for a particular transaction, while a DBA has a general view of the database and SQL Server as a whole. Cooperation between both groups is the key to success.

What This Article Does NOT Cover

A full discussion on indexes could take up an entire chapter in a book. This article assumes that you know basically what an index is and how it functions. Attention is given to the thought process involved in choosing the best indexes, knowing when to add indexes, and knowing when to drop them.

For brevity's sake, this article avoids certain aspects of indexes including:

- Basic structure and architecture of indexes and tables, including pages and extents
- Basic overview of creating, modifying, and deleting indexes
- Creating, modifying, renaming, or dropping indexes
- The affect of text, ntext, and image datatypes upon indexes
- Locking algorithms and isolation levels
- Full-text indexes
- The role of the Sysindexes system table

Selecting Indexes: Rules of Thumb

There are a few rules that make easy work of a first guess when selecting columns for an index.

The rules of thumb, in order of priority, are:

1. Frequency that data is retrieved from a table based upon the values a specific column. The more often data is retrieved based on the values of a column within a table, the greater the need for an index on that column. The data may be specifically required, such as a social security number, or found in a range of values, such as order date.
2. Whether the column is used to build joins with other tables. Joins are almost always guaranteed better performance when the join columns in both tables are indexed. For example, foreign key columns should have an index associated with them.
3. When the data in the column is usually needed in the same order every time. Clustered indexes physically arrange the rows of a table on the disk in either ascending or descending order. Thus, clustered indexes can greatly speed queries using the *ORDER BY* clause if they are frequently needed in the exact order of the clustered index.
4. When the values in the index leaf node can answer the query without going to the data node (refer to clustered indexes later in this article).
5. When duplicate values need to be avoided. Using a primary key or unique key constraint also establishes an index on the columns that prevent any duplicate values from being entered into the table.
6. When columns have a large number of distinct values. Do not index columns that contain relatively few distinct values, such as a "Gender" column.
7. Columns that help group the rows of a table into distinct categories.

8. Tables that have many rows. Tables that have relatively few rows, for example, a look-up table with a code for every state within the U.S., might take as long to scan the entire table as an extra index. It's often better to allow the whole look-up table to be cached and not bother with the index.

Index Structure and the Importance of Fill Factor

A quick description of the structure of tables is in order. SQL Server uses the Btree structure to organize indexes. Btrees have a root node, intermediate nodes, and leaf nodes. (Nodes are interchangeably called pages.) Btrees are also really fast to traverse.

The data of a clustered index is always stored on the leaf nodes. The leaf node is where the physical data is actually written on the disk. Non-clustered indexes store pointer keys on intermediate and root nodes, without storing the actual physical data of the table itself.

When creating a clustered index, SQL Server default behavior allows each data and index page to be filled 100% full. This strategy conserves space and makes reads faster because all the data is contiguous. However, this strategy also means that inserting any new data into a completely full data or index page will result in a page split. Page splits are an expensive operation that injects free space into a table as new data is inserted. Page splits move about half of the rows of the current data or index page into a whole new page so that there will be enough room for the new row. Page splits also contribute to table fragmentation because the new page that is created from the page split may be added to the end of the table. Thus, breaking the contiguous data into two (or more) non-continuous pages.

SQL Server allows you to minimize the performance hit of page splits by specifying a *fill factor* for the clustered index of your table. Fill factor is a gap placed on each data page when a clustered index is created. The gap is a percentile value describing how much data should be on the data page. Thus, a value of 100% means that the data page will be 100% full of data, while a value of 80% means that 20% of the data page will be free for new inserts. You can even ensure free space in the intermediate and index pages of a table by using the *pad index* option of the *CREATE INDEX* statement or via the SQL Enterprise Manager GUI. (You can also set a system-wide value for fill factor using the *sp_configure* system stored procedure).

> Fill factor and pad index only on tables are useful for OLTP applications with many reads, writes, and updates. A good rule of thumb value on such tables is 80. However, fill factor can actually *decrease* the performance of tables that are primarily read-only. So leave fill factor at 100 on tables used primarily for read-only.

Fill factor is only established when the index is created. SQL Server doesn't do anything to maintain the fill factor. Therefore, it is a good idea to schedule rebuilds on indexes on a regular basis where fill factors need to be maintained. Refer to the section on Index Maintenance later in this white paper.

Clustered vs. Non-Clustered Indexes

A clustered index defines the actual physical order of the rows written to disk. If you define a table with a clustered index based on ascending social security numbers, then each new record written to the table will be in ascending according to the value in each social security number field. The clustered index actually is the data of the table. On the other hand, non-clustered indexes serve as quick look-ups to the actual physical records, much like the index of a book or magazine. Non-clustered indexes contain only pointers to the rows of the table and have no bearing on the actual physical order of records in a table.

The SQL Server optimizer dearly loves clustered indexes. They are more efficient for many types of queries than an identical non-clustered index. Only one clustered index is allowed per table though you can have up to 249 non-clustered indexes. When choosing indexes, keep in mind that they should always be chosen so that the most important queries are optimized. For example, if you have two important queries on the same table using different keys in the *ORDER BY* clause, you have to choose which one is more critical as your clustered index. You can create a clustered index to optimize one report and a non-clustered index for another. However, it is likely that the query answered by the non-clustered index will not see as much performance boost as that answered by the clustered index. In such conflict situations you have to consider carefully which query is more important to optimize first.

It is important to have a clustered index on every table. They help to manage database space more efficiently and enable the use of fill factor (discussed later). Tables without a clustered index, called a heap, have numerous problems. For example, all scans against a heap is a full table scan. Also, heaps have a tendency to expand over time and use more space for the same number of rows compared to non-clustered indexes. With heaps, all new rows are added to the bottom of the table (the last data page of the table). Compare this to tables with a clustered index, where new rows may be inserted throughout the table and added to partially used pages.

Clustered indexes are good for queries that use:

- GROUP BY that use all or the first few columns of the clustered index key,
- ORDER BY that use all or the first few columns of the clustered index key,
- WHERE clause conditions comparing to the first or the first few columns of the clustered index key and retrieving many rows,
- Long keys (either based on long columns or composite keys comprised of many columns), because a clustered index on a long key takes no extra space for the leaf level. A non-clustered index on a long key may be quite large, because it takes a lot of space to store keys of the leaf level of the index.

The first three types benefit from the fact that requested rows are located continuously on the table. SQL Server only has to find the first qualifying row and then keep on scanning until all rows are done. Several rows found on a single page reduce the number of I/O operations needed to access all the data. Additionally, Microsoft SQL Server has a performance booster - the read-ahead feature that automatically detects sequential reads from the same table and pre-

fetches data pages before the query asks for them. With a non-clustered index it usually turns out that requested rows are scattered around different pages and it may take additional I/O operation. Therefore, a query using a clustered index is much more likely to take advantage of read-ahead.

Some factors to juggle when considering where to put a clustered index:

- Columns with monotonously increasing values, such as a column with the *IDENTITY* property or *TIMESTAMP* columns, can be dangerous. They create a "hot spot" of activity on the last page. Good for certain types of OLTP applications because it minimizes page splits. But can also be bad for certain locking situations.
- Build the clustered key around the most commonly sorted columns, such as those most commonly used in your GROUP BY and ORDER BY clauses.

Non-clustered indexes are good for accessing one or a few rows. When you retrieve one row you usually cannot take advantage of multiple rows located on the same page. They are also efficient for queries that can find all columns necessary to produce results in an index. In such cases SQL Server doesn't even look at data pages. (Refer to the section on Covering Indexes for more information.) Most non-clustered indexes have keys narrower than the underlying table. They consume considerably less space than the tables they belong to. Therefore, it takes fewer I/O operations to scan the index than the table giving the whole transaction has a better response time.

Clustered indexes usually have one less B-tree level on top of the leaf data pages than non-clustered ones. In such cases it takes one extra I/O to access a row via a non-clustered index compared to a clustered one. But the benefit may be negligible when index pages are found in cache since the extra I/O is an inexpensive logical read against data stored in the cache.

More vs. Fewer Indexes

Realize that more indexes are not always better. Every new index on a table may improve *some* of the *SELECT* statements, but may also slow down *other INSERT*, *UPDATE*, and *DELETE* transactions on the same table. This is because SQL Server has to update every index key for each affected row in the transaction. Simply put, the more indexes, the faster the *SELECT* statements. The more indexes, the greater the tax on *INSERT*, *UPDATE*, and *DELETE* transactions.

Indexes require special maintenance on tables with volatile data. A later section, Index Maintenance, discusses some of the aspects for keeping indexes at peak performance.

Decision support applications and reporting databases typically require more indexes per table in order to satisfy the most frequently run queries. Their users perceive good performance based on query response time. Conversely, they usually tolerate slower modification operations because those are limited to during non-peak hour batch jobs. On the other hand are

OLTP applications that are very sensitive to slower *INSERT, UPDATE*, and *DELETE* transactions, but might tolerate a tick or two longer when processing a report.

But with all that duly noted, new indexes usually improve the overall performance. For example, consider an application, where addition of a new index could significantly improve a rarely executed report query, at the expense of marginally slowing down an occasionally executed UPDATE transaction.

Short vs. Long Key Indexes

This element of your indexing strategy boils down to the question of "one column in the index" or "two or more columns in the index"? Indexes with short keys, especially those with a single column, are generally more efficient than those with long keys. It is a matter of simple I/O, because more keys fit on an 8K page thus reducing the total number of I/Os needed to answer a request. Of course, columns comprising an index key should be chosen based on application requirements and the underlying data model. (A covering index is the only exception to this rule).

Very often you have a choice of adding an extra column to the key that doesn't make a big difference in index selectivity. It may be more efficient to make the index "leaner and meaner" rather than use long composite keys, even though they may be more *selective*. A longer key means that fewer keys fit on a single database page and the index may require more levels in the B-tree. As a result, it could take more I/O operations to get the same data through a wider index thus slowing down the transactions.

Covering Indexes

Sometimes it pays to create an index with a long composite key that includes all columns needed to process a certain critical query. A non-clustered index that includes (or *covers*) all columns used in a specific query is called a *covering index*. The values of the covering index are stored on their own leaf nodes. When SQL Server requests data using the covering index, the query can be resolved simply by scanning the index without having to scan the actual table and its data pages. This can result in dramatically fewer I/Os for the query, especially on large tables.

For example, suppose that your application allows a search on the employee last name:

```
SELECT lname FROM employee WHERE lname LIKE '%ttlieb%'
```

This query does a full table scan on the *employee* table. But if we create the following covering index the query becomes more efficient:

```
CREATE INDEX employee_lname ON employee ( lname )
```

Using the default PUBS database, the table is small and occupies only two pages. But the index *employee_lname* is even more compact – it takes only one page. Therefore, with the help of the

covering index the same query requires only one physical I/O operation instead of two. If the table possessed thousands of rows, the improvement in I/O would be factored in the hundreds.

In a multi-column key, the covering index has to include all columns in the query, as in the following example:

```
SELECT  fname, lname
FROM    employee
WHERE   lname LIKE '%ttlieb%'
    AND job_lvl < 200
```

A covering index for this query has to include three columns:

```
CREATE INDEX employee_name_lvl ON employee ( lname, fname, job_lvl )
```

Although covering indexes are a great way to speed specific queries, it is important not to overdo your indexes. If a covering index benefits only a single query, but takes a toll on every *INSERT, UPDATE*, or *DELETE* transaction on the same table, then it may not be a good idea to create it. Making covering index keys too wide reduces their benefits as well, because an index on a wide key takes more space and may not be much more effective than scanning data pages of the table.

Helping SQL Server Choose Indexes

There are a number of tricks and tips you can apply to help SQL Server make your queries perform at their absolute peak. The mere existence of an index is not enough for SQL Server to make best use of it. Generally, you have to specify conditions in the *WHERE* clause that may take advantage of an indexes. This section lists several important tips and tricks that can turn a ho-hum query into a high-performance query.

Index Selection

Using the right indexes is the most important aspect of optimization. The query optimizer selects the best indexes to:

- Evaluate filter conditions on each table
- Satisfy join conditions
- Find column values without going to the actual data pages (covering index)

Filter conditions are also known as *search arguments* or *SARG*. A SARG compares table columns to a constant, a variable, or an expression of constants and variables. The optimizer may find an index based on the SARG criteria that significantly narrow down the number of table rows qualifying for the result set.

Querying against Composite Keys

Composite indexes are composed of several columns of a table. One of the most common examples of a composite index is an index built on last_name and first_name. Composite

indexes are used from leftmost column to right. Thus, a query against a composite index containing last_name and first_name will only use the index if the *WHERE* clause filters on last_name and first_name, or only last_name (being the leftmost element of the composite index).

Let's look at an example composite index that contains four columns (a, b, c, and d):

```
CREATE NONCLUSTERED INDEX ndx_foo ON foo(a, b, c, d)
GO
```

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	Key Columns That May Be Used

`WHERE a = @a`
`  AND b = @b`
`  AND c = @c`
`  AND d = @d`
	a, b, c, d

`WHERE a = @a`
`  AND b = @b`
`  AND c = @c`
	a, b, c

`WHERE a = @a`
`  AND b = @b`
`  AND d = @d`
	a, b

`WHERE a = @a`
`  AND c = @c`
`  AND d = @d`
	a

`WHERE b = @b`
`  AND c = @c`
`  AND d = @d`
	Index cannot be used because the *WHERE* clause does not start with the leftmost column.

`WHERE b = @b` `  AND a = @a`	Again, the index cannot be used, despite both columns being indexed, because the WHERE clause does not analyze the leftmost column first.

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 leftmost column in the composite index and work towards the right.

Join Order

Queries that involve joining two or more tables may be performed in a multitude of ways. The more tables you join, the more possible join orders SQL Server may come up with. The number of possible join order permutations is the factorial of the number of tables involved:

Table 2. Number of Join Orders

Number of joined tables	Number of possible join orders
1	1
2	2
3	6 There are six orders for three tables (shown as table 1, 2, and 3): 123, 132, 213, 231, 312, 321.
4	24 There are 24 ways to join four tables (shown as 1, 2, 3, and 4): 1234, 1243, 1324, 1342, 1423, 1432, 2134, 2143, 2314, 2341, 2413, 2431, 3124, 3142, 3214, 3241, 3412, 3421, 4123, 4132, 4213, 4231, 4312, and 4321.
5	120
6	720
7	5,040
8	40,320
9	362,880
10	3,628,800
11	39,916,800
12	479,001,600
13	6,227,020,800
14	87,178,291,200
15	1,307,674,368,000
16	20,922,789,888,000 Even if you could evaluate a million different join orders per second, it would still take 242 days to find the best plan for a 16-way join!

Evaluating every plan involves considering different indexes on each table and the possibility of creating worktables to improve the performance. This means that the number of conceivable query execution plans is even greater than the number of join orders.

Past versions of Microsoft SQL Server did not handle joins of more than four tables well. And it is still a recommended best practice that you do not join more than five tables, even with the best hardware and latest editions of the software running in your environment. Because SQL Server analyzes only two tables at a time, you could break the queries into several steps. Microsoft no longer limits queries to any specific number of tables, provided that you do not exceed 16 tables. (Refer to the section on Transitive Properties for a hit around this problem.)

Transitive Properties

An interesting little tuning tip you can use to get better query performance is based on the old algebra concept called Transitive Properties. In this concept, we were taught that if A = B and B = C, then it is also true to say A = C. This is a fundamental concept that hardly registers on the conscious level for most of us. However, SQL Server does not know this. Therefore, you can get a big performance boost in multi-table queries or multi-join queries by including the extra step of checking for A = C.

For example, a query in a popular CRM application checked a customer's order against the product catalog and the product-shipping registry. It looked something like this:

```
SELECT c.customer_id
FROM   customer_order c,
       product p,
       shipping_registry s
WHERE  c.product_id = p.product_id
  AND  c.product_id = s.product_id
```

The performance on this query was only mediocre. However, adding one more option for comparison gave the query optimizer the choices it needed to turn a long-running query into a couple seconds query:

```
SELECT c.customer_id
FROM   customer_order c,
       product p,
       shipping_registry s
WHERE  c.product_id = p.product_id
  AND  c.product_id = s.product_id
  AND  p.product_id = s.product_id
```

Naturally, you should ensure that indexes exist on all the columns being evaluated!

Queries that include OR statements

Queries that include an *OR* statement pose particular performance problems to the SQL Server query engine. The *OR* condition is effectively the same as two queries with a *UNION* statement because any of the conditions maybe true to complete the query. The query will find each result set or scan for the whole result set. That means you will only get good performance from queries that contain *OR* statements in the *WHERE* clause when each and every conditions in the *OR* clause are indexed. This can definitely lead to poor performance!

As a workaround, you should evaluate the query as a *UNION* with each *SELECT* in the *UNION* using its own highly selective SARGs. This alternative approach will enable you to ensure that at least some portion of the query performs index seeks rather than full table scans.

Queries that use LIKE and Wildcards

Wildcards are a quick way to grab a lot of data without analyzing the query too much. After all, there are times when it's just easier to get the whole load, than to take all the time needed to figure out precisely what you need. However, you should not use wildcard queries in your

production systems! Avoiding wildcards in the *SELECT* list (as in, `SELECT * FROM foo`) offers several advantages:

1. Reduces network activity since unneeded columns are not returned to the client
2. Improves self-documentation of the code, since unacquainted coders can more easily discern the important data in the query
3. Alleviates the need for SQL Server to rebuild the query plan on procedures, views, triggers, functions or even frequently run queries any time the underlying table structure is changed
4. Narrows the query engine to the most pertinent index choices

Wildcards in a *LIKE* clause can be dangerous. There are two simple rules to remember when evaluating how useful a *LIKE* statement is to the performance of a query:

- *LIKE* can benefit from indexes if the pattern starts with a character string, such as `WHERE lname LIKE 'w%'`
- *LIKE* cannot use an index if the pattern starts with a wildcard, such as `WHERE lname LIKE '%alton'`

Thus, a query searching for records with a lname value of 'Walton' would access the table's index using the first *WHERE* clause, but would not access the index using the second *WHERE* clause.

Queries using Functions and Calculations in the WHERE clause

There are a certain number of predictable and controllable situations where an index will be disregarded, despite referencing it in the *WHERE* clause of your query, primarily the use of functions and calculations. Learning these simple rules of thumb will enable you to ensure your queries perform at their peak.

First, the use of calculations against a column in the *WHERE* clause of a query invalidates any index on that column. For example:

```
SELECT ord_num
FROM sales
WHERE qty * 12 > 10000
```

This query MUST do a full table scan to deliver the result set, even if there is an index on the sales.qty column.

```
SELECT ord_num
FROM sales
WHERE qty > 10000/12
```

The query above performs a seek on the index created on sales.qty!

Second, the use of functions against a column in the *WHERE* clause of a query invalidates any index on that column. For example:

```
SELECT ord_num
FROM sales
WHERE ISNULL(ord_date, 'Jan 01,2001') > 'Jan 01, 2002 12:00:00 AM'
```

This query MUST do a full table scan to deliver the result set, even if there is an index on the sales.ord_date column.

```
SELECT ord_num
FROM sales
WHERE ord_date IS NOT NULL
  AND ord_date > 'Jan 01, 2002 12:00:00 AM'
```

The query above performs a seek on the index created on sales.ord_date!

Index Tuning Wizard

The importance of the Index Tuning Wizard, even for experienced DBAs and developers, cannot be overstated. In a nutshell, the Index Tuning Wizard selects and creates indexes and statistics without requiring a deep understanding of the database tables, their workload, or SQL Server internals.

> The Index Tuning Wizard can consume significant CPU resources on the server where it is run so you might want to: A) avoid running it on production servers, B) run it on a separate computer, C) run it on small subsets of the tables in the database, and/or D) disable the **Perform thorough analysis** option.

The true power of the Index Tuning Wizard is brought to bear by processing a SQL Profiler trace (or SQL script) through the wizard. (If you're unfamiliar with SQL Profiler, you can use the **Sample 1 – TSQL** trace definition template to capture all the information you'll need in a file for the wizard.) The most valuable information you can provide for the wizard is a day (or at least an hour or two) of normal database activity. (Alternately, you can use the wizard to provide tuning recommendations for a specific query or set of queries.) The wizard then analyzes the workload and recommends an index configuration that will improve the performance of your database. The wizard can even return recommendations including important advanced considerations like disk space constraints.

The recommendation returned by the wizard are SQL statements that can be run directly in the SQL Query Analyzer to create the new and more effective indexes in your database. You can alternately schedule the SQL script as a job to execute later or save it to a file for manual execution. The recommendations returned by the wizard are based upon the data existing in the tables. If the data changes, so may the recommendations were you to run the wizard again. The wizard may not return a recommendation if there is not enough data in the tables being analyzed or if the recommendations it comes up with do not offer enough improvement in performance. Hints can also through off the wizard causing it to give you less than optimal recommendations, if you use hints in your queries.

The Index Tuning Wizard handles a maximum of 32,767 queries and queries that do not contain quoted identifiers. There is also an option, **Keep all existing indexes**, that tells the wizard not to drop any existing indexes if it is enabled. When disabled, the wizard can give better recommendations that include dropping existing but ineffective indexes, as well as dropping and replacing the clustered index on another set of columns. (The wizard does not touch primary keys or unique indexes, though it may move the clustered indexes from an existing primary key or unique index).

There are some situations where the Index Tuning Wizard is not recommended:

- Tuning system tables
- Choosing primary keys and unique indexes
- Tables referenced in cross-database queries

The wizard does not work on version 6.5 databases or earlier. It also does not give an error when saving a script to disk where insufficient space is available. Only members of the fixed **Sysadmin** server role can access the Index Tuning Wizard.

You can start the Index Tuning Wizard by:

1. Connect to a server with Enterprise Manager
2. Highlight a database server
3. Open the **Tools** menu and select **Wizards**.
4. Expand **Management**.
5. Double-click **Index Tuning Wizard**.

You can just follow the steps to complete the wizard.

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/.