

SQL_PERFORM_ETECH_V1.DOC

WHITE PAPER

Version:	1.0
Date:	4/13/2008 10:57 AM
Author	Nilesh Sane
File Name:	SQL_PERFORM_Etech_V1.doc

CONTENTS

1	INTRODUCTION.....	3
2	PURPOSE OF THE DOCUMENT	4
2.1	The Deliverable	4
3	SQL PERFORMANCE – OBJECT WISE	5
3.1	Data Types	5
3.2	Joins.....	6
3.3	Temporary Tables.....	7
3.4	CURSORS	9
3.5	Stored Procedures	10
3.6	Indexes.....	12
4	REFERENCES.....	14

1 INTRODUCTION

SQL Server performance tuning is more art than science. There cannot be step-by-step guide to increase the performance of SQL Server. There are a lot of factors, which affect the performance of SQL Server, and while it is virtually impossible to control every factor that influences SQL Servers' scalability and performance, what you can do is make the most of what you can control.

Here are some (not all) of the factors that affect an application's performance:

- SQL Server (the program itself)
- SQL Server's Configuration Settings
- The Application's Transact-SQL Code
- The Application's non-Transact-SQL Code
- The Database's Design
- The Operating System (server and client)
- The Middleware (Microsoft Transaction Server, Microsoft Messaging Server)
- The Hardware (server and client)
- The Network Hardware and Bandwidth (LAN and WAN)
- The Number of Clients
- The Client Usage Patterns
- The Type and Quantity of Data Stored in SQL Server
- Whether the Application is OLTP- or OLAP-based

2 PURPOSE OF THE DOCUMENT

The purpose of this document is to provide DBA's a ready beckoner for performance tuning and trouble shooting activities. This document can also be used a guideline for designing and implementing SQL Server databases.

2.1 THE DELIVERABLE

This document is divided into three parts. The first part is about optimizing SQL server performance by "object" optimization. The second part is about optimizing "SQL SERVER" settings. The third and final part provides a checklist, which can be used for auditing the performance of SQL Server.

3 SQL PERFORMANCE – OBJECT WISE

3.1 DATA TYPES

- Always specify the narrowest columns you can. The narrower the column, the less amount of data SQL SERVER has to store, and the faster SQL Server is able to read and write data. In addition, if any sorts need to be performed on the column, the narrower the column, the faster the sort will be. [6.5, 7.0, 2000]
- If you need to store large strings of data, and they are less than 8,000 characters, use a VARCHAR data type instead of a TEXT data type. TEXT data types have extra overhead that drag down performance. [7.0, 2000]
- Don't use the NVARCHAR or NCHAR data types unless you need to store 16-bit character (Unicode) data. They take up twice as much space as VARCHAR or CHAR data types, increasing server I/O and wasting unnecessary space in your buffer cache. [7.0, 2000]
- If the text data in a column varies greatly in length, use a VARCHAR data type instead of a CHAR data type. The amount of space saved by using VARCHAR over CHAR on variable length columns can greatly reduce I/O reads, improving overall SQL Server performance.
- Another advantage of using VARCHAR over CHAR columns is that sorts performed on VARCHAR columns are generally faster than on CHAR columns. This is because the entire width of a CHAR column needs to be sorted. [6.5, 7.0, 2000]
- Always choose the smallest data type you need to hold the data you need to store in a column. For example, if all you are going to be storing in a column are the numbers 1 through 10, then the TINYINT data type is more appropriate than the INT data type. The same goes for CHAR and VARCHAR data types. Don't specify more characters in character columns than you need. This allows you to store more rows in your data and index pages, reducing the amount of I/O needed to read them. It also reduces the amount of data moved from the server to the client, reducing network traffic and latency. And last of all, it reduces the amount of wasted space in your buffer cache. [6.5, 7.0, 2000]
- If you are using fixed length columns (CHAR, NCHAR) in your table, do your best to avoid storing NULLs in them. If you do, the entire amount of space dedicated to the column will be used up. For example, if you have a fixed length column of 255 characters, and if you place a NULL in it, then 255 characters have to be stored in the database. This is a large waste of space to store a NULL. This added unnecessary space causes SQL Server to have to perform extra disk I/O to read data pages, and it also wastes space in the data cache buffer. Both of these contribute to reduced SQL Server performance.
Instead of using NULLs, use a coding scheme similar to this in your databases:
NA: Not applicable
NYN: Not yet known
TUN: Truly unknown
Such a scheme provides the benefits of using NULLs, but without the drawbacks.
If you really must use NULLs (I doubt if your reason is very good), use a variable length column instead of a fixed length column. Variable length columns only use a very small amount of space to store a NULL. [7.0, 2000]
- Avoid using the SQL Server 2000 sql_variant datatype. Besides being a performance hog, it significantly affects what you can do with the data stored as a sql_variant. For example, sql_variant columns cannot be a part of primary or foreign keys, can be used in indexes and unique keys if they are shorter than 900 bytes, cannot have an identity property, cannot be part of a computed column, must convert the data to another datatype when moving data to objects with other datatypes, are automatically converted to nvarchar(4000) when accessed by client applications using the SQL Server 7.0 OLE DB or ODBC providers
- If you use a lot of temporary tables in your SQL Server-based applications, take a look at SQL Server 2000 TABLE datatype. The TABLE datatype is used to temporarily hold a rowset, much like a temporary table, but unlike a temporary table, it is not written to the tempdb database. Instead, it is held in memory, making them very fast to use. [2000]

Take care when using Unicode data in your queries, as it can affect query performance. A classic problem is related to an application passing in Unicode literals, while the column searched in the database table is non-Unicode. This, of course, may be visa-versa depending on your scenario.

3.2 JOINS

- One of the best ways to boost JOIN performance is to limit how many rows need to be JOINed. This is especially beneficial for the outer table in a JOIN. Only return absolutely only those rows needed to be JOINed, and no more. [6.5, 7.0, 2000]
- If you perform regular joins between two or more tables in your queries, performance will be optimized if each of the joined columns have their own indexes. This includes adding indexes to the columns in each table used to join the tables. Generally speaking, a clustered key is better than a non-clustered key for optimum JOIN performance. [6.5, 7.0, 2000]
- JOIN performance has a lot to do with how many rows you can stuff in a data page. For example, let's say you want to JOIN two tables. Most likely, one of these two tables will be smaller than the other, and SQL Server will most likely select the smaller of the two tables to be the inner table of the JOIN. When this happens, SQL Server tries to put the relevant contents of this table into the buffer cache for faster performance. If there is not enough room to put all the relevant data into cache, then SQL Server will have to use additional resources in order to get data into and out of the cache as the JOIN is performed.
- If all of the data can be cached, the performance of the JOIN will be faster than if it is not. This comes back to the original statement, that the number of rows in a table can affect JOIN performance. In other words, if a table has no wasted space, it is much more likely to get all of the relevant inner table data into cache, boosting speed. The moral to this story is to try to get as much data stuffed into a data page as possible. This can be done through the use of a high fillfactor, rebuilding indexes often to get rid of empty space, and to optimize datatypes and widths when creating columns in tables. [6.5, 7.0, 2000]
- Keep in mind that when you create foreign keys, an index is not automatically created at the same time. If you ever plan to join a table to the table with the foreign key, using the foreign key as the linking column, then you should consider adding an index to the foreign key column. An index on a foreign key column can substantially boost the performance of many joins. [6.5, 7.0, 2000]
- Avoid joining tables based on columns with few unique values. Ideally, for best performance, joins should be done on columns that have unique indexes. [6.5, 7.0, 2000]
- For best join performance, the indexes on the columns being joined should be numeric data types, not CHAR or VARCHAR, or other non-numeric data types. The overhead is lower and join performance is faster. [6.5, 7.0, 2000]
- When you create joins using Transact-SQL, you can choose between two different types of syntax: either ANSI or Microsoft. ANSI refers to the ANSI standard for writing joins, and Microsoft refers to the old Microsoft style of writing joins. For example:
ANSI JOIN Syntax
SELECT fname, lname, department
FROM names INNER JOIN departments ON names.employeeid = departments.employeeid
Former Microsoft JOIN Syntax
SELECT fname, lname, department
FROM names, departments
WHERE names.employeeid = departments.employeeid
If written correctly, either format will produce identical results. But that is a big if. The older Microsoft join syntax lends itself to mistakes because the syntax is a little less obvious. On the other hand, the ANSI syntax is very explicit and there is no chance you can make a mistake.
- For example, I ran across a slow-performing query from an ERP program. After reviewing the code, which used the Microsoft JOIN syntax, I noticed that instead of creating a LEFT JOIN, the developer had accidentally created a CROSS JOIN instead. In this particular example, less than 10,000 rows should have resulted from the LEFT JOIN, but because a CROSS JOIN was used, over 11 million rows were returned instead. Then the developer used a SELECT DISTINCT to get rid of all the unnecessary rows created by the CROSS JOIN. As you can guess, this made for a very lengthy query. Unfortunately, all I could do was notify the vendor's support department about it.
- The moral of this story is that you probably should be using the ANSI syntax, not the old Microsoft syntax. Besides reducing the odds of making silly mistakes, this code is more portable between database, and eventually, I imagine Microsoft will eventually stop supporting the old format, making the ANSI syntax the only option. [6.5, 7.0, 2000]
- If your join is slow, and currently includes hints, remove the hints to see if the optimizer can do a better job on the join optimization than you can.
- One of the best ways to boost JOIN performance is to ensure that the JOINed tables include an appropriate WHERE clause to minimize the number of rows that need to be JOINed.

- In the SELECT statement that creates your JOIN, don't use an * (asterisk) to return all of the columns in both tables. This is bad form for a couple of reasons. First, you should only return those columns you need, as the less data you return, the faster your query will run. It would be rare that you would need all of the columns in all of the tables you have joined. Second, you will be returning two of each column used in your JOIN condition, which ends up returning way more data that you need, and hurting performance.
- For very large joins, consider placing the tables to be joined in separate physical files in the same filegroup. This allows SQL Server to spawn a separate thread for each file being accessed, boosting performance. [6.5, 7.0, 2000]
- Don't use CROSS JOINS, unless this is the only way to accomplish your goal. What some inexperienced developers do is to join two tables using a CROSS JOIN, then they use either the DISTINCT or the GROUP BY clauses to "clean up" the mess they have created. This, as you might imagine, can be a huge waste of SQL Server resources. [6.5, 7.0, 2000]
- If you have the choice of using a JOIN or a subquery to perform the same task, generally the JOIN (often an OUTER JOIN) is faster. But this is not always the case. For example, if the returned data is going to be small, or if there are no indexes on the joined columns, then a subquery may indeed be faster.
- If you are having difficulty tuning the performance of a poorly performing query that has one or more JOINS, check to see if the query plan created by the query optimizer is using a hash join. When the query optimizer is asked to join two tables that don't have appropriate indexes, it will often perform a hash join.
- A hash join is resource intensive (especially CPU and I/O) and can slow the performance of your join. If the query in question is run often, you should consider adding appropriate indexes. For example, if you are joining column1 in table1 to column5 in table2, then column1 in table1 and column5 in table2 need to have indexes.
- Once indexes are added to the appropriate columns used in the joins in your query, the query optimizer will most likely be able to use these indexes, performing a nested-loop join instead of a hash join, and performance will improve. [7.0, 2000]

3.3 TEMPORARY TABLES

Generally speaking, temp tables should be avoided, if possible. They are created in the tempdb database and create additional overhead for [SQL Server](#), slowing overall performance. As an alternative to temp tables, consider the following alternatives:

- Rewrite your code so that the action you need completed can be done using a standard query or stored procedure, without using a temp table.
 - [Use a derived table](#).
 - Consider using a correlated sub-query.
 - Use a permanent table instead.
 - Use a UNION statement to mimic a temp table. [7.0, 2000]
- **One legitimate use for temp tables is to use them to pass recordsets from a nested stored procedure to a calling stored procedure.** This is the only way to pass recordsets from one stored procedure to another.
 - **Another legitimate reason you might want to consider using a temp table is to avoid having to use a cursor.** [SQL](#) Server cursors have huge overhead and slow SQL Server's performance. One alternative of using a cursor is to use a temp table instead. In almost all cases, using a temp table over a cursor will produce less overhead and better performance. Of course, if you don't have to use a temp table, and you find another way to get away from using a cursor, so much the better.
 - **SQL Server 2000 offers a data type called "table."** Its main purpose is for the temporary storage of a set of rows. A variable, of type "table," behaves as if it is a local variable. And like local variables, it has a limited scope, which is within the batch, function, or stored procedure in which it was declared. In most cases, a table variable can be used like a normal table. SELECTs, INSERTs, UPDATEs, and DELETEs can all be made against a table variable.

If you need a temporary table in your Transact-SQL code, consider using a table variable instead of creating a conventional temporary table instead. Table variables are created and manipulated in memory instead of the tempdb [database](#), making them much faster in some cases. But not in all cases. Because of this, you will need to test both options to determine which works best for you under your particular circumstances.

In addition, table variables found in stored procedures result in fewer compilations (than when using temporary tables), and transactions using table variables only last as long as the duration of an update on the table variable, requiring less locking and logging resources. [2000]

- **If you have no choice but to use a temp table, you can help to optimize their performance** by taking one or more of the following steps:
 - Only include the columns and rows you actually need in the table, no more.
Do not use SELECT INTO to create your temp table, as it places locks on system objects. Instead, create the table using using standard Transact-SQL DDL statements, and then use INSERT INTO to populate the table.
 - Consider using a clustered and non-clustered indexes on your temp tables, especially for very large temp tables. You will have to test to see if indexes help or hurt overall performance.
 - When you are done with your temp table, delete it to free up tempdb resources. Don't wait for the table to be automatically deleted when the connection is ended.
 - If the tempdb database is not already on its own dedicated disk or array, consider taking this step. By isolating the tempdb database on its own disk, disk contention is reduced and performance is increased. Since the tempdb database does not need to be backed up, the tempdb database can be located on a single disk, or for best performance on a [RAID 0](#) array.
- **If you have to use a temp table, don't create it from within a transaction. If you do, then it will lock some system** tables (syscolumns, sysindexes, syscomments), preventing others from executing the same query, greatly hurting concurrency and performance. In effect, this turns your application into a single-user application.
To avoid this problem, create the temporary table before the transaction. This way, the system tables are not locked and multiple users will have the ability to run this same query at the same time, helping concurrency and performance

Derived Tables

As queries become more complex, temporary tables are used more and more. While temporary table may sometimes be unavoidable, they can often be sidestepped by using derived tables instead. In brief, a derived table is the result of using another SELECT statement in the FROM clause of a SELECT statement. By using derived tables instead of temporary tables, we can boost our application's performance. Let's find out more.

How the Use of Temporary Tables Affect Performance

Temporary tables slow performance dramatically. The problem with temporary tables is the amount of overhead that goes along with using them. In order to get the fastest queries possible, our goal must be to make them do as little work as possible. For example, with a SELECT statement, SQL Server reads data from the disk and returns the data. However, temporary tables require the system to do much more.

For example, a piece of Transact-SQL code using temporary tables usually will:

- 1) CREATE the temporary table
- 2) INSERT data into the newly created table
- 3) SELECT data from the temporary table (usually by JOINing to other physical tables) while holding a lock on the entire tempdb database until the transaction has completed.
- 4) DROP the temporary table

This represents a lot of disk activity, along with the potential for contention problems. And all of this adds up to poor performance.

Eliminate A Few Steps!

The biggest benefit of using derived tables over using temporary tables is that they require fewer steps, and everything happens in memory instead of a combination of memory and disk. The fewer the steps involved, along with less I/O, the faster the performance.

Here are the steps when you use a temporary table:

- 1) Lock tempdb database
- 2) CREATE the temporary table (write activity)
- 3) SELECT data & INSERT data (read & write activity)
- 4) SELECT data from temporary table and permanent table(s) (read activity)

- 5) DROP TABLE (write activity)
- 4) Release the locks

Compare the above to the number of steps it takes for a derived table:

- 1) CREATE locks, unless isolation level of "read uncommitted" is used
- 2) SELECT data (read activity)
- 3) Release the locks

As is rather obvious from this example, using derived tables instead of temporary tables reduces disk I/O and can boost performance.

3.4 CURSORS

If possible, **avoid using SQL Server cursors**. They generally use a lot of SQL Server resources and reduce the performance and scalability of your applications. If you need to perform row-by-row operations, try to find another method to perform the task.

Here are some alternatives to using a cursor:

- Use WHILE LOOPS
- Use temp tables
- Use derived tables
- Use correlated sub-queries
- Use the CASE statement
- Perform multiple queries

More often than not, there are non-cursor techniques that can be used to perform the same tasks as a SQL Server cursor. [6.5, 7.0, 2000]

- **If a transaction you have created contains a cursor** (try to avoid this if at all possible), ensure that the number of rows being modified by the cursor is small. This is because the modified rows may be locked until the transaction completes or aborts. The greater the number of rows being modified, the greater the locks, and the higher the likelihood of lock contention on the server, hurting performance. [6.5, 7.0, 2000]
- If you have no choice but to use cursors in your application, try to **locate the SQL Server tempdb database on its own physical device** for best performance. This is because cursors use the tempdb for temporary storage of cursor data. The faster your disk array, the faster your cursor will be. [6.5, 7.0, 2000]
- If you have no choice but to use cursors in your application, try to **locate the SQL Server tempdb database on its own physical device** for best performance. This is because cursors use the tempdb for temporary storage of cursor data. The faster your disk array, the faster your cursor will be. [6.5, 7.0, 2000]
- If you have no choice but to use a server-side cursor in your application, **try to use a FORWARD-ONLY or FAST-FORWARD, READ-ONLY cursor**. When working with unidirectional, read-only data, use the FAST_FORWARD option instead of the FORWARD_ONLY option, as it has some internal performance optimizations to speed performance. This type of cursor produces the least amount of overhead on SQL Server.
- If you are unable to use a fast-forward cursor, then try the following cursors, in this order, until you find one that meets your needs. They are listed in the order of their performance characteristics, from fastest to slowest: dynamic, static, and keyset. [6.5, 7.0, 2000].
- **Using cursors can reduce concurrency and lead to unnecessary locking and blocking**. To help avoid this, use the READ_ONLY cursor option if applicable, or if you need to perform updates, try to use the OPTIMISTIC cursor option to reduce locking. Try to avoid the SCROLL_LOCKS cursor option, which reduces concurrency.
- If you do find you must use a cursor, try to reduce the number of records to process. One way to do this is to move the records that need to be processed into a temp table first, then create the cursor to use the records in the temp table, not from the original table. This of course assumes that the subset of records to be inserted into the temp table are substantially less than those in the original table. The lower the number of records to process, the faster the cursor will finish. [6.5, 7.0, 2000]

If the **number of rows you need to return from a query is small**, and you need to perform row-by-row operations on them, don't use a server-side cursor. Instead, consider returning the entire rowset to the client and have the client perform the necessary action on each row, then return any updated rows to the server. [6.5, 7.0, 2000]

3.5 STORED PROCEDURES

- **Whenever a client application needs to send Transact-SQL** to SQL Server, send it in the form of a stored procedure instead of a script or embedded Transact-SQL. Stored procedures offer many benefits, including:
 - Reduces network traffic and latency, boosting application performance.
 - Stored procedure execution plans can be reused, staying cached in SQL Server's memory, reducing server overhead. Yes, SQL Server 7.0 and 2000 can also catch the execution plans of Transact-SQL not in stored procedures.
 - Client execution requests are more efficient. For example, if an application needs to INSERT a large binary value into an image data column not using a stored procedure, it must convert the binary value to a character string (which doubles its size), and send it to [SQL Server](#). When SQL Server receives it, it then must convert the character value back to the binary format. This is a lot of wasted overhead. A stored procedure eliminates this issue as parameter values stay in the binary format all the way from the application to SQL Server, reducing overhead and boosting performance.
 - Stored procedures help promote code reuse. While this does not directly boost an application's performance, it can boost the productivity of developers by reducing the amount of code required, along with reducing debugging time.
 - Stored procedures can encapsulate logic. You can change stored procedure code without affecting clients (assuming you keep the parameters the same and don't remove any result sets columns). This saves developer time.
 - Stored procedures provide better security to your data. If you use stored procedures exclusively, you can remove direct SELECT, INSERT, UPDATE, and DELETE rights from the tables and force developers to use stored procedures as the method for data access. This saves DBA's time.

Keep in mind that just because you use a stored procedure does not mean that it will run fast. The code you use within your stored procedure must be well designed for both speed and reuse. [6.5, 7.0, 2000]

- **One of the biggest advantages of using stored procedures over not using stored procedures is the ability to significantly reduce network traffic.**
 - By default, every time a stored procedure is executed, a message is sent from the server to the client indicating the number of rows that were affected by the stored procedure. Rarely is this information useful to the client. By turning off this default behavior, you can reduce network traffic between the server and the client, helping to boost overall performance of your server and applications.

There are two main ways to turn this feature off. You can also turn this feature off using a server trace setting, but it is unnecessary as there are easier ways, as described here.

To turn this feature off on at the stored procedure level, you can include the statement:

```
SET NOCOUNT ON
```

at the beginning of each stored procedure you write. This statement should be included in every stored procedure you write.

If you want this feature turned off for your entire server, you can do this by running these statements at your server:

```
SP_CONFIGURE 'user options', 512  
RECONFIGURE
```

- **Keep Transact-SQL transactions as short as possible within a stored procedure.** This helps to reduce the number of locks, helping to speed up the overall performance of your SQL Server application.
- **When a stored procedure is first executed** (and it does not have the WITH RECOMPILE option), it is optimized and a query plan is compiled and cached in SQL Server's buffer. If the same stored procedure is called again from the same connection, it will use the cached query plan instead of creating a new one, often saving time and boosting performance.

- Many stored procedures have the option to accept multiple parameters. This in and of itself is not a bad thing. But what can often cause problems is if the parameters are optional, and the number of parameters varies greatly each time the stored procedure runs. There are two ways to handle this problem, the slow performance way and fast performance way.
- If you want to save your development time, but don't care about your application's performance, you can write your stored procedure generically so that it doesn't care how many parameters it gets. The problem with this method is that you may end up unnecessarily joining tables that don't need to be joined based on the parameters submitted for any single execution of the stored procedure.

Another, much better performing way, although it will take you more time to code, is to include IF...ELSE logic in your stored procedure, and create separate queries for each possible combination of parameters that are to be submitted to the stored procedure. This way, you can be sure your query is as efficient as possible each time it runs. [6.5, 7.0, 2000]

- While temporary stored procedures can provide a small performance boost in some circumstances, **using a lot of temporary stored procedures in your application can actually create contention in the system tables and hurt performance.**

Instead of using temporary stored procedures, you may want to consider using the SP_EXECUTESQL stored procedure instead. It provides the same benefits on temporary stored procedures, but it does not store data in the systems tables, avoiding the contention problems. [7.0, 2000]

- **If you are creating a stored procedure to run in a database other than the Master database, don't use the prefix "sp_" in its name.** This special prefix is reserved for system stored procedures. Although using this prefix will not prevent a user defined stored procedure from working, what it can do is to slow down its execution ever so slightly.

The reason for this is that by default, any stored procedure executed by SQL Server that begins with the prefix "sp_", is first attempted to be resolved in the Master database. Since it is not there, time is wasted looking for the stored procedure.

Before you are done with your stored procedure code, review it for any unused code that you may have forgotten to remove while you were making changes, and remove it. Unused code just adds unnecessary bloat to your stored procedures, although it will not negatively affect performance of the stored procedure. [6.5, 7.0, 2000]

SQL Server will **automatically recompile a stored procedure** if any of the following happens:

- If you include a WITH RECOMPILE clause in a CREATE PROCEDURE or EXECUTE statement.
- If you run sp_recompile for any table referenced by the stored procedure.
- If any schema changes occur to any of the objects referenced in the stored procedure. This includes adding or dropping rules, defaults, and constraints.
- New distribution statistics are generated.
- If you restore a database that includes the stored procedure or any of the objects it references.
- If the stored procedure is aged out of SQL Server's cache.
- An index used by the execution plan of the stored procedure is dropped.
- A major number of INSERTS, UPDATES or DELETES are made to a table referenced by a stored procedure.
- The stored procedure includes both DDL (Data Definition Language) and DML (Data Manipulation Language) statements, and they are interleaved with each other.
- If the stored procedure performs certain actions on temporary tables.

- **One hidden performance problem of using stored procedures is when a stored procedure recompiles too often.** Normally, you want a stored procedure to compile once and to be stored in SQL Server's cache so that it can be re-used without it having to recompile each time it is used. This is one of the major benefits of using stored procedures.
- Use `sp_executesql` instead of `EXECUTE` to run Transact-SQL strings in your stored procedures.
- Instead of writing one very large stored procedure, instead break down the stored procedure into two or more sub-procedures, and call them from a controlling stored procedure.
- If your stored procedure is using temporary tables, use the `KEEP PLAN` query hint, which is used to stop stored procedure recompilations caused by more than six changes in a temporary table, which is the normal behavior. This hint should only be used for stored procedures that access temporary tables a lot, but don't make many changes to them. If many changes are made, then don't use this hint.
- **Improper use of temporary tables** in a stored procedure can force them to be recompiled every time the stored procedure is run. Here's how to prevent this from happening:
 - Any references to temporary tables in your stored procedure should only refer to tables created by that stored procedure, not to temporary tables created outside your stored procedure, or in a string executed using either the `sp_executesql` or the `EXECUTE` statement.
 - All of the statements in your stored procedure that include the name of a temporary table should appear syntactically after the temporary table.
 - The stored procedure should not declare any cursors that refer to a temporary table.
 - Any statements in a stored procedure that refer to a temporary table should precede any `DROP TABLE` statement found in the stored procedure.
 - The stored procedure should not create temporary tables inside a control-of-flow statement.

[7.0, 2000]

- To find out **if your SQL Server is experiencing excessive recompilations of stored procedures**, a common cause of poor performance, create a trace using Profiler and track the `SP:Recompile` event. A large number of recompilations should be an indicator if you potentially have a problem. Identify which stored procedures are causing the problem, and then take correction action (if possible) to reduce or eliminate these excessive recompilations. [7.0, 2000]
- **Avoid nesting stored procedures**, although it is perfectly legal to do so. Nesting not only makes debugging more difficult, it makes it much more difficult to identify and resolve performance-related problems. [6.5, 7.0, 2000]
- **If you use input parameters in your stored procedures, you should validate all of them at the beginning of your stored procedure.**

3.6 INDEXES

How to Select Indexes for Optimal Database Performance

Index selection is a mystery for many SQL Server DBAs and developers. Sure, we know what they do and

how they boost performance. The problem often is how to select the ideal type of index (clustered vs. non-clustered), the number of columns to index (do I need multi-column indexes?), and which columns should be indexed.

In this section we will take a brief look at how to answer the above questions. Unfortunately, there is no absolute answer for every occasion. Like much of SQL Server performance tuning and optimization, you may have to do some experimenting to find the ideal indexes. So let's begin by looking at some general index creation guidelines, then we will take a more detailed look at selecting clustered and non-clustered indexes.

Is There Such a Thing as Too Many Indexes?

Yes. Some people think that all you have to do is index everything, and then all of your performance issues will go away. It doesn't work that way. Just as an index can speed data access, it can also degrade access if it is inappropriately selected. The problem with extra indexes is that SQL Server must maintain them every time that a record is INSERTED, UPDATED, or DELETED from a table. While maintaining one or two indexes on a table is not too much overhead for SQL Server to deal with, if you have four, five, or more indexes, they can be a large performance burden on tables. Ideally, you want to have as few as indexes as you can. It is often a balancing act to select the ideal number of indexes for a table in order to find optimal performance.

As a general rule of thumb, don't automatically add indexes to a table because it seems like the right thing to do. Only add indexes if you know that they will be used by the queries run against the table. If you don't know what queries will be run against your table, then don't add any indexes until you know for sure. It is too easy to make a guess on what queries will be run, create indexes, and then later find out your guesses were wrong. You must know the type of queries that will be run against your data, and then these need to be analyzed to determine the most appropriate indexes, and then the indexes must be created and tested to see if they really help or not.

The problem of selecting optimal indexes is often difficult for OLTP applications because they tend to experience high levels of INSERT, UPDATE, and DELETE activity. While you need good indexes to quickly locate records that need to be SELECTED, UPDATED, or DELETED, you don't want every INSERT, UPDATE, or DELETE to result in too much overhead because you have too many indexes. On the other hand, if you have an OLAP application that is virtually read-only, then adding as many indexes as you need is not a problem because you don't have to worry about INSERT, UPDATE, or DELETE activity. As you can see, how your application is used makes a large difference in your indexing strategy.

Another thing to think about when selecting indexes is that the SQL Server Query Optimizer may not use the indexes you select. If the Query Optimizer chooses not to use your indexes, then they are a burden on SQL Server and should be deleted. So how come the SQL Server Query Optimizer won't always use an index if one is available?

This is too large a question to answer in detail here, but suffice to say, sometimes it is faster for SQL Server to perform a table scan on a table than it is to use an available index to access data in the table. Two reasons that this may happen is because the table is small (not many rows), or if the column that was indexed isn't at least 95% unique. How do you know if SQL Server won't use the indexes you create? We will answer this question a little later when we take a look at how to use the SQL Server Query Analyzer later in this article.

Tips for Selecting a Clustered Index

Since you can only create one clustered index per table, take extra time to carefully consider how it will be used. Consider the type of queries that will be used against the table, and make an educated guess as to which query is the most critical, and if this query will benefit from having a clustered index.

In general, use these rules of thumb when selecting a column for a possible clustered index.

- The primary key you select for your table should not always be a clustered index. If you create the primary key and don't specify otherwise, then SQL Server automatically makes the primary key a clustered index. Only make the primary key a clustered index if it meets one of the following recommendations.
- Clustered indexes are ideal for queries that select by a range of values or where you need sorted results. This is because the data is already presorted in the index for you. Examples of this include when you are using BETWEEN, <, >, GROUP BY, ORDER BY, and aggregates such as MAX, MIN, and COUNT in your queries.
- Clustered indexes are good for queries that look up a record with a unique value (such as an employee number) and when you need to retrieve most or all of the data in the record. This is because the query is covered by the index.
- Clustered indexes are good for queries that access columns with a limited number of distinct values, such as a columns that holds country or state data. But if column data has little distinctiveness, such as columns with a yes or no, or male or female, then these columns should not be indexed at all.
- Clustered indexes are good for queries that use the JOIN or GROUP BY clauses.

- Clustered indexes are good for queries where you want to return a lot of rows, just not a few. This is because the data is in the index and does not have to be looked up elsewhere.
- Avoid putting a clustered index on columns that increment, such as an identity, date, or similarly incrementing columns, if your table is subject to a high level of INSERTS. Since clustered indexes force the data to be physically ordered, a clustered index on an incrementing column forces new data to be inserted at the same page in the table, creating a table hot spot, which can create disk I/O bottlenecks. Ideally, find another column or columns to become your clustered index.

What can be frustrating about the above advice is that there might be more than one column that should be clustered. But as we know, we can only have one clustered index per table. What you have to do is evaluate all the possibilities (assuming more than one column is a good candidate for a clustered index) and then select the one that provides the best overall benefit.

Tips for Selecting Non-Clustered Indexes

Selecting non-clustered indexes is somewhat easier than clustered indexes because you can create as many as is appropriate for your table. Here are some tips for selecting which columns in your tables might be helped by adding non-clustered indexes.

- Non-clustered indexes are best for queries that return few rows (including just one row) and where the index has good selectivity (above 95%).
- If a column in a table is not at least 95% unique, then most likely the SQL Server Query Optimizer will not use a non-clustered index based on that column. Because of this, don't add non-clustered indexes to columns that aren't at least 95% unique. For example, a column with "yes" or "no" as the data won't be at least 95% unique.
- Keep the "width" of your indexes as narrow as possible, especially when creating composite (multi-column) indexes. This reduces the size of the index and reduces the number of reads required to read the index, boosting performance.
- If possible, try to create indexes on columns that have integer values instead of characters. Integer values have less overhead than character values.
- If you know that your application will be performing the same query over and over on the same table, consider creating a covering index on the table. A covering index includes all of the columns referenced in the query. Because of this, the index contains the data you are looking for and SQL Server doesn't have to look up the actual data in the table, reducing logical and/or physical I/O. On the other hand, if the index gets too big (too many columns), this can increase I/O and degrade performance.
- An index is only useful to a query if the WHERE clause of the query matches the column(s) that are leftmost in the index. So if you create a composite index, such as "City, State", then a query such as "WHERE City = 'Houston'" will use the index, but the query "WHERE STATE = 'TX'" will not use the index.

Generally, if a table needs only one index, make it a clustered index. If a table needs more than one index, then you have no choice but to use non-clustered indexes. By following the above recommendations, you will be well on your way to selecting the optimum indexes for your tables.

4 REFERENCES

SQL Server Books On Line (BOL)