Articles » Database » Database » SQL Server

Top 10 steps to optimize data access in SQL Server: Part II (Re-factor TSQL and apply best practices)

Al-Farooque Shubho, 1 Jun 2009 CPOL

★★★★★ 4.92 (143 votes)

As part of a series of articles on data access optimization steps in SQL Server, this article focuses on refactoring and applying TSQL best practices to improve performance.

Introduction

Remember we were in a mission? Our mission was to optimize the performance of a SQL Server database. We had an application that was built on top of that database. The application was working pretty fine while testing, but soon after deployment at production, it started to perform slowly as the data volume increased in the database. Within a few months, the application started performing so slowly that the poor developers (including me) had to start this mission to optimize the database and thus, optimize the application.

Please have a look at the previous article to know how it started and what we did to start the optimization process:

- Top 10 steps to optimize data access in SQL Server: Part I (Use indexing)

Well, in the first 3 steps (discussed in the previous article), we implemented indexing in our database. That was because we had to do something that improved the database performance in a quick amount of time, with the least amount of effort. But, what if our data access code was written in an inefficient way? What if our TSQLs were written poorly?

Applying indexing will obviously improve data access performance, but at the most basic level, in any data access optimization process, you have to make sure that you have written your data access code and TSQLs in the most efficient manner, applying the best practices.

So, in this article, we are going to focus on writing or refactoring data access code using the best practices. But, before we start playing the game, we need to prepare the ground first. So let's do the groundwork in this very next step:

Step 4: Move TSQL code from the application into the database server

I know you may not like this suggestion at all. You might have used an ORM that generates all the SQL for you on the fly. Or, you or your team might have a "principle" of keeping SQL in your application code (in the Data Access Layer methods). But still, if you need to optimize data access performance, or if you need to troubleshoot a performance problem in your application, I would suggest you move your SQL code into your database server (using Stored Procedures, Views, Functions, and Triggers) from your application. Why? Well, I do have some strong reasons for this recommendation:

- Moving SQL from application and implementing them using Stored Procedures/Views/Functions/Triggers will enable you to eliminate any duplicate SQL in your application. This will also ensure re-usability of your TSQL codes.
- Implementing all TSQL using database objects will enable you to analyze the TSQLs more easily to find possible inefficient codes that are responsible for the slow performance. Also, this will let you manage your TSQL codes from a central point.
- Doing this will also enable you to re-factor your TSQL codes to take advantage of some advanced indexing techniques (going to be discussed in the later parts in this series of articles). This will also help you to write more "Set based" SQLs along with eliminating any "Procedural" SQLs that you might have already written in your application.

Despite the fact that indexing (in Step 1 to Step 3) will let you troubleshoot performance problems in your application in a quick time (if properly done), following step 4 might not give you a real performance boost instantly. But, this will mainly enable you to perform other subsequent optimization steps and apply other techniques easily to further optimize your data access routines.

If you have used an ORM (say, NHibernate) to implement the data access routines in your application, you might find your application performing quite well in your development and test environment. But if you face performance problems in a production system where lots of transactions take place each second, and where too many concurrent database connections are there, in order to optimize your application's performance, you might have to re-think about your ORM based data access logic. It is possible to optimize an ORM based data access routine, but, it is always true that if you implement your data access routines using TSQL objects in your database, you have the maximum opportunity to optimize your database.

If you have come this far while trying to optimize your application's data access performance, come on, convince your management and get some time to implement a TSQL object based data operational logic. I can promise you, spending one or two man-months doing this might save you a man-year in the long run!

OK, let's assume that you have implemented your data operational routines using TSQL objects in your database. Having done this step, you are done with the "ground work" and ready to start playing. Let's move towards the most important step in our optimization adventure. We are going to re-factor our data access code and apply best practices.

Step 5: Identify inefficient TSQL, re-factor, and apply best practices

No matter how good indexing you apply to your database, if you use poorly written data retrieval/access logic, you are bound to get slow performance.

We all want to write good code, don't we? While we write data access routines for a particular requirement, we really have lots of options to follow for implementing a particular data access routine (and the application's business logic). But, in most cases, we have to work in a team with members of different caliber, experience, and ideologies. So, while at development, there are strong chances that our team members may write code in different ways, and some of them will skip best practices. While writing code, we all want to "get the job done" first (most of the time). But when our code runs in production, we start to see the problems.

Time to re-factor the code now. Time to implement the best practices in your code.

I have some SQL best practices for you that you can follow. But I am sure that you already know most of them. Problem is, in reality, you just don't implement these good stuff in your code (of course, you always have some good reasons for not doing so). But what happens at the end of the day? Your code runs slowly, and your client becomes unhappy.

While you should know that best practices alone is not enough, you have to make sure that you follow the best practices while writing TSQL. This is the most important thing to remember.

Some TSQL Best Practices

Don't use "SELECT*" in a SQL query

- Unnecessary columns may get fetched that will add expense to the data retrieval time.
- The database engine cannot utilize the benefit of "Covered Index" (discussed in the previous article), and hence the query performs slowly.

Avoid unnecessary columns in the SELECT list and unnecessary tables in join conditions

- Selecting unnecessary columns in a Select query adds overhead to the actual query, specially if the unnecessary columns are of LOB types.

- Including unnecessary tables in join conditions forces the database engine to retrieve and fetch unnecessary data and increases the query execution time.

Do not use the COUNT() aggregate in a subquery to do an existence check

- Do not use:

  ```
  SELECT column_list FROM table WHERE 0 < (SELECT count(*) FROM
  table2 WHERE ..)
  ```

  Instead, use:

  ```
  SELECT column_list FROM table WHERE EXISTS (SELECT * FROM table2
  WHERE ...)
  ```

- When you use `COUNT()`, SQL Server does not know that you are doing an existence check. It counts all matching values, either by doing a table scan or by scanning the smallest non-clustered index.
- When you use `EXISTS`, SQL Server knows you are doing an existence check. When it finds the first matching value, it returns `TRUE` and stops looking. The same applies to using `COUNT()` instead of `IN` or `ANY`.

Try to avoid joining between two types of columns

- When joining between two columns of different data types, one of the columns must be converted to the type of the other. The column whose type is lower is the one that is converted.
- If you are joining tables with incompatible types, one of them can use an index, but the query optimizer cannot choose an index on the column that it converts. For example:

  ```
  SELECT column_list FROM small_table, large_table WHERE
  smalltable.float_column = large_table.int_column
  ```

  In this case, SQL Server converts the integer column to `float`, because `int` is lower in the hierarchy than `float`. It cannot use an index on large_table.int_column, although it can use an index on smalltable.float_column.

Try to avoid deadlocks

- Always access tables in the same order in all your Stored Procedures and triggers consistently.
- Keep your transactions as short as possible. Touch as few data as possible during a transaction.
- Never, ever wait for user input in the middle of a transaction.

Write TSQL using "Set based approach" rather than "Procedural approach"

- The database engine is optimized for Set based SQL. Hence, Procedural approach (use of Cursor or UDF to process rows in a result set) should be avoided when large result sets (more than 1000) have to be processed.
- How can we get rid of "Procedural SQL"? Follow these simple tricks:

   - Use inline sub queries to replace User Defined Functions.
   - Use correlated sub queries to replace Cursor based code.
   - If procedural coding is really necessary, at least, use a table variable instead of a cursor to navigate and process the result set.

For more info on "set" and "procedural" SQL, see Understanding "Set based" and "Procedural" approaches in SQL.

Try not to use COUNT(*) to obtain the record count in a table

- To get the total row count in a table, we usually use the following `Select` statement:

  ```
  SELECT COUNT(*) FROM dbo.orders
  ```

  This query will perform a full table scan to get the row count.

- The following query would not require a full table scan. (Please note that this might not give you 100% perfect results always, but this is handy only if you don't need a perfect count.)

  ```
  SELECT rows FROM sysindexes
  WHERE id = OBJECT_ID('dbo.Orders') AND indid < 2
  ```

Try to avoid dynamic SQL

Unless really required, try to avoid the use of dynamic SQL because:

- Dynamic SQL is hard to debug and troubleshoot.
- If the user provides the input to the dynamic SQL, then there is possibility of SQL injection attacks.

Try to avoid the use of temporary tables

- Unless really required, try to avoid the use of temporary tables. Rather use table variables.
- In 99% of cases, table variables reside in memory, hence it is a lot faster. Temporary tables reside in the TempDb database. So operating on temporary tables require inter database communication and hence will be slower.

Instead of LIKE search, use full text search for searching textual data

Full text searches always outperform LIKE searches.

- Full text searches will enable you to implement complex search criteria that can't be implemented using a LIKE search, such as searching on a single word or phrase (and optionally, ranking the result set), searching on a word or phrase close to another word or phrase, or searching on synonymous forms of a specific word.
- Implementing full text search is easier to implement than LIKE search (especially in the case of complex search requirements).
- For more info on full text search, see http://msdn.microsoft.com/en-us/library/ms142571(SQL.90).aspx

Try to use UNION to implement an "OR" operation

- Try not to use "`OR`" in a query. Instead use "`UNION`" to combine the result set of two distinguished queries. This will improve query performance.
- Better use `UNION ALL` if a distinguished result is not required. `UNION ALL` is faster than `UNION` as it does not have to sort the result set to find out the distinguished values.

Implement a lazy loading strategy for large objects

- Store Large Object columns (like `VARCHAR(MAX)`, `Image`, `Text` etc.) in a different table than the main table, and put a reference to the large object in the main table.
- Retrieve all the main table data in a query, and if a large object is required to be loaded, retrieve the large object data from the large object table only when it is required.

Use VARCHAR(MAX), VARBINARY(MAX), and NVARCHAR(MAX)

- In SQL Server 2000, a row cannot exceed 8000 bytes in size. This limitation is due to the 8 KB internal page size of SQL Server. So to store more data in a single column, you need to use `TEXT`, `NTEXT`, or `IMAGE` data types (BLOBs) which are stored in a collection of 8 KB data pages.
- These are unlike the data pages that store other data in the same table. These pages are arranged in a B-tree structure. These data cannot be used as variables in a procedure or a function, and they cannot be used inside string functions such as `REPLACE`, `CHARINDEX`, or `SUBSTRING`. In most cases, you have to use `READTEXT`, `WRITETEXT`, and `UPDATETEXT`.
- To solve this problem, use `VARCHAR(MAX)`, `NVARCHAR(MAX)`, and `VARBINARY(MAX)` in SQL Server 2005. These data types can hold the same amount of data BLOBs can hold (2 GB), and they are stored in the same type of data pages used for other data types.
- When data in a `MAX` data type exceeds 8 KB, an over-flow page is used (in the `ROW_OVERFLOW` allocation unit), and a pointer to the page is left in the original data page in the `IN_ROW` allocation unit.

Implement the following good practices in User Defined Functions

- Do not call functions repeatedly within your Stored Procedures, triggers, functions, and batches. For example, you might need the length of a string variable in many places of your procedure, but don't call the `LEN` function whenever it's needed; instead, call the `LEN` function once, and store the result in a variable for later use.

Implement the following good practices in Stored Procedures

- Do not use "SP_XXX" as a naming convention. It causes additional searches and added I/O (because the system Stored Procedure names start with "SP_"). Using "SP_XXX" as the naming convention also increases the possibility of conflicting with an existing system Stored Procedure.
- Use "`Set Nocount On`" to eliminate extra network trip.

- Use the `WITH RECOMPILE` clause in the `EXECUTE` statement (first time) when the index structure changes (so that the compiled version of the Stored Procedure can take advantage of the newly created indexes).
- Use default parameter values for easy testing.

Implement the following good practices in Triggers

- Try to avoid the use of triggers. Firing a trigger and executing the triggering event is an expensive process.
- Never use triggers that can be implemented using constraints.
- Do not use the same trigger for different triggering events (Insert, Update, Delete).
- Do not use transactional code inside a trigger. The trigger always runs within the transactional scope of the code that fires the trigger.

Implement the following good practices in Views

- Use views for re-using complex TSQL blocks, and to enable it for indexed views (Will be discussed later).
- Use views with the `SCHEMABINDING` option if you do not want to let users modify the table schema accidentally.
- Do not use views that retrieve data from a single table only (that will be an unnecessary overhead). Use views for writing queries that access columns from multiple tables.

Implement the following good practices in Transactions

- Prior to SQL Server 2005, after `BEGIN TRANSACTION` and each subsequent modification statement, the value of `@@ERROR` had to be checked. If its value was non-zero, then the last statement caused an error, and if an error occurred, the transaction had to be rolled back and an error had to be raised (for the application). In SQL Server 2005 and onwards, the `Try...Catch` block can be used to handle transactions in TSQL. So try to use `Try...Catch` based transactional code.
- Try to avoid nested transactions. Use the `@@TRANCOUNT` variable to determine whether a transaction needs to be started (to avoid nested transactions).
- Start a transaction as late as possible and commit/rollback the transaction as fast as possible to reduce the time period of resource locking.

And, that's not the end. There are lots of best practices out there! Try finding some of them from the following URL: MSDN.

Remember, you need to implement the good things that you know; otherwise, your knowledge will not add any value to the system that you are going to build. Also, you need to have a process for reviewing and monitoring the code (that is written by your team) to see whether the data access code is being written following the standards and best practices.

How to analyze and identify scope for improvement in your TSQL?

In an ideal world, you always prevent diseases rather than cure. But, in reality, you just can't prevent always. I know your team is composed of brilliant professionals. I know you have a good review process, but still bad code is written and poor design takes place. Why? Because, no matter what advanced technology you are going to use, your client requirement will always be way much advanced, and this is a universal truth in software development. As a result, designing, developing, and delivering a system based on requirements will always be a challenging job for you.

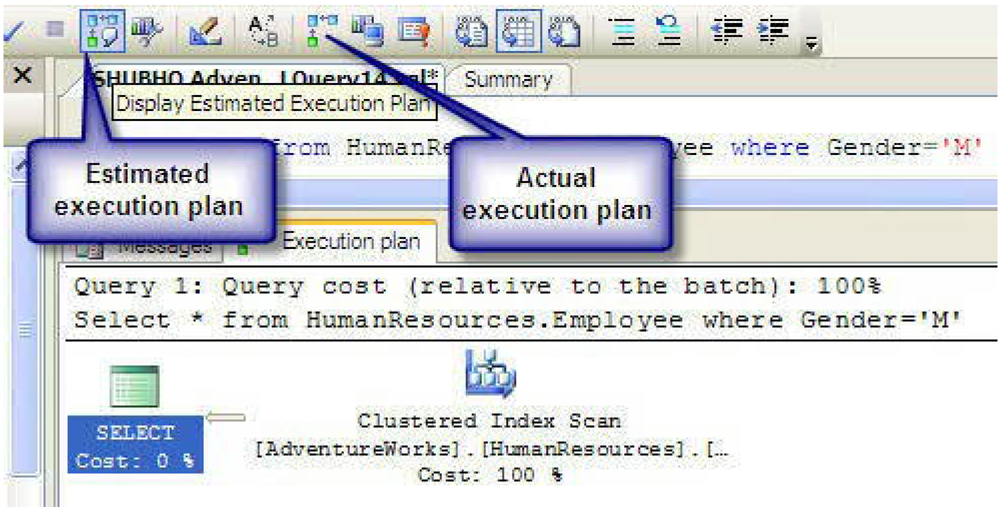
So, it's equally important that you know how to cure. You really need to know how to troubleshoot a performance problem after it happens. You need to learn ways to analyze yout TSQL code, identify the bottlenecks, and re-factor those to troubleshoot performance problems. There are numerous ways to troubleshoot database and TSQL performance problems, but at the most basic level, you have to understand and review the execution plan of the TSQL that you need to analyze.

Understanding the query execution plan

Whenever you issue a SQL statement in the SQL Server engine, SQL Server first has to determine the best possible way to execute it. In order to carry this out, the Query Optimizer (a system that generates the optimal query execution plan before executing the query) uses several information like the data distribution statistics, index structure, metadata, and other information to analyze several possible execution plans and finally select one that is likely to be the best execution plan most of the time.

Did you know? You can use SQL Server Management Studio to preview and analyze the estimated execution plan for the query that you are going to issue. After writing the SQL in SQL Server Management Studio, click on the estimated execution plan icon (see below) to see the execution plan before actually executing the query.

(Note: Alternatively, you can switch the actual execution plan option "on" before executing the query. If you do this, Management Studio will include the actual execution plan that is being executed along with the result set in the result window.)



```
Query 1: Query cost (relative to the batch): 100%
Select * from HumanResources.Employee where Gender='M'
```

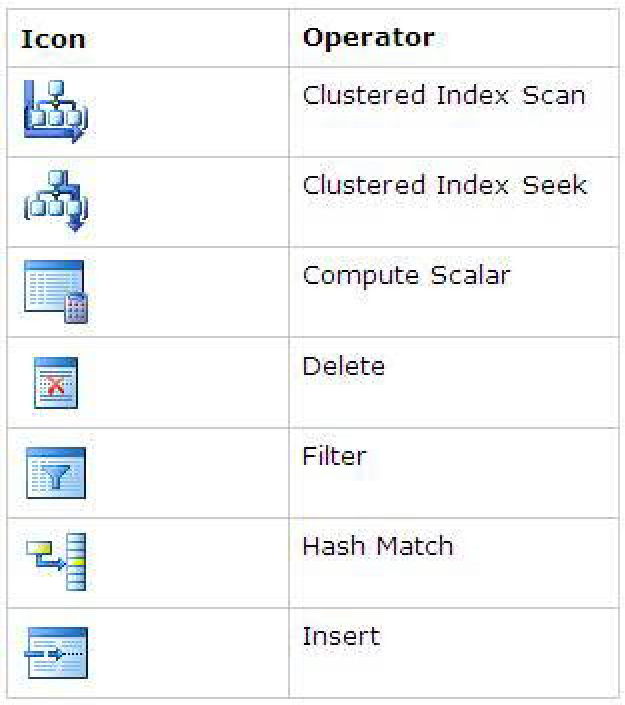
Estimated execution plan in Management Studio

Understanding the query execution plan in detail

Each icon in the execution plan graph represents an action item (Operator) in the plan. The execution plan has to be read from right to left, and each action item has a percentage of cost relative to the total execution cost of the query (100%).

In the above execution plan graph, the first icon in the right most part represents a "Clustered Index Scan" operation (reading all primary key index values in the table) in the HumanResources table (that requires 100% of the total query execution cost), and the left most icon in the graph represents a SELECT operation (that requires only 0% of the total query execution cost).

Following are the important icons and their corresponding operators you are going to see frequently in the graphical query execution plans:

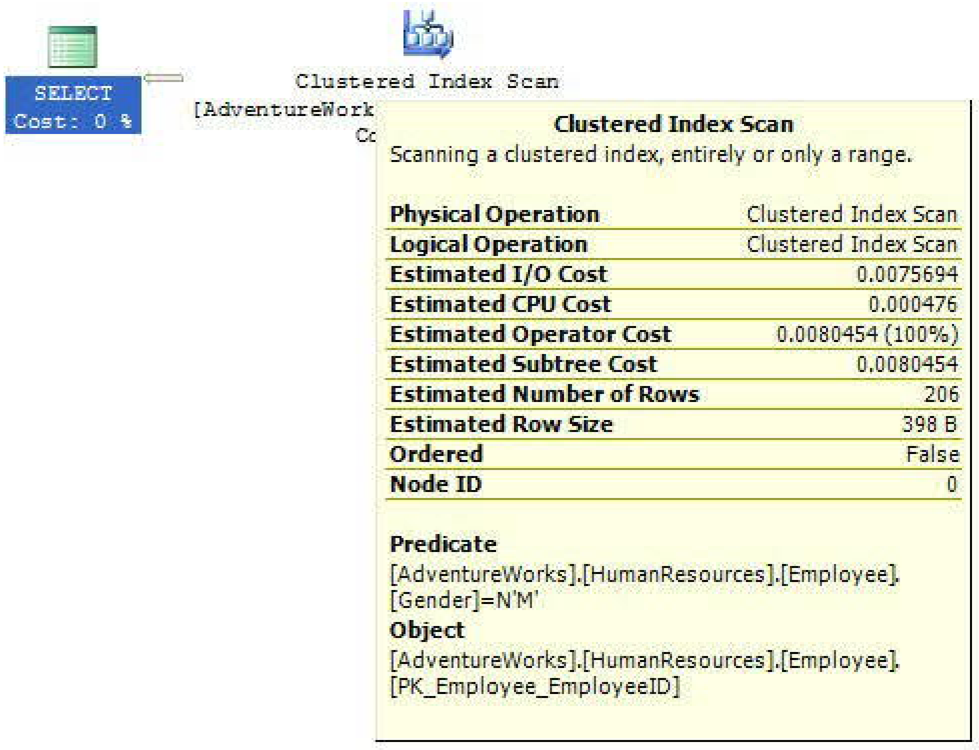


Icon	Operator
	Clustered Index Scan
	Clustered Index Seek
	Compute Scalar
	Delete
	Filter
	Hash Match
	Insert

(Each icon in the graphical execution plan represents a particular action item in the query. For a complete list of the icons and their corresponding action items, go to http://technet.microsoft.com/en-us/library/ms175913.aspx.)

Note the "Query cost" in the execution plan given above. It has 100% cost relative to the batch. That means, this particular query has 100% cost among all queries in the batch as there is only one query in the batch. If there were multiple queries simultaneously executed in the query window, each query would have its own percentage of cost (less than 100%).

To know more details for each particular action item in the query plan, move the mouse pointer on each item/icon. You will see a window that looks like the following:



Clustered Index Scan
Scanning a clustered index, entirely or only a range.

Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Estimated I/O Cost	0.0075694
Estimated CPU Cost	0.000476
Estimated Operator Cost	0.0080454 (100%)
Estimated Subtree Cost	0.0080454
Estimated Number of Rows	206
Estimated Row Size	398 B
Ordered	False
Node ID	0

Predicate
[AdventureWorks].[HumanResources].[Employee].[Gender]=N'M'
Object
[AdventureWorks].[HumanResources].[Employee].[PK_Employee_EmployeeID]

This window provides detailed estimated information about a particular query item in the execution plan. The above window shows the estimated detailed information for the clustered index scan and it looks for the row(s) which have/has Gender = 'M' in the Employee table in HumanResources schema in the AdventureWorks database. The window also shows the estimated IO, CPU, number of rows, with the size of each row, and other costs that is used to compare with other possible execution plans to select the optimal plan.

I found an article that can help you further understand and analyze TSQL execution plans in detail. You can take a look at it here: http://www.simple-talk.com/sql/performance/execution-plan-basics/.

What information do we get by viewing the execution plans?

Whenever any of your query performs slowly, you can view the estimated (and, actual if required) execution plan and can identify the item that is taking the most amount of time (in terms of percentage) in the query. When you start reviewing any TSQL for optimization, most of the time, the first thing you would like to do is view the execution plan. You will most likely quickly identify the area in the SQL that is creating the bottleneck in the overall SQL.

Keep watching for the following costly operators in the execution plan of your query. If you find one of these, you are likely to have problems in your TSQL and you need to re-factor the TSQL to improve performance.

Table Scan: Occurs when the corresponding table does not have a clustered index. Most likely, creating a clustered index or defragmenting index will enable you to get rid of it.

Clustered Index Scan: Sometimes considered equivalent to Table Scan. Takes place when a non-clustered index on an eligible column is not available. Most of the time, creating a non-clustered index will enable you to get rid of it.

Hash Join: The most expensive joining methodology. This takes place when the joining columns between two tables are not indexed. Creating indexes on those columns will enable you to get rid of it.

Nested Loops: Most cases, this happens when a non-clustered index does not include (Cover) a column that is used in the SELECT column list. In this case, for each member in the non-clustered index column, the database server has to seek into the clustered index to retrieve the other column value specified in the SELECT list. Creating a covered index will enable you to get rid of it.

RID Lookup: Takes place when you have a non-clustered index but the same table does not have any clustered index. In this case, the database engine has to look up the actual row using the row ID, which is an expensive operation. Creating a clustered index on the corresponding table would enable you to get rid of it.

TSQL Refactoring - A real life story

Knowledge comes into value only when applied to solve real-life problems. No matter how knowledgeable you are, you need to utilize your knowledge in an effective way in order to solve your problems.

Let's read a real life story. In this story, Mr. Tom is one of the members of the development team that built the application that we mentioned earlier.

When we started our optimization mission in the data access routines (TSQLs) of our application, we identified a Stored Procedure that was performing way below the expected level of performance. It was taking more than 50 seconds to process and retrieve sales data for one month for particular sales items in the production database. Following is how the Stored Procedure was getting invoked for retrieving sales data for 'Caps' for the year 2009:

```
exec uspGetSalesInfoForDateRange '1/1/2009', 31/12/2009,'Cap'
```

Accordingly, Mr. Tom was assigned to optimize the Stored Procedure.

Following is a Stored Procedure that is somewhat close to the original one (I can't include the original Stored Procedure for proprietary issues):

```
ALTER PROCEDURE uspGetSalesInfoForDateRange
    @startYear DateTime,
    @endYear DateTime,
    @keyword nvarchar(50)
AS
BEGIN
    SET NOCOUNT ON;
    SELECT
       Name,
       ProductNumber,
       ProductRates.CurrentProductRate Rate,
    ProductRates.CurrentDiscount Discount,
    OrderQty Qty,
    dbo.ufnGetLineTotal(SalesOrderDetailID) Total,
    OrderDate,
    DetailedDescription
    FROM
    Products INNER JOIN OrderDetails
    ON Products.ProductID = OrderDetails.ProductID
    INNER JOIN Orders
    ON Orders.SalesOrderID = OrderDetails.SalesOrderID
    INNER JOIN ProductRates
    ON
    Products.ProductID = ProductRates.ProductID
    WHERE
    OrderDate between @startYear and @endYear
    AND
    (
        ProductName LIKE '' + @keyword + ' %' OR
        ProductName LIKE '% ' + @keyword + ' ' + '%' OR
        ProductName LIKE '% ' + @keyword + '%' OR
        Keyword LIKE '' + @keyword + ' %' OR
        Keyword LIKE '% ' + @keyword + ' ' + '%' OR
        Keyword LIKE '% ' + @keyword + '%'
    )
    ORDER BY
    ProductName
END

GO
```

Analyzing the indexes

As a first step, Mr. Tom wanted to review the indexes of the tables that were being queried in the Stored Procedure. He had a quick look into the query and identified the fields that the tables should have indexes on (for example, fields that have been used in the join queries, `WHERE` conditions, and `ORDER BY` clauses). Immediately, he found that several indexes are missing on some of these columns. For example, indexes on the following two columns were missing:

- OrderDetails.ProductID
- OrderDetails.SalesOrderID

He created non-clustered indexes on those two columns, and executed the Stored Procedure as follows:

```
exec uspGetSalesInfoForDateRange '1/1/2009', 31/12/2009 with recompile
```

The Stored Procedure's performance was improved now, but still below the expected level (35 seconds). (Note the "`with recompile`" clause. It forces the SQL Server engine to recompile the Stored Procedure and re-generate the execution plan to take advantage of the newly built indexes).

Analyzing the query execution plan

Mr. Tom's next step was to see the execution plan in the SQL Server Management Studio. He did this by writing the '`exec`' statement for the Stored Procedure in the query window and viewing the "Estimated execution plan". (The execution plan is not included here as it is quite a big one that is not going to fit in the screen.)

Analyzing the execution plan, he identified some important scopes for improvement:

- A table scan was taking place on a table while executing the query even though the table has a proper indexing implementation. The table scan was taking 30% of the overall query execution time.
- A "nested loop join" (one of three kinds of joining implementation) was occurring for selecting a column from a table specified in the SELECT list in the query.

Curious about the table scan issue, Mr. Tom wanted to know if any index fragmentation took place or not (because all indexes were properly implemented). He ran a TSQL that reports the index fragmentation information on table columns in the database (he collected this from a CodeProject article on data access optimization) and was surprised to see that two of the existing indexes (in the corresponding tables used in the TSQL in the Stored Procedure) had fragmentation that were responsible for the table scan operation. Immediately, he defragmented those two indexes and found out that the table scan was not occurring and the Stored Procedure was taking 25 seconds now to execute.

In order to get rid of the "nested loop join", he implanted a "Covered index" in the corresponding table including the column in the SELECT list. As a result, when selecting the column, the database engine was able to retrieve the column value in the non-clustered index node. Doing this reduced the query performance up to 23 seconds now.

Implementing some best practices

Mr. Tom now decided to look for any piece of code in the Stored Procedure that did not conform to the best practices. Following were the changes that he did to implement some best practices:

Getting rid of the "Procedural code"

Mr. Tom identified that a UDF `ufnGetLineTotal(SalesOrderDetailID)` was getting executed for each row in the result set, and the UDF simply was executing another TSQL using a value in the supplied parameter and was returning a scalar value. Following was the UDF definition:

```
ALTER FUNCTION [dbo].[ufnGetLineTotal]
(
    @SalesOrderDetailID int
)
RETURNS money
AS
BEGIN

    DECLARE @CurrentProductRate money
    DECLARE @CurrentDiscount money
    DECLARE @Qty int

    SELECT
        @CurrentProductRate = ProductRates.CurrentProductRate,
        @CurrentDiscount = ProductRates.CurrentDiscount,
        @Qty = OrderQty
    FROM
        ProductRates INNER JOIN OrderDetails ON
        OrderDetails.ProductID = ProductRates.ProductID
    WHERE
        OrderDetails.SalesOrderDetailID = @SalesOrderDetailID

    RETURN (@CurrentProductRate-@CurrentDiscount)*@Qty
END
```

This seemed to be a "Procedural approach" for calculating the order total, and Mr. Tom decided to implement the UDF's TSQL as an inline SQL in the original query. Following was the simple change that he had to implement in the Stored Procedure:

```
dbo.ufnGetLineTotal(SalesOrderDetailID) Total        -- Old Code
(CurrentProductRate-CurrentDiscount)*OrderQty Total -- New Code
```

Immediately after executing the query, Mr. Tom found that the query was taking 14 seconds now to execute.

Getting rid of the unnecessary Text column in the SELECT list

Exploring for further optimization scope, Mr. Tom decided to take a look at the column types in the SELECT list in the TSQL. Soon he discovered that a `Text` column (Products.DetailedDescription) was included in the SELECT list. Reviewing the application code, Mr. Tom found that this column value was not being used by the application immediately. A few columns in the result set were being displayed in a listing page in the application, and when the user clicked on a particular item in the list, a detail page was appearing containing the `Text` column value.

Excluding that `Text` column from the SELECT list dramatically reduced the query execution time from 14 seconds to 6 seconds! So, Mr. Tom decided to apply a "Lazy loading" strategy to load this `Text` column using a Stored Procedure that accepts an "ID" parameter and selects the `Text` column value. After implementation, he found out that the newly created Stored Procedure executes in a reasonable amount of time when the user sees the detail page for an item in the item list. He also converted those two "`Text`" columns to `VARCHAR(MAX)` columns, and that enabled him to use the `len()` function on one of these two columns in the TSQL in other places (that also allowed him to save some query execution time because he was calculating the length using `len` (`Text_Column as Varchar(8000)`) in the earlier version of the code.

Optimizing further: Process of elimination

What's next? All the optimization steps so far reduced the execution time to 6 seconds. Comparing to the execution time of 50 seconds before optimization, this is a big achievement so far. But Mr. Tom thinks the query could have further improvement scope. Reviewing the TSQL code, Mr. Tom didn't find any significant option left for further optimization. So he indented and re-arranged the TSQL (so that each individual query statement (say, Product.ProductID = OrderDetail.ProductID) is written in a particular line) and started executing the Stored Procedure again and again by commenting out each line that he suspected for having improvement scope.

Surprise! Surprise! The TSQL had some `LIKE` conditions (the actual Stored Procedure basically performed a keyword search on some tables) for matching several patterns against some column values. When he commented out the `LIKE` statements, suddenly the Stored Procedure execution time jumped below 1 second. Wow!

It seemed that having done with all the optimizations so far, the LIKE searches were taking the most amount of time in the TSQL code. After carefully looking at the LIKE search conditions, Mr. Tom became pretty sure that the LIKE search based SQL could easily be implemented using Full Text Search. It seemed that two columns needed to be full text search enabled. These were: ProductName and Keyword.

It just took 5 minutes for him to implement the FTS (creating the Full Text catalog, making the two columns full text enabled, and replacing the `LIKE` clauses with the `FREETEXT` function), and the query started executing now within a stunning 1 second!

Great achievement, isn't it?

What's next?

We've learned lots of things in optimizing data access code, but we've still miles to go. Data access optimization is an endless process that gives you endless thrills and fun. No matter how big systems you have, no matter how complex your business processes are, believe me, you can make them run faster, always!

So, let's not stop here. Let's go through the next article in this series:

- Top 10 steps to optimize data access in SQL Server: Part III (Apply advanced indexing and denormalization)

Help optimizing. Have fun!

History

- First version: 19 April 2009.

License

Share

About the Author

Al-Farooque Shubho
Founder DropCue, SmartAspects
Bangladesh 🇧🇩

I write codes to make life easier, and that pretty much describes me.

Sorry for not being able to contribute to CodeProject these days. I'be been busy with DropCue, which is not just another "to-do-list" or calendar management app, but an app to manage all of your "personal aspects" in one single place in such a simple, easy and innovative approach that no other system offers.

Give it a try, I bet you'll love it!

Follow on Twitter

Comments and Discussions

104 messages have been posted for this article Visit http://www.codeproject.com/Articles/35665/Top-steps-to-optimize-data-access-in-SQL-Serv to post and view comments on this article, or click here to get a print view with messages.