**We have a database to optimize, let's start!**

When a database based application performs slowly, there is a 90% probability that the data access routines of that application are not optimized, or not written in the best possible way. So, you need to review and optimize your data access/manipulation routines for improving the overall application performance.

**Let us start our optimization mission in a step-by-step process:**

**Step 1: Apply proper indexing in the table columns in the database**

Well, some could argue whether implementing proper indexing should be the first step in the performance optimization process for a database. But I would prefer applying indexing properly in the database in the first place, because of the following two reasons:

1. This will allow you to achieve the best possible performance in the quickest amount of time in a production system.
2. Applying/creating indexes in the database will not require you to do any application modification, and thus will not require any build and deployment.

Of course, this quick performance improvement can be achieved if you find that indexing is not properly done in the current database. However, if indexing is already done, I would still recommend you to go through this step.

**What is indexing?**

I believe you know what indexing is. But, I've seen many people being unclear on this. So, let us try to understand indexing once again. Let us read a small story.

*Long ago, there was a big library in an ancient city. It had thousands of books, but the books ware not arranged in any order in the book shelves. So, each time a person asked for a book to the librarian, the librarian had no way but to check every book to find the required book that the person wanted. Finding the desired book used to take hours for the librarian, and most of the time, the persons who asked for the book had to wait for a long time.*

[Hmm... sounds like a table that has no primary key. When data is searched for in a table, the database engine has to scan through the entire table to find the corresponding row, which performs very slow.]

*Life was getting miserable for the librarian as the number of books and persons asking for books increased day by day. Then one day, a wise guy came to the library, and seeing the librarian's measurable life, he advised him to number each book and arrange the book shelves according to their numbers. "What benefit would I get?", asked the librarian. The wise guy answered, "Well, now if somebody gives you a book number and asks for that book, you will be able to find the shelves quickly that contains the book's number, and within that shelf, you can find that book very quickly as books are arranged according to their number".*

[Numbering the books sounds like creating a primary key in a database table. When you create a primary key in a table, a clustered index tree is created, and all data pages containing the table rows are physically sorted in the file system according to their primary key values. Each data page contains rows, which are also sorted within the data page according to their primary key values. So, each time you ask for any row from the table, the database server finds the corresponding data page first using the clustered index tree (like finding the book shelf first) and then finds the desired row within the data page that contains the primary key value (like finding the book within the shelf).]

*"This is exactly what I need!" The excited librarian instantly started numbering the books and arranging them across different book shelves. He spent a whole day to do this arrangement, but at the end of the day, he tested and found that a book now could be found using the number within no time at all! The librarian was extremely happy.*

[That's exactly what happens when you create a primary key in a table. Internally, a clustered index tree is created, and the data pages are physically sorted within the data file according to the primary key values. As you can easily understand, only one clustered index can be created for a table as the data can be physically arranged only using one column value as the criteria (primary key). It's like the books can only be arranged using one criterion (book number here).]

*Wait! The problem was not completely solved yet. The very next day, a person asked a book by the book's name (he didn't have the book's number, all he had was the book's name). The poor librarian had no way but to scan all the numbered books from 1 to N to find the one the person asked for. He found the book in the 67th shelf. It took 20 minutes for the librarian to find the book. Earlier, he used to take 2-3 hours to find a book when they were not arranged in the shelves, so that was an improvement. But, comparing to the time to search a book using its number (30 seconds), this 20 minute seemed to be a very high amount of time to the librarian. So, he asked the wise man how to improve on this.*

[This happens when you have a Product table where you have a primary key ProductID, but you have no other index in the table. So, when a product is to be searched using the Product Name, the database engine has no way but to scan all physically sorted data pages in the file to find the desired item.]

*The wise man told the librarian: "Well, as you have already arranged your books using their serial numbers, you cannot re-arrange them. Better create a catalog or index where you will have all the book's names and their corresponding serial numbers. In this catalog, arrange the book names in their alphabetic number and group the book names using each alphabet so that if any one wants to find a book named "Database Management System", you just follow these steps to find the book:*

1. *Jump into the section "D" of your book name "catalog" and find the book name there.*
2. *Read the corresponding serial number of the book and find the book using the serial number (you already know how to do this).*

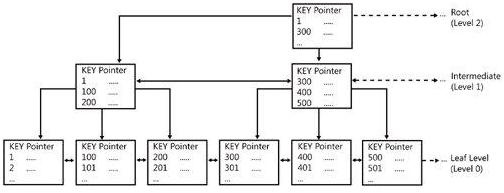
*"You are a genius!", exclaimed the librarian. Spending some hours, he immediately created the "Book name" catalog, and with a quick test, he found that he only required a minute (30 seconds to find the book's serial number in the "Book name" catalog and 30 seconds to find the book using the serial number) to find a book using its name.*

*The librarian thought that people might ask for books using several other criteria like book name and/or author's name etc., so he created a similar catalog for author names, and after creating these catalogs, the librarian could find any book using a common book finding criteria (serial number, book name, author's name) within a minute. The miseries of the librarian ended soon, and lots of people started gathering at the library for books as they could get books really fast, and the library became very popular.*

*The librarian started passing his life happily ever after. The story ends.*

By this time, I am sure you have understood what indexes really are, why they are important, and what their inner workings are. For example, if we have a "Products" table, along with creating a clustered index (that is automatically created when creating the primary key in the table), we should create a non-clustered index on the *ProductName* column. If we do this, the database engine creates an index tree for the non-clustered index (like the "book name" catalog in the story) where the product names will be sorted within the index pages. Each index page will contain a range of product names along with their corresponding primary key values. So, when a product is searched using the product name in the search criteria, the database engine will first seek the non-clustered index tree for product name to find the primary key value of the book. Once found, the database engine then searches the clustered index tree with the primary key to find the row for the actual item that is being searched.

Following is how an index tree looks like:



***Index tree structure***

This is called a B+ Tree (Balanced tree). The intermediate nodes contain a range of values and directs the SQL engine where to go while searching for a specific index value in the tree, starting from the root node. The leaf nodes are the nodes which contain the actual index values. If this is a clustered index tree, the leaf nodes are the physical data pages. If this is a non-clustered index tree, the leaf nodes contain index values along with the clustered index keys (which the database engine uses to find the corresponding row in the clustered index tree).

Usually, finding a desired value in the index tree and jumping to the actual row from there takes an extremely small amount of time for the database engine. So, indexing generally improves the data retrieval operations.

Time to apply indexing in your database to retrieve results fast!

Follow these steps to ensure proper indexing in your database:

**Make sure that every table in your database has a primary key.**

This will ensure that every table has a clustered index created (and hence, the corresponding pages of the table are physically sorted in the disk according to the primary key field). So, any data retrieval operation from the table using the primary key, or any sorting operation on the primary key field or any range of primary key values specified in the where clause will retrieve data from the table very fast.

**Create non-clustered indexes on columns which are:**

* Frequently used in the search criteria
* Used to join other tables
* Used as foreign key fields
* Of having high selectivity (column which returns a low percentage (0-5%) of rows from a total number of rows on a particular value)
* Used in the ORDER BY clause
* Of type XML (primary and secondary indexes need to be created; more on this in the coming articles)

Following is an example of an index creation command on a table:

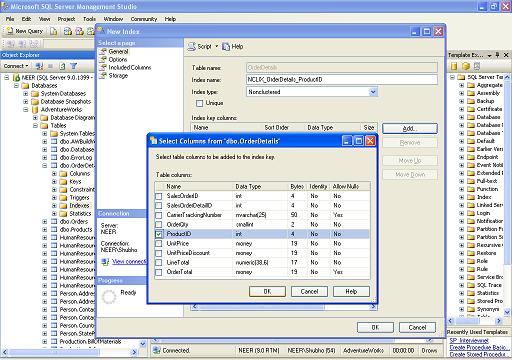
Hide Copy Code

CREATE INDEX

NCLIX\_OrderDetails\_ProductID ON

dbo.OrderDetails(ProductID)

Alternatively, you can use SQL Server Management Studio to create an index on the desired table



***Creating an index using SQL Server Management Studio***

**Step 2: Create the appropriate covering indexes**

So you have created all the appropriate indexes in your database, right? Suppose, in this process, you have created an index on a foreign key column (*ProductID*) in the *Sales(SelesID,SalesDate,SalesPersonID,ProductID,Qty)* table. Now, assuming that the *ProductID* column is a "highly selective" column (selects less than 5% of the total number of rows rows using any *ProductID* value in the search criteria), any SELECT query that reads data from this table using the indexed column (*ProductID*) in the where clause should run fast, right?

Yes, it does, compared to the situation where no index is created on the foreign key column (*ProductID*), in which case, a full table scan is done (scanning all related pages in the table to retrieve desired data). But still, there is further scope to improve this query.

Let's assume that the Sales table contains 10,000 rows, and the following SQL selects 400 rows (4% of the total rows):

Hide Copy Code

SELECT SalesDate, SalesPersonID FROM Sales WHERE ProductID = 112

Let's try to understand how this SQL gets executed in the SQL execution engine:

1. The *Sales* table has a non-clustered index on the *ProductID* column. So it "seeks" the non-clustered index tree for finding the entry that contains *ProductID*=112.
2. The index page that contains the entry *ProductID* = 112 also contains all the clustered index keys (all primary key values, that is *SalesIDs*, that have *ProductID* = 112 assuming that the primary key is already created in the Sales table).
3. For each primary key (400 here), the SQL Server engine "seeks" into the clustered index tree to find the actual row locations in the corresponding page.
4. For each primary key, when found, the SQL Server engine selects the *SalesDate* and *SalesPersonID* column values from the corresponding rows.

Please note that in the above steps, for each of the primary key entries (400 here) for *ProductID* = 112, the SQL Server engine has to search the clustered index tree (400 times here) to retrieve the additional columns (*SalesDate*, *SalesPersonID*) in the query.

It seems that, along with containing clustered index keys (primary key values), if the non-clustered index page could also contain two other column values specified in the query (*SalesDate*, *SalesPersonID*), the SQL Server engine would not have to perform steps 3 and 4 above and, thus, would be able to select the desired results even faster just by "seeking" into the non-clustered index tree for the *ProductID* column and reading all the three mentioned column values directly from that index page.

Fortunately, there is a way to implement this feature. This is what is called "covered index". You create "covered indexes" in table columns to specify what additional column values the index page should store along with the clustered index key values (primary keys). Following is an example of creating a covered index on the *ProductID* column in the Sales table:

Hide Copy Code

CREATE INDEX NCLIX\_Sales\_ProductID*--Index name*

ON dbo.Sales(ProductID)*--Column on which index is to be created*

INCLUDE(SalesDate, SalesPersonID)*--Additional column values to include*

Please note that covered index should be created including a few columns that are frequently used in the Select queries. Including too many columns in the covered indexes would not give you too much benefit. Rather, doing this would require too much memory to store all the covered index column values, resulting in over consumption of memory and slow performance.

**Use the Database Tuning Advisor's help while creating covered index**

We all know, when a SQL is issued, the optimizer in the SQL Server engine dynamically generates different query plans based on:

* Volume of data
* Statistics
* Index variation
* Parameter value in TSQL
* Load on server

That means, for a particular SQL, the execution plan generated in the production server may not be the same execution plan that is generated in the test server, even though the table and index structure are the same. This also indicates that an index created in the test server might boost some of your TSQL performance in the test application, but creating the same index in the production database might not give you any performance benefit in the production application! Why? Well, because the SQL execution plans in the test environment utilizes the newly created indexes and thus gives you better performance. But the execution plans that are being generated in the production server might not use the newly created index at all for some reasons (for example, a non-clustered index column is not "highly" selective in the production server database, which is not the case in the test server database).

So, while creating indexes, we need to make sure that the index would be utilized by the execution engine to produce faster results. But, how can we do this?

The answer is, we have to simulate the production server's load in the test server, and then need to create the appropriate indexes and test those. Only then, if the newly created indexes improve performance in the test environment, these will most likely improve performance in the production environment.

Doing this should be hard, but fortunately, we have some friendly tools to do this. Follow these instructions:

1. Use SQL Profiler to capture traces in the production server. Use the Tuning template (I know, it is advised not to use SQL Profiler in a production database, but sometimes you have to use it while diagnosing performance problems in production). If you are not familiar with this tool, or if you need to learn more about profiling and tracing using SQL Profiler, read <http://msdn.microsoft.com/en-us/library/ms181091.aspx>.
2. Use the trace file generated in the previous step to create a similar load in the test database server using the Database Tuning Advisor. Ask the Tuning Advisor to give some advice (index creation advice in most cases). You are most likely to get good realistic (index creation) advice from the Tuning Advisor (because the Tuning Advisor loads the test database with the trace generated from the production database and then tried to generate the best possible indexing suggestion). Using the Tuning Advisor tool, you can also create the indexes that it suggests. If you are not familiar with the Tuning Advisor tool, or if you need to learn more about using the Tuning Advisor, read <http://msdn.microsoft.com/en-us/library/ms166575.aspx>.

Step 3: Defragment indexes if fragmentation occurs

OK, you created all the appropriate indexes in your tables. Or, may be, indexes are already there in your database tables. But you might not still get the desired good performance according to your expectations.

There is a strong chance that index fragmentation has occurred.

**What is index fragmentation?**

Index fragmentation is a situation where index pages split due to heavy insert, update, and delete operations on the tables in the database. If indexes have high fragmentation, either scanning/seeking the indexes takes much time, or the indexes are not used at all (resulting in table scan) while executing queries. Thus, data retrieval operations perform slow.

Two types of fragmentation can occur:

* **Internal Fragmentation**: Occurs due to data deletion/update operations in the index pages which end up in the distribution of data as a sparse matrix in the index/data pages (creates lots of empty rows in the pages). Also results in an increase of index/data pages that increases query execution time.
* **External Fragmentation**: Occurs due to data insert/update operations in the index/data pages which end up in page splitting and allocation of new index/data pages that are not contiguous in the file system. That reduces performance in determining the query result where ranges are specified in the "where" clauses. Also, the database server cannot take advantage of the read-ahead operations as the next related data pages are not guaranteed to be contiguous, rather these next pages could be anywhere in the data file.

**How to know whether index fragmentation has occurred or not?**

Execute the following SQL in your database (the following SQL will work in SQL Server 2005 or later databases). Replace the database name '*AdventureWorks*' with the target database name in the following query:

Hide Copy Code

SELECT object\_name(dt.object\_id) Tablename,si.name

IndexName,dt.avg\_fragmentation\_in\_percent AS

ExternalFragmentation,dt.avg\_page\_space\_used\_in\_percent AS

InternalFragmentation

FROM

(

SELECT object\_id,index\_id,avg\_fragmentation\_in\_percent,avg\_page\_space\_used\_in\_percent

FROM sys.dm\_db\_index\_physical\_stats (db\_id('AdventureWorks'),null,null,null,'DETAILED'

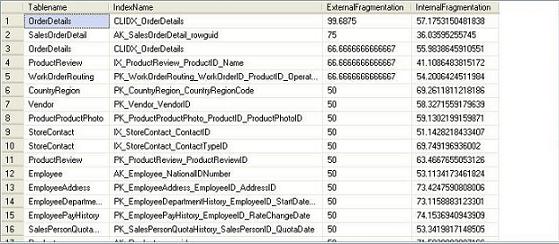
)

WHERE index\_id <> 0) AS dt INNER JOIN sys.indexes si ON si.object\_id=dt.object\_id

AND si.index\_id=dt.index\_id AND dt.avg\_fragmentation\_in\_percent>10

AND dt.avg\_page\_space\_used\_in\_percent<75 ORDER BY avg\_fragmentation\_in\_percent DESC

The above query shows index fragmentation information for the '*AdventureWorks*' database as follows:



***Index fragmentation information***

Analyzing the result, you can determine where the index fragmentation has occurred, using the following rules:

* *ExternalFragmentation* value > 10 indicates external fragmentation occurred for the corresponding index
* *InternalFragmentation* value < 75 indicates internal fragmentation occurred for the corresponding index

**How to defragment indexes?**

You can do this in two ways:

* Reorganize the fragmented indexes: execute the following command to do this:

Hide Copy Code

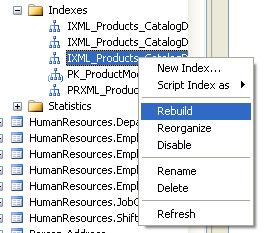
ALTER INDEX ALL ON TableName REORGANIZE

* Rebuild indexes: execute the following command to do this:

Hide Copy Code

ALTER INDEX ALL ON TableName REBUILD WITH (FILLFACTOR=90,ONLINE=ON)

You can also rebuild or reorganize individual indexes in the tables by using the index name instead of the 'ALL' keyword in the above queries. Alternatively, you can also use SQL Server Management Studio to do index defragmentation.



***Rebuilding index using SQL Server Management Studio***

**When to reorganize and when to rebuild indexes?**

You should "reorganize" indexes when the External Fragmentation value for the corresponding index is between 10-15 and the Internal Fragmentation value is between 60-75. Otherwise, you should rebuild indexes.

One important thing with index rebuilding is, while rebuilding indexes for a particular table, the entire table will be locked (which does not occur in the case of index reorganization). So, for a large table in the production database, this locking may not be desired, because rebuilding indexes for that table might take hours to complete. Fortunately, in SQL Server 2005, there is a solution. You can use the *ONLINE* option as *ON* while rebuilding indexes for a table (see the index rebuild command given above). This will rebuild the indexes for the table, along with making the table available for transactions.

**Last words**

It's really tempting to create an index on all eligible columns in your database tables. But, if you are working with a transactional database (an OLTP system where update operations take place most of the time), creating indexes on all eligible columns might not be desirable every time. In fact, creating heavy indexing on OLTP systems might reduce the overall database performance (as most operations are update operations, updating data means updating indexes as well).

A rule of thumb can be suggested as follows: if you work on a transactional database, you should not create more than 5 indexes on the tables on an average. On the other hand, if you work on a data warehouse application, you should be able to create up to 10 indexes on the tables on an average.

**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)**](http://www.codeproject.com/KB/database/OptimizeDBUseIndexing.aspx)

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:

Hide Copy Code

SELECT column\_list FROM table WHERE 0 < (SELECT count(\*) FROM table2 WHERE ..)

Instead, use:

Hide Copy Code

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:

Hide Copy Code

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.](http://www.codeproject.com/KB/database/SetAndProceduralSQL.aspx)

**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:

Hide Copy Code

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.)

Hide Copy Code

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](http://msdn.microsoft.com/en-us/library/ms142571%28SQL.90%29.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](http://code.msdn.microsoft.com/SQLExamples/Wiki/View.aspx?title=Best%20practices%20%2C%20Design%20and%20Development%20guidelines%20for%20Microsoft%20SQL%20Server).

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.

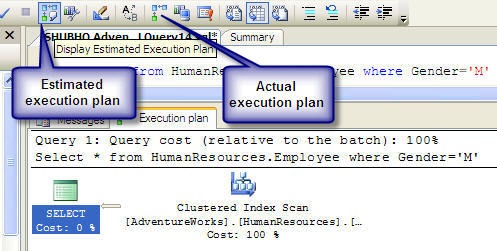
So, it's equally important that you know how to cure. You really need to know how to troubleshoot a performance problem after it happens. You need to learn ways to analyze yout TSQL code, identify the bottlenecks, and re-factor those to troubleshoot performance problems. There are numerous ways to troubleshoot 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.)



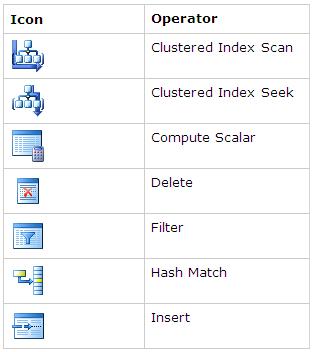
***Estimated execution plan in Management Studio***

**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).

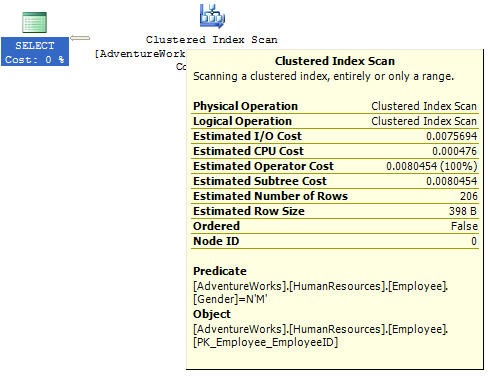
Following are the important icons and their corresponding operators you are going to see frequently in the graphical query execution plans:



(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:



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:

Hide Copy Code

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):

Hide Shrink http://www.codeproject.com/images/arrow-up-16.pngCopy Code

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:

Hide Copy Code

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](http://www.codeproject.com/KB/database/OptimizeDBUseIndexing.aspx) 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:

Hide Copy Code

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:

Hide Copy Code

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?