

Increasing Database Performance using Indexes

Cecilia CIOLOCA, Mihai GEORGESCU

Economic Informatics Department, Academy of Economic Studies

Bucharest, ROMANIA

cecilia_cioloca@yahoo.com, mihai.georgescu@europe.com

The performance issues are presented. The process of performance tuning is described. The indexing concept is introduced. The two types of defining an index are identified. The paper presents and explains the way a clustered and nonclustered index works. A compared analysis between clustered and nonclustered indexes is created. The benefits of using indexes on databases are identified and explained. A demonstration of using SEEK and SCAN indexes is presented, showing the logic behind these two types of indexes. Cases when an index must be used and how this index is created, are presented along with the performance improvement that is obtained after successfully creating the index.

Keywords: Clustered index, Nonclustered index, Optimization, Database, Performance

1 Introduction

Performance is one of the most important metric that describes if a project is a success or a mistake. It is also one of the most common problems the programmers are dealing with. Either if we are taking into consideration a new starting project or an application that is already running on production we should always keep in mind the performance aspects. This means that the design for performance process should start early in the development of an application. The architecture of the system should be design in a manner to meet the performance requirements and to allow performance tuning. The book [1] presents techniques to improve or fine tune a database to achieve maximum performance.

There is no recipe of designing perfect databases, but there are techniques and tips that can improve the quality of the design. Improving the database performance is a cycling activity that should be including in each of development stage.

The performance tuning process includes three steps:

- Time response measurement before tuning;
- Tuning performed;

- Time response measurement after tuning.

The database designer should focus on those techniques that provide the most benefits. Among all the techniques of improving the database performance, indexing and query optimization stand up as they provide visible results. On the other hand, abusing indexes and inappropriate indexes might harm the performance of the system.

The structure of the database used for this demonstration is described in figure 1.

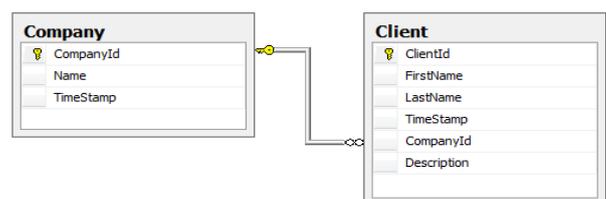


Fig. 1 – Structure of test database

In order to define the index concept we will take a look at how the data is retrieved from database with no indexes.

```
SELECT * FROM Client WHERE
LastName = 'LastName4'
```

In the above query the SQL Server will look in all the rows from the beginning of the table to the end of it searching for those rows that are meeting the condition in the

WHERE clause. If the searched table contains few rows the response of the above query might be very prompt, but in case of tables that contain millions of rows the query might take a while. For this reason creating an index on the table allows SQL Server to get the result without searching the whole data in the table. Indexing is the best way to reduce the logical read and disk input/output as it provides a mechanism of structured search instead of row by row search [8]. Basically an index is a copy of the data in the table, sorted in a certain logical manner. There are two different ways of defining an index:

- *Like a dictionary*: a dictionary is a list of words ordered alphabetically. An index defined like a dictionary is a set of data ordered in a lexicographic manner. For this reason the search on the index will not include all the rows but will position easier based on the ordered data.
- *Like a book index*: this approach of creating an index will not alter the layout of the data in the table, but just like the index of a book will position the data in the table to the corresponding position in the table. An index defined in this manner will contain the data in the indexed column and the corresponding row number of the data.

An example of how to design a spatial database is presented in detail in [3].

2 Clustered indexes

A clustered index is an index that contains the table data ordered in a physical manner. When creating a clustered index on a table, the data rows are reordered on the disk based on the index key sequence so that it meets the indexed ordering. For this reason only one clustered index is allowed to be created on one single table.

For the Client table, when creating a clustered index on *FirstName* column,

the data in the table is physically alphabetically sorted based on *FirstName* value. When inserting a new row into the database, it will be inserted in a certain position so that the sorting is still kept. Figure 2 schematically presents the tree structure of the clustered index on *Client* table column *FirstName*.

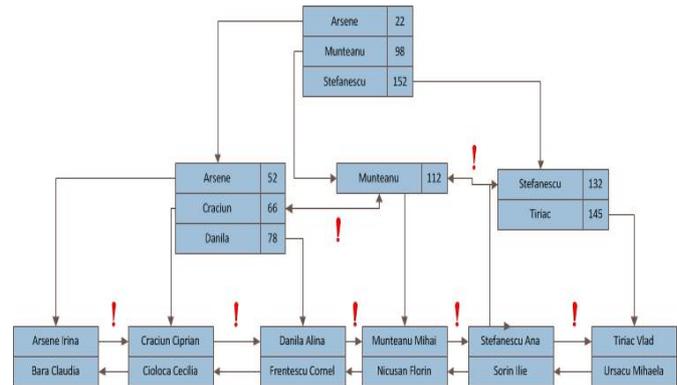


Fig. 2 – The structure of the clustered index

As shown in the figure above the leaf nodes represent the actual data pages while the intermediate nodes of the tree structure are index pages. All the pages in the structure are linked. The top node in the structure is the root index page, while the middle level nodes are intermediate index pages. Each row in an index page refers either another index page or a data page. This reference is a pointer to a page number. The root index page contains a row with the value *Munteanu* which points to the intermediate index page number 98, while the index page 98 contains a row with the value *Danila* which points to the data page 78. The data page 78 contains two rows with the corresponding values.

When searching using the clustered index, the row to row search will be avoided. For the following query

```
SELECT * FROM Client WHERE
LastName='Cioloca'
```

SQL Server will first get the root index page from the sysindexes table, and then it will search in the rows of it to find the highest key value not greater than *Cioloca*, and will

3 Nonclustered indexes

Similar to the clustered indexes, nonclustered indexes are balanced tree structures that start from a root index node, and include intermediate index nodes and leaf nodes. The main differences between clustered and nonclustered indexes is that in case of nonclustered indexes the key values are not ordered in any specific way and that the leaf nodes from nonclustered indexes are not data pages but index pages that point to either a index node into a clustered index or to a data row into the heap. Figure 3 describes the tree structure of a nonclustered index. The dotted line suggests that the leaf nodes point to data rows into the heap table.

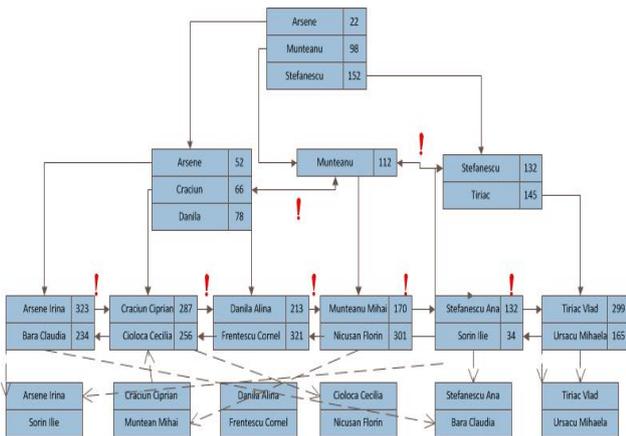


Fig. 3 – Structure of nonclustered index

Due to its properties, a nonclustered index does not physically change the order of the rows into the table. The figure above also shows that each data row is pointed by an index node, which was not the case for clustered index. The nonclustered index is larger than its counterpart, as a supplementary index level is introduced and it has to contain the pointers to the data rows. Even when the clustered index rows have physically changed the nonclustered row locaters continue to point to the same clustered index nodes. Instead of updating each row locator for each nonclustered index node, SQL Server adds a pointer to the old data page in

order to point to the new datapage. This involves a new level of indirection. The leaf index nodes from the nonclustered index still point to the old data page, while these old data pages are pointing to the new ones. This way the user is transferred to the new location of the data pages.

While there can be only one single clustered index for each table, there can be a bunch of nonclustered indexes on the same table. The reason of that is that this type of index does not change the physical order of the data into the table.

When using nonclustered indexes it is useful to keep in mind the following aspects:

- Nonclustered indexes might be find very usefull when retrieving a small set of data from a large amount of data. The nonclustered indexes is also recommended when using wide keys and when the table columns are frequently updated;
- Nonclustered indexes are not the best when retrieving large sets of data, as by using this type of index there still must be read a large amount of data pages. In this case the use of clustered indexes is recommended.

Using noncluster indexes should be used when:

- Once or more rows will be retrieved—that is, the query is highly selective.
- The nonclustered index key is used in the ORDER BY clause or GROUP BY clause of critical queries.
- The nonclustered index key is used in important joins to relate the tables.
- A covered query is required.

4 Index SEEK versus Index SCAN

In order to evaluate the performance of an sql query, one of the most common tool is SQL Server trace which provides information about the “logical reads” counter.

As shown in [7], the logical and physical reads are explained: “The I/O from an instance of SQL Server is divided into logical and physical I/O. A logical read

occurs every time the database engine requests a page from the buffer cache. If the page is not currently in the buffer cache, a physical read is then performed to read the page into the buffer cache. If the page is currently in the cache, no physical read is generated; the buffer cache simply uses the page already in memory.”

Physical and logical reads are available in SQL Server 2008 by using the following command:

```
SET STATISTICS IO ON
```

After successfully running this command, all future queries will show the operations performed.

First we will create a nonclustered index on column *LastName*:

```
CREATE NONCLUSTERED INDEX
LastNameIndex on Client(LastName)
```

By using the system procedure *sp_helpindex* we can check what indexes are defined on a table:

```
sp_helpindex Client
```

In figure 4, we can see that we have 2 indexes defined on our Client table, PK_Client index is generated automatically when we specified the primary key for the table.

index_name	index_description	index_keys
1 LastNameIndex	nonclustered located on PRIMARY	LastName
2 PK_Client	clustered, unique, primary key located on PRIMARY	ClientId

Fig. 4 – Current indexes on table Client

To understand how index seek and index scan works, the execution plan is turned on and the following queries are executed:

```
Query 1:
select * from Client where
LastName = 'LastName4'
```

```
Query 2:
select * from Client where
FirstName = 'FirstName4'
```

We observe for the first query that SQL Server requested a page from the buffer cache only 15 times. Also in figure 5, by examining the execution plan, the database engine used an Index Seek to fetch the information that was already sorted using the LastNameIndex that we created earlier.

The second query used an Index Scan which means that the entire Client table was scanned to find the Clients that have *LastName = 'LastName4'* and requested data from cache 19458 times, a lot more than the first query which used the index.

Another information that the execution plan provides is the query cost. For the first query is almost 0% compared to the second query which scanned the entire table.

Query 1:

(4 row(s) affected)
Table 'Client'. Scan count 1, logical reads 15, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Query 2:

(4 row(s) affected)
Table 'Client'. Scan count 1, logical reads 19458, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

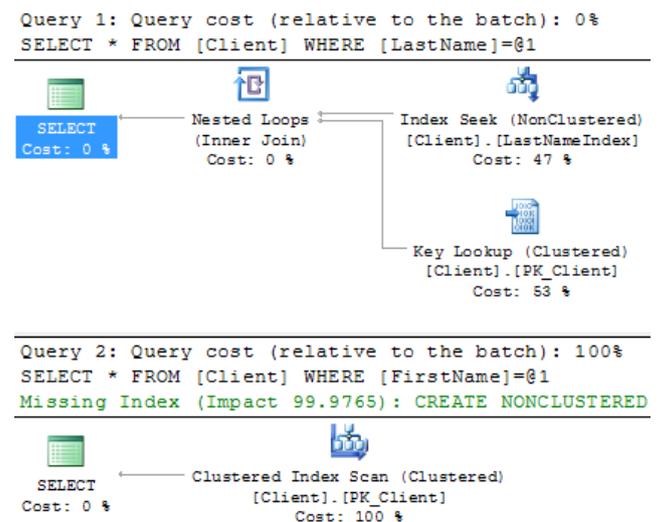


Fig. 5 – Execution plan of Query 1 and Query 2

An index scan either clustered or nonclustered will do a complete scan in a

clustered or nonclustered index. Neither index uses the b-tree structure of the index but only reads the leaf pages in order using each page's reference to the next in the chain.

An index seek is a seek through the b-tree structure of a nonclustered index, from the root down to the leaf. A clustered index seek is looking in the structure of a clustered index from the root down to the leaf.

Index seek is preferred for the highly selective queries, when the query is returning a fewer number of rows, less than 15% of total rows in the table.

If there is an index on a table and if the query is fetching a large amount of data, more than 70% of total rows in a table, then the database engine will scan all the data pages to retrieve the data rows.

5 Clustered indexes compared to nonclustered indexes

Clustered and Non Clustered indexes have the following common characteristics as presented in [5]:

Both clustered and nonclustered indexes are created based on one or more columns. The columns are put in the index in the order you specify them in the CREATE INDEX statement. They are also sorted in this order as well. Indexes are first sorted on the first column in the index, then any duplicates of the first column and sorted by the second column, etc;

Neither clustered or nonclustered indexes will guarantee the sort order of the data when it is being returned to you. If the order of the data matters to you, you should always sort the data with the ORDER BY clause in your select statement.

Both clustered indexes and nonclustered indexes take up additional disk space. The amount of space that they require will depend on the columns in the index, and the number of rows in the table. The clustered index will also grow as you add columns to the table;

Adding indexes (both clustered and nonclusterd) will increase the amount of execution time that the INSERT, UPDATE and DELETE statement take, as the data has to be updated in the table an also in each index;

Columns of the IMAGE data type cannot be indexed using normal indexes;

Clustered Indexes

SQL Server supports only one clustered index for each database table. This index should be the most common column that is in the WHERE clause for queries against this table. In most cases the search is by primary key and this index is automatically created.

The reordering of the data occurs every time the index changes;

Reorders the way records are physically stored;

Nonclustered Indexes

Nonclustered indexes, also called "indexes" are the normal indexes that are created on your tables. SQL Server supports 999 nonclustered indexes per table.

Each nonclustered index can contain upto 1023 included columns. Columns of the TEXT, NTEXT and IMAGE data types cannot be included either.

Nonclustered indexes are not copies of the table but a sorting of the columns that refer to the data pages in the clustered index. Because of this, the clustered index affects all other indexes.

There are two modes for the nonclustered index: non-unique and unique. The non-unique type specifies that the index allows identical rows to be inserted, compared to the unique type which does not allow any identical rows.

As presented in [6], the nonclustered index does not re-order the actual table data. If the clustered index is missing, then the nonclustered index will point to the actual data in table.

Logical order of the index is not identical with the physical order of the rows stored on disk.

Nonclustered indexes should be used for queries that return small amount of data and to columns used in WHERE clauses that return exact matches. Large result sets are reading more tables pages anyway so they will not benefit from a nonclustered index.

6 Index fragmentation

When the user performs operations like INSERT, UPDATE or DELETE on the database, table fragmentation may occur. Also when changes affect the data that is covered by the index, then index fragmentation occurs and the information in the index is scattered over the database. Due to this, when a query is performed against a heavy fragmented table, the operation will take longer time. Index fragmentation comes in two different forms: external fragmentation and internal fragmentation. In each of these forms, the pages within the index are used inefficiently. External fragmentation means that the logical order of the pages are wrong and internal fragmentation represents that the amount of data stored within each page is less than the data page.

Each leaf page of an index must be in a logical order otherwise external fragmentation occurs. When an index is created, the index keys are sorted in a logical order on a set of index pages. Each time new data is inserted in the index, there could be the possibility that the new keys are inserted between existing keys. This may lead to the creation of new index pages to accommodate the existing keys that were moved so that new keys can be inserted in correct order.

Let's assume the current index structure presented in figure 6.



Fig. 6 – Current index structure

When an INSERT statement is executed,

new data is added to the index. In our example we will add 5 and automatically a new page will be created and the 7 and 8 will be moved to the new page in order to make room for the 5 on the original page. Due to this, the index will be out of logical order as seen in figure 7.

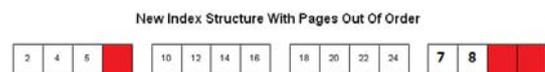


Fig. 7 – New index structure

This type of fragmentation will affect the performance of queries that do not have specific searches or that return unordered result sets, but will affect queries that returned ordered sets. An example of an ordered result set is a query that is returning everything from page 4 to 12. This query has to complete an extra page switch in order to return the 7 and 8 pages. If the fragmentation affects tables with hundreds of pages the amount of extra page switches will be significantly greater.

In order to determine the level of fragmentation the following command can be used:

```
DBCC SHOWCONTIG (TableName)
```

The syntax of this command is:

```
DBCC SHOWCONTIG
[ ( { table_name
    | table_id
    | view_name
    | view_id }
  [ , index_name | index_id ]
)
]
[ WITH { ALL_INDEXES
    | FAST [ , ALL_INDEXES ]
    | TABLERESULTS [ , { ALL_INDEXES } ]
  [ , { FAST | ALL_LEVELS } ]
}
]
```

After running this command on a table or view, the results are presented below:

```
DBCC SHOWCONTIG scanning 'Person' table...
Table: 'Person' (839842950); index ID: 1,
database ID: 5
```

```

TABLE level scan performed.
- Pages Scanned.....: 53600
- Extents Scanned.....: 6712
- Extent Switches.....: 8579
- Avg. Pages per Extent.....: 8.0
- Scan Density [Best Count:Actual
Count].....: 78.09% [6700:8580]
- Logical Scan Fragmentation .....: 5.31%
- Extent Scan Fragmentation .....: 43.19%
- Avg. Bytes Free per Page.....: 753.2
- Avg. Page Density (full).....: 90.69%
DBCC execution completed. If DBCC printed
error messages, contact your system
administrator.

```

The command returns the number of pages scanned, the number of extents scanned the number of times the DBCC statement moved from one extent to another while parsing the pages of the table or index, the average number of pages per extent, the scan density.

Pages Scanned: If the number of rows contained in the table or index divided by the approximate row size is significantly greater than the number of pages scanned then there could be internal fragmentation of the index.

Extents Scanned: Take the number of pages scanned and divide that number by 8, rounded to the next highest interval. This figure should match the number of extents scanned returned by DBCC SHOWCONTIG. If the number returned by DBCC SHOWCONTIG is higher, then you have some external fragmentation. The seriousness of the fragmentation depends on just how high the shown value is from the estimated value.

Extent Switches: This number should be equal to (Extents Scanned – 1). Higher numbers indicate external fragmentation.

Extent Scan Fragmentation: Shows any gaps between extents. This percentage should be 0% and higher percentages indicate external fragmentation.

By analyzing the results provided by DBCC SHOWCINTIG on our “Person” table we can see that the number of extent switches is much greater than the number of extents scanned. In this case there is external fragmentation.

The fragmentation can be reduced and the performance improved by using the following methods:

- Dropping and recreating the index;
- Rebuilding the index using the DBCC DBREINDEX statement;
- Defragmenting an index by using the DBCC INDEXDEFRAG statement.

Dropping and rebuilding an index has the advantage of completely rebuilding an index and does reorder the index pages, compacting the pages, and dropping any unneeded pages. This operation should be done on indexes that show high levels of both internal and external fragmentation.

DROP INDEX and CREATE INDEX

Rebuilding the index can reduce fragmentation and it is done by using the following statement:

DBCC DBREINDEX

This operation is similar to dropping and creating the index, except that it will rebuild the index physically allowing the SQL Server to assign new pages to the index and reduce both internal and external fragmentation. This statement also recreates indexes with existing constraints.

Defragmenting an index by using the DBCC INDEXDEFRAG statement reduces the external fragmentation by rearranging the existing leaf pages of an index to the logical order of the index key and internal fragmentation by compacting the rows within index pages then discarding unnecessary pages. The time needed to execute this statement is longer than recreating an index if the amount of fragmentation is high. DBCC INDEXDEFRAG can defragment an index while other processes are accessing the index, eliminating the blocking restrictions.

7 Conclusions

Using indexes on the database improves the performance of the whole system. These indexes are of different type and should be used in an intelligent way in order to achieve the wished behavior. Clustered indexes should be used for queries that return a large amount of data. The clustered index is recommended to be created on the primary key of the table. For all other columns nonclustered indexes should be created.

A noncluster index is better to be used when running queries that return a smaller amount of data. By using this type of indexes large amount of data will have to read in lot of data pages, and the performance of the system will decrease. Choosing the number and type of indexes can dramatically affect the performance and can mean the difference between data being retrieved in seconds, with few disk I/Os or minutes, even hours, with many disk I/Os. Choosing the optimum number of indexes to support the critical queries is therefore an extremely important task. Before creating a new index one must make sure that there aren't any unused indexes. Also remove the minimally useful or redundant/duplicate indexes like subsets of other indexes.

Combine existing indexes to create a consolidated index:

```
Index1:
(LastName,   FirstName)   INCLUDE
(phone)
```

```
Index2:
(LastName,   FirstName,   MiddleName)
INCLUDE (PIN)
```

```
Combined Index:
(LastName,   FirstName,   MiddleName)
INCLUDE (phone, PIN)
```

Verify the health of the existing indexes periodically for:

- Splits using sys.dm_db_index_operational_stats
- Fragmentation using sys.dm_db_index_physical_stats

- Contention using sys.dm_db_index_operational_stats

The database performance can be improved by reducing the table or index fragmentation. This is achieved by dropping and recreating the index, rebuild the index by using DBCC DBREINDEX statement or by defragmenting the index using DBCC INDEXDEFRAG statement.

8 Acknowledgment

This work was cofinanced from the European Social Fund through Sectoral Operational Programme Human Resources Development 2007-2013, project number POSDRU/107/1.5/S/77213 „Ph.D. for a career in interdisciplinary economic research at the European standards”.

References

- [1] G. Fritchey, S. Dam, *SQL Server 2008 Query Performance Tuning Distilled*, Apress, 2009, ISBN 978-1-4302-1903-3, 560 pages
- [2] K. England, *Microsoft SQL Server 2000 Performance Optimisation and Tuning Handbook*, Digital Press, 2001, ISBN 978-1555582418, 320 pages
- [3] A. Mocanu, M. Velicanu, *Building a Spatial Database for Romanian Archaeological Sites*, Database Systems Journal, Vol. II, ISSN: 2069 – 323, p. 3-12.
- [4] <http://www.sql-server-performance.com/2010/logical-reads/>
- [5] <http://itknowledgeexchange.techtarget.com/sql-server/back-to-basics-clustered-vs-nonclustered-indexes-whats-the-difference/>
- [6] <http://www.devtoolshed.com/content/clustered-index-vs-non-clustered-index-sql-server>
- [7] <http://blogs.msdn.com/b/craigfr/archiar/2006/06/30/652639.aspx>
- [8] C. Churcher, *Beginning Database Design. From novice to professional*, Apress, 2007, ISBN 1-59059-69-9, 267 pages



Cecilia CIOLOCA has graduated the Faculty of Economic Cybernetics, Statistics and Informatics from the Bucharest Academy of Economic Studies in 2008. She is currently conducting doctoral research in Economic Informatics at the Academy of Economic Studies. She is experienced in software development using open source technologies. She has successfully completed several projects using various programming languages and open source frameworks. She is proficient in using Java Enterprise Edition.



Mihai GEORGESCU has graduated the Faculty of Electronics, Telecommunications and Information Technology – Politehnica University Bucharest in 2005 and also completed a master program at the Faculty of Economic Cybernetics, Statistics and Informatics from the Bucharest Academy of Economic Studies in 2007. He has experience in the development of software products using various programming languages like C/C++, C# both Windows Forms and Web.