DOI: 10.5604/01.3001.0010.5210

OPTIMISATION ANALYSIS OF TRANSACT-SQL QUERIES BASED ON INDEXES

Dominika Hodun, Maria Skublewska-Paszkowska
Lublin University of Technology, Institute of Computer Science

***Abstract**. The article presents a discussion of the optimisation methods involving the modification of query syntax on the example of the use of indexes that served for the application of the methods in practice, and the presentation and analysis of the test results and their comparison. All the queries were made in a Microsoft SQL Server Management Studio 2014 environment, using the AdventureWorksDW2012 test database provided by Microsoft. The object of the research were SELECT queries, which consequently returned the desired set of output data.*

Keywords: Transact-SQL, optimization, queries

ANALIZA OPTYMALIZACJI KWEREND JĘZYKA TRANSACT-SQL OPARTYCH O INDEKSY

***Streszczenie.** Artykuł obejmuje omówienie metod optymalizacji polegających na modyfikacji składni zapytań na przykładzie wykorzystania indeksów, które posłużyły do zastosowania metod w praktyce, a także przedstawienie i analizę otrzymanych wyników badań i ich porównanie. Wszystkie zapytania zostały wykonane w środowisku Microsoft SQL Server Management Studio 2014 przy użyciu bazy testowej, udostępnionej przez firmę Microsoft, AdventureWorksDW2012. Obiektem badań były zapytania SELECT, które w rezultacie zwracały pożądany zbiór danych wynikowych.*

Słowa kluczowe: Transact-SQL, optymalizacja, zapytania

Introduction

Optimisation techniques involving the modification of query syntax are based on two types of methods: universal ones, independent of the type of database used, and ones specific to individual database providers, described in detail in the documentation supplied by the manufacturers of the software [1]. Their application allows, among other things, the use of appropriate indexes, transmission of relevant data to the cost-based optimiser, as well as determination of the optimal order of joins, executions of subqueries and outer queries. Prior determination of the proper execution plan makes it easier to practise in order to achieve it, and allows the use of the simplest tools. The aim of the article is to present the query optimisation in the Transact-SQL language, based on selected queries using indexes. The optimisation analysis was performed in Microsoft SQL Server Management Studio 2014 using the AdventureWorks2012 test database provided by Microsoft.

1. Optimisation methods

There are two types of method classification for optimising queries in Transact-SQL. The first of these divides optimisation methods into static and dynamic ones. Static optimisation aims at establishing a "most favourable" execution plan for the query before the start of its implementation. The plan does not have to change during the execution of the query – hence its name. Dynamic optimisation consists in finding the "most favourable" execution plan for the query before the query execution, but in the course of its implementation the plan may change. Currently, database systems provide only static optimisation, although its effectiveness is generally lower compared to that of dynamic optimisation, which is much more expensive [4].

The second type of classification distinguished divides optimisation methods into single query one, and simultaneous optimisation of multiple queries. In the case of single query optimisation, only one query is optimised at the same time. In simultaneous optimisation of multiple queries, the partial results of a single query can be used by other queries, leading to a reduction in the execution time of a set of queries.

Current database systems offer only single query optimisation.

2. Indexes

Indexes are database structures enabling faster data references, with a high impact on system performance, but not increasing its correct operation. Some indexes are created automatically, e.g. while creating a primary key. If the table does not have an index, its data form a pile – a disordered collection of pages that belong to the table. Table scan operations are very expensive and require frequent references to the data on the disk, because they rely on searching all sides – it is impossible to tell whether all the records with specified search conditions have been found until the last row of the table is reached [2]. This mechanism is extremely suboptimal. Table scans can be avoided thanks to indexes, which provide quick access to data while searching the table. Indications for the use of the indexing mechanism are cases of receiving as a result of a query execution repeatably less than 15% of the records from a large size table, or the presence of columns used to join multiple tables, as shown in figures 1 to 7.

3. Research methodology

Analysis of optimisation was performed in the Microsoft SQL Server Management Studio 2014 environment, using the AdventureWorks2012 test base. Database queries were tested that used indexes. The tests were conducted using a clustered and a nonclustered index. The analysis involved queries without an index on the table-joining columns and with indexes, as well as using the leading, other than the leading and all the columns of an index as a selection condition.

Query time was measured using the command SET STATISTICS TIME ON [5]. The values that can be read from the returned statistics are parsing and compilation time and execution time.

Comparing the two queries, the desired action is to measure the full time of their execution – from making an inquiry to obtaining the result, not just another call from the cache of already processed requests. Thus, an important element is clearing the cache and Microsoft SQL Server Management Studio buffers. To prepare the environment for meaningful tests, two procedures should be used: DBCC FREEPROCACHE and DBCC DROPCLEANBUFFERS [2], to do with cleaning the cache of plans and buffers.

The time and readings statistics are switched directly in the connection session using the commands SET STATISTICS IO ON and SET STATISTICS TIME ON [5]. Automatic updating is performed during query optimisation by the query optimiser when the option Auto update statistics is switched on. A manual update can be called by the command UPDATE STATISTICS [5]. It must be carried out, among other things, when creating a new index, before inserting data into a table if the table was truncated or when inserted a large number of records is inserted into the table (especially with a small amount of data) directly used in queries. Statistics can be viewed using the DBCC SHOW_STATISTICS command.

3.1. Analysis of query execution with clustered and nonclustered indexes

Listing 1 shows the query that returns all records with the place name "Monroe". The query plan is illustrated in figure 1.

Listing 1. Query with a clustered index

```
SELECT * FROM dbo.DimGeography WHERE City='Monroe'
```

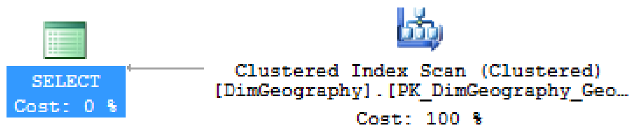
SELECT
Cost: 0 %
Clustered Index Scan (Clustered)
[DimGeography].[PK_DimGeography_Geo…
Cost: 100 %

Fig. 1. Plan of query execution shown in Listing 1 using a clustered index

Listing 2. Statistics of query execution using a clustered index

```
Table 'DimGeography'. Scan count 1, logical reads
27.
CPU time = 0 ms,  elapsed time = 49 ms.
```

With the command CREATE INDEX ON IX_Geography dbo.DimGeography (City) a nonclustered index was created, containing one column. The execution of the query shown in Listing 1, using the newly created index, allows to get the results shown in figure 2.

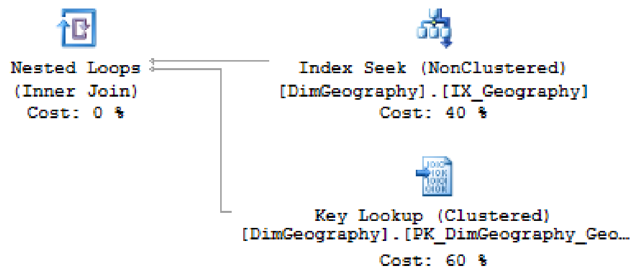
Nested Loops
(Inner Join)
Cost: 0 %
Index Seek (NonClustered)
[DimGeography].[IX_Geography]
Cost: 40 %
Key Lookup (Clustered)
[DimGeography].[PK_DimGeography_Geo…
Cost: 60 %

Fig. 2. Plan of query execution shown in Listing 1 using a nonclustered index

Listing 3. Statistics of query execution using a clustered and a nonclustered index

```
Table 'DimGeography'. Scan count 1, logical reads
6.
CPU time = 0 ms,  elapsed time = 36 ms.
```

Table 1. Comparison of the statistics of query execution using a nonclustered index

Type of query	Number of scans (jointly)	Number of readings from memory (jointly)	Time of using the processor [ms]	Time of query execution [ms]
Query with a clustered index	1	27	0	49
Query with a nonclustered index	1	6	0	36

Analysis of the statistics of query execution using a clustered and nonclustered index shows that, for reading data from a table containing disordered values, the more favourable was the use of a nonclustered index referring to individual records (Table 1). The nonclustered index does not transfer data between the sides of the data heap, which results in a smaller number of readings from memory, and hence, shorter query execution.

3.2. Analysis of query execution with and without an index on the table-joining columns

Listing 4 shows the query that returns all records from the table DimEmployee, for which the value SalesTerritoryKey occurs in the table DimSalesTerritory. The query plan is illustrated in figure 3.

Listing 4. Zapytanie bez wykorzystania indeksu na kolumnach łączenia tabel

```
SELECT * FROM dbo.DimEmployee de
INNER JOIN dbo.DimSalesTerritory dst ON
dst.SalesTerritoryKey =
de.SalesTerritoryKey
```

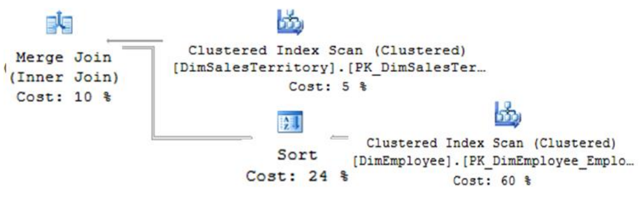
Merge Join
(Inner Join)
Cost: 10 %
Clustered Index Scan (Clustered)
[DimSalesTerritory].[PK_DimSalesTer…
Cost: 5 %
Sort
Cost: 24 %
Clustered Index Scan (Clustered)
[DimEmployee].[PK_DimEmployee_Emplo…
Cost: 60 %

Fig. 3. Plan of query execution shown in Listing 4 without using an index on the table-joining columns

Listing 5. Statistics of query execution without using an index on the table-joining columns

```
Table 'DimEmployee'. Scan count 1, logical reads
49.
Table 'DimSalesTerritory'. Scan count 1, logical
reads 3.
CPU time = 15 ms,  elapsed time = 502 ms.
```

With the command CREATE INDEX IX_DimEmployee_SalesTerritoryKeyON dbo.Dim Employee (SalesTerritoryKey) a nonclustered index was created, containing one column. The execution of a simplified query using the newly created index allows to get the results shown in figure 4.

Listing 6. Query using an index on the table-joining columns

```
SELECT * FROM dbo.DimEmployee de WITH
(INDEX(IX_DimEmployee_SalesTerritoryKey))
INNER JOIN dbo.DimSalesTerritory dst ON
dst.SalesTerritoryKey = de.SalesTerritoryKey
```

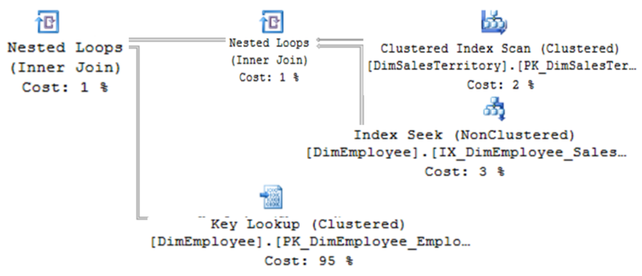
Nested Loops
(Inner Join)
Cost: 1 %
Nested Loops
(Inner Join)
Cost: 1 %
Clustered Index Scan (Clustered)
[DimSalesTerritory].[PK_DimSalesTer…
Cost: 2 %
Index Seek (NonClustered)
[DimEmployee].[IX_DimEmployee_Sales…
Cost: 3 %
Key Lookup (Clustered)
[DimEmployee].[PK_DimEmployee_Emplo…
Cost: 95 %

Fig. 4. Plan of query execution shown in Listing 6 using an index on the table-joining columns

Listing 7. Statistics of query execution using an index on the table-joining columns

```
Table 'DimEmployee'. Scan count 11, logical reads
641.
Table 'DimSalesTerritory'. Scan count 1, logical
reads 3.
CPU time = 0 ms,  elapsed time = 332 ms.
```

Analysis of query execution statistics using an index established on the table-joining columns and without using it, shown in Listings 4 and 6, illustrates the benefits of setting up such an index, allowing acceleration of the joining operations (Table 2). The time to execute the query using the index on the table-joining columns is almost twice as low as without the use of an index. Lack of indexes on table-joining columns can lead to less efficient algorithms for the implementation of the joining.

Table 2. Comparison of the statistics of query execution using an index on the table-joining columns and without using an index on the table-joining columns

Type of query	Number of scans (jointly)	Number of readings from memory (jointly)	Time of using the processor [ms]	Time of query execution [ms]
Query without an index on the table-joining columns	2	52	15	502
Query with an index on the table-joining columns	12	644	0	332

For the use of the index to be effective, it is necessary to define a sufficiently selective condition for the first (or only) index column. The condition must be expressed in such a way that the database might determine a suitably narrow range of index values. If the range is too large, the optimiser may consider that it is not worth using the index and appoint another path to accessing the data.

There are two most commonly used types of indexes: clustered and nonclustered [2]. A clustered index stores data in ascending order, according to the index key (an indicated column or columns), while the nonclustered index is a separate object of the database, indicating individual records in the table, but without regard to how they are stored. This means that the creation of clustered indexes is important for columns on the basis of which the data read from the table are sorted or data sets are returned. On the other hand, creating nonclustered indexes matters for tables with various values, used to connect or search for data. An important issue when deciding whether to create a clustered index is the choice of the appropriate column or columns, as well as their sequence [2].

3.3. Analysis of queries using the leading, some other than the leading, and all the columns of the index as a selection condition

With the command CREATE INDEX ON IX_Geography dbo.DimGeography (City, PostalCode) a nonclustered index was created, containing two columns. Implementation of a simplified query using the newly created index allows to get the results presented in figure 5.

Listing 8. Query using the leading column of the index as a selection condition

```
SELECT * FROM dbo.DimGeography WITH (INDEX
(IX_Geography))
WHERE City = 'Monroe'
```

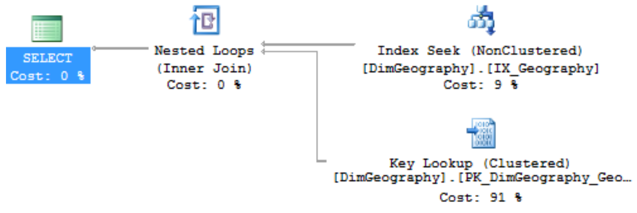
SELECT
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 0 %
Index Seek (NonClustered)
[DimGeography].[IX_Geography]
Cost: 9 %
Key Lookup (Clustered)
[DimGeography].[PK_DimGeography_Geo...
Cost: 91 %

Fig. 5. Plan of query execution shown in Listing 8 using the leading column of the index as a selection condition

Listing 9. Statistics of query execution using the leading column of the index as a selection condition

```
Table 'DimGeography'. Scan count 1, logical reads
30.
CPU time = 0 ms,  elapsed time = 87 ms.
```

The query result derives significant benefits from the use of the leading column of the index in a table search operation. If instead of the leading column another column of the index is used, the results obtained deteriorate.

Listing 10 shows a query that returns all the records where the postal code is "98272". The query plan is illustrated in figure 6.

Listing 10. Query using some other than the leading column of the index as a selection condition

```
SELECT * FROM dbo.DimGeography WITH (INDEX
(IX_Geography))
WHERE PostalCode = '98272'
```

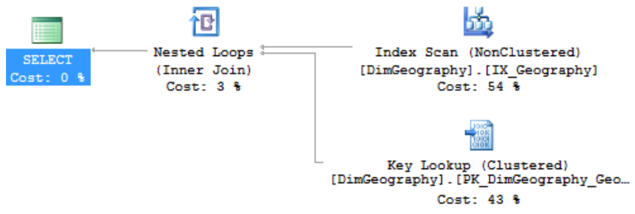
SELECT
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 3 %
Index Scan (NonClustered)
[DimGeography].[IX_Geography]
Cost: 54 %
Key Lookup (Clustered)
[DimGeography].[PK_DimGeography_Geo...
Cost: 43 %

Fig. 6. Plan of query execution shown in Listing 10 using some other than the leading column of the index as a selection condition

Listing 11. Statistics of query execution using some other than the leading column of the index as a selection condition

```
Table 'DimGeography'. Scan count 1, logical reads
10.
CPU time = 0 ms,  elapsed time = 40 ms.
```

Listing 11 shows a query that returns all the records where the postal code is "98272", but with forcing the use of the index created. The query plan is illustrated in figure 7.

Listing 12. Query using all the columns of the index as a selection condition

```
SELECT * FROM DimGeography WITH (INDEX
(IX_Geography))

WHERE City = 'Monroe'
AND PostalCode = '98272'
```

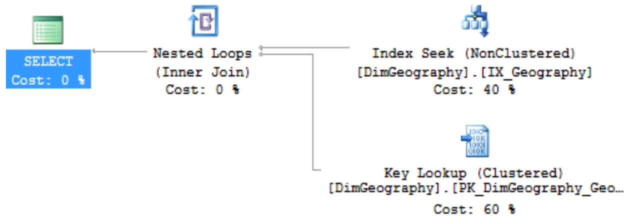
SELECT
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 0 %
Index Seek (NonClustered)
[DimGeography].[IX_Geography]
Cost: 40 %
Key Lookup (Clustered)
[DimGeography].[PK_DimGeography_Geo...
Cost: 60 %

Fig. 7. Plan of query execution shown in Listing 12 using all the columns of the index as a selection condition

Listing 13. Statistics of query execution using all the columns of the index as a selection condition

```
Table 'DimGeography'. Scan count 1, logical reads
6.
CPU time = 15 ms, elapsed time = 35 ms.
```

Table 3. Comparison of the statistics of query execution using the leading, some other than the leading, and all the columns of the index as a selection condition

Type of query	Number of scans (jointly)	Number of readings from memory (jointly)	Time of using the processor [ms]	Time of query execution [ms]
Query using the leading column of the index as a selection condition	1	30	0	87
Query using other than the leading column of the index as a selection condition	1	10	0	40
Query using all the columns of the index as a selection condition	1	6	15	35

Analysis of the statistics of query execution with the leading column of the index, some other than the leading column of the index, and all the columns of the index as a condition of selection, illustrated in Listings 8, 10 and 12, shows how important the order of the columns is in the index (Table 3). Results of implementation of the first query are slightly worse than the results of the second question, it should however be noted that the first query returned 14 rows, and the second just 2. The best results can be obtained using the column index in the order they were placed in the declaration of the index, as shown in the last of the queries.

The key index should be as short as possible, which will allow placing a greater number of index entries on a single page, which in turn reduces the number of pages of the whole index. One should also take into account the fact that it should not be allowed to modify the columns of the clustered index key, because this results also in the modification of the index pages and forces a re-ordering of the data pages.

4. Conclusions

The aim of the article was to present the query optimisation in the Transact-SQL language on the example of selected queries using indexes.

Before using the index one should estimate its utility as part of a specific inquiry. Evaluation of usability can be carried out on the basis of index statistics in conjunction with information about the number of records in the table and on the distribution of instances of individual values or their ranges contained in the indexed column. Statistics play an important role in obtaining high-performance queries. Indexes to accelerate the search and scaling of database systems will be used properly only when the query optimiser has current statistics on the data in tables and their distribution. One should remember to update the statistics. This is important, because with outdated data the query optimiser may operate suboptimally.

Analysis of the statistics of query execution using a clustered and nonclustered index revealed that for reading data from a table containing disordered values the more favourable was the use of a nonclustered index referring to individual records (Table 1). A nonclustered index does not transfer data between the sides of the data heap, which results in a smaller number of readings from memory, and hence, shorter query execution.

Analysis of the statistics of query execution using an index established on the table-joining columns and without using such an index, shown in Listings 4 and 6, illustrated the benefits of setting up the index, allowing acceleration of the joining operations (Table 2). The query execution time using an index is almost twice as low as that without the use of an index. Lack of indexes on the table-joining columns can lead to less efficient algorithms for the implementation of the joining.

Analysis of the statistics of query execution from the leading column, some other than the leading column, and all the columns of the index as a selection condition, shown in Listings 8, 10 and 12, showed how important the order of the columns is in the index (Table 3). The best results can be obtained using the index columns in the order they were placed in the declaration of the index.

Bibliography

[1] Ben-Gan I.: Microsoft SQL Server 2012. Podstawy języka T-SQL. APN Solid, 2012.
[2] Bertrand A., Stellato E., Berry G., Hall J., Sack J., Kehayias J., Kline K., Randal P., White P.: High performance techniques for Microsoft SQL Server. SQL Sentry. , Tom 1-5, 2013.
[3] Morzy T.: Optymalizacja zapytań. http://wazniak.mimuw.edu.pl/images/c/c7/BD-2st-1.2-w12.tresc-1.1.pdf
[4] Nevarez B.: Microsoft SQL Server 2014. Optymalizacja zapytań. Helion, 2014.
[5] Ptasznik A.: Optymalizacja zapytań SQL. Warszawska Wyższa Szkoła Informatyki, 2009.
[6] Tow D.: SQL Optymalizacja. Helion, 2014.
[7] Wojciechowski M., Zakrzewicz M.: Kosztowy optymalizator zapytań. http://www.cs.put.poznan.pl/mzakrzewicz/pubs/plsem02.pdf

M.Sc. Eng. Dominika Hodun
e-mail: dominikahodun@gmail.com

In 2016 she graduated from the Institute of Computer Science at the Faculty of Electrical Engineering and Computer Science at the Lublin University of Technology with an M.Sc. degree in engineering. Areas of interest: SQL, and in particular Transact-SQL and its optimisation, business issues in computing and ERP systems.

Ph.D. Eng. Maria Skublewska-Paszkowska
e-mail: maria.paszkowska@pollub.pl

Researcher-lecturer in the Institute of Computer Science at the Faculty of Electrical Engineering and Computer Science of the Lublin University of Technology, where she received her MSc. She obtained her PhD at the Silesian University of Technology. Her research activity is connected with: motion acquisition methods, 3D motion data analysis, 3D algorithms, mobile programming.

otrzymano/received: 14.09.2016 *przyjęto do druku/accepted: 14.08.2017*