ISSN 1330-3651 (Print), ISSN 1848-6339 (Online) https://doi.org/10.17559/TV-20200121134626
Original scientific paper

Techniques for Ensuring Index Usage Predictability in Microsoft SQL Server

Blerta HAXHIJAHA EMINI*, Jaumin AJDARI, Bujar RAUFI, Besnik SELIMI

Abstract: The demand for carrying out high-performance operations with data is growing in parallel with the vast growth of data itself. The retrieval of data for analysis, the manipulation of data, as well as its insertion in data stores must all be performed very efficiently, using techniques that ensure speed, reliability and accuracy. This paper investigates the techniques and practices that improve the performance of data retrieving by the use of SQL and Microsoft SQL Server. Being that SQL is a declarative language that specifies what should be produced as a result, instead of how to achieve that result, this paper will look at the internals of SQL Server that affect the "how" of queries and data operations, in order to propose techniques that ensure performance gains. The paper will aim to shed light on the limitations and variance in index usage, and to answer the question why indexes are sometimes used, and other times not, for the same query. To overcome the index limitations the "index fusion" technique is proposed.

Key words: database; indexes; index fusion; SQL

1 INTRODUCTION

Database operations are often responsible for a substantial portion of the delays associated with completing a computer related action on a computer program that operates on or with data. In the early days of relational databases, performance issues were extensive because of limited hardware resources and immature optimizers, so performance was a priority consideration. But even today, despite the huge growth in resources, there is even more growth in the amount of data available due to technological advances [1] and the amount and variety of connected devices [2], so performance continues to be of critical importance.

Due to the ever-growing need for performance enhancements in data operations in today's data-driven world, this paper will look at techniques that improve the performance of SQL queries executed on Microsoft SQL Server. Because indexes are recognized and widely documented to improve performance of queries executed on Microsoft SQL Server, this paper will also focus on them. It will aim to propose techniques that ensure usage predictability and remove variations in index usage that occur even when working with exactly the same queries.

The paper will look at indexes through the lens of their storage internals inside SQL Server, in order to provide explanations why they often go unused, even after being positively tested during testing stages. This is especially confusing for DBAs and/or database developers, who add indexes that are successfully used in testing environments but fail to be used in production environments for the same queries. According to this situation we can conclude the following:

- Due to the storage internals of data and indexes inside SQL Server, there are cases when the SQL Server engine decides not to use an index for a given query because the cost of usage is too high, even though for exactly the same query the index is used on other cases.
- Covering indexes is a technique that can ensure better predictability of index usage in queries, but this may lead to having too many complex indexes on server level that are costly for maintenance and storage. Therefore this paper proposes a new technique called index fusion, which aims to consolidate the indexes used on sever level and reduces their number, while still ensuring that indexes do get consistently used.

The following research questions are set in order to address the conclusions outlined above:

1. For a given query which uses the indexes defined for the columns in the WHERE clause are those indexes consistently used or there exist variations in index usage?
2. How can the limitations and variance on index usage be removed, in order to achieve better index usage predictability for critical query workloads?

We attempt to suggest techniques for removing these limitations and variance on index usage by following an experimental approach in trying to answer the questions above. Even though the query covering technique by using the INCLUDE keyword is well known and documented for performance improvement on query level, this paper goes further to propose the index fusion technique which operates on server level. The novelty of this technique is in providing index usage predictability and removing variations of index usage on server level. This is done through consolidation of indexes in an existing critical workload, instead of focusing just on performance improvement on query level. Our technique also helps reduce the total number of indexes present in the server that occupy storage space and impose the need of maintenance which is often a costly operation.

The rest of this paper is organized as follows: section Related Work provides a summary of relevant research in the area of performance improvement. The section Experimental Set-up and Implementation describes in detail the implementation and the outcome of the experiments used in our research. Finally, conclusions summarize this paper by outlining the major contributions.

2 RELATED WORK

There is an abundance of research work available focusing on performance improvement of relational database management systems, and the Microsoft SQL Server RDBMs is no exception. Numerous research papers have attempted to propose techniques and solutions towards better performance in Microsoft SQL Server, focusing on different areas for improvement.

Several papers have approached the performance improvement goal through proposing techniques for writing T-SQL queries in ways that maximize execution speed. In [3], the authors provide several recommendations for optimizing query execution in Microsoft SQL Server, including the use of temporary indexes that are created just before running specific queries and reports, and afterwards are dropped. Some further recommend the usage of stored procedures over ad-hoc queries. In order to reuse the execution plan, the authors advise using the sys.sp_executes ql system stored procedure for running ad-hoc queries.

In [4], Habimana lists several recommendations for writing efficient and faster SQL queries. The author advises to un-nest sub queries, arguing that rewriting nested queries as joins often leads towards more efficient execution. Habimana also postulates that using an 'OR' in the join condition will slow down the query by at least a factor of two.

In [5], the authors look at several sample queries written in T-SQL using different alternatives, in order to determine which alternative executes in the most efficient manner. They conclude that the difference in performance when local variables are used in scripts containing multiple queries (especially when these variables are transmitted from one query to another), and when local variables are not used, is about 50%. The authors also recommend using, where possible, the BETWEEN clause instead of the IN or OR conditionals. The reason behind this is that SQL Server 2008, in case of using the IN conditional, will access the index for several times equal to the number of values in the search. On the other hand, when using the BETWEEN clause, the index will be accessed only once, since the optimizer will turn the BETWEEN clause into a pair of $\geq$ $\leq$ conditions.

In [6], the authors propose a cost-based range estimation model that maps a top-k query into a cost-optimal range query for efficient execution. The top k query efficiency is also discussed in [7], in the context of retrieving data from different web services. Here the authors propose an approach based on two strategies: pipeline-parallel strategy and necessary invocation principle. The first one serves as an initial step to speed up the top-k query execution, whereas the second strategy or principle minimizes the number of service invocations during the execution of a composition plan, by determining if an invocation is truly necessary to provide the final top-k result.

An ambitious undertaking that focuses on database performance tuning is the AutoAdmin project by Microsoft Research, which in a nutshell, aims to make database systems self-tuning and self-administering. In scope of this project, numerous research papers have been published, and [8] makes a summary of the progress from a decade of research in self-tuning database systems, while [9] reviews the lessons learned from the AutoAdmin project at Microsoft Research up to year 2011. Another aspect of database operations automation is discussed in [10], where the authors present a tool that automates the formulation of SQL join queries when retrieving data from multiple heterogeneous and large databases.

In [11], the authors discuss the importance of parallelism in achieving high performance and propose an efficient parallel implementation scheme of relational tables, by employing an extendible multidimensional array.

Rahman in [12] proposes the usage of a scorecard tool that measures SQL query performance in order to make sure that the resource consumption of SQL queries is predictable and the database system environment is stable, in environments where hundreds of queries are run at any point in time.

Other papers have focused on indexes, recognizing them as crucial in performance improvement strategies. Several advanced indexing strategies such as R tree, Bitmap, Octree have been developed, each with its own advantages and limitations. In [13], a novel SkipNet and B+ tree based index structure is presented, which adopts a two-layer architecture. In the lower layer, it uses the B+ tree to construct an efficient local index, whereas in the upper one, it adaptively selects among local index nodes to form a SkipNet based global overlay. The index structure is claimed to efficiently support a variety of types of queries and provide high availability. In [14], a dynamic double layer indexing structure with Skipnet overlay for global indexing and an Octree index technique for local indexing has been proposed, and the authors claim it to be better than the previous double-layer indexing techniques for complex queries.

Narasayya and Syamala in their paper [15] address the performance degradation in queries that require scanning large indexes that are defragmented. They argue that the DBA task of deciding which indexes to defragment is very challenging due to the following two limitations: little support by database engines to estimate the impact of defragmenting an index on the performance of a query and the fact that defragmentation can only be performed on an entire B+ tree, which is very costly. The authors propose methods to address these limitations and they also investigate which index is most appropriate to defragment for a given workload.

Lee et al. in [16] discuss the importance of the ability to estimate the overall progress of execution of a query. This feature would be valuable to DBAs in order to decide if a long-running, resource intensive query should be terminated or allowed to run to completion. The authors also discuss the value of having progress estimates for individual operators in a query execution plan, since this can help DBAs understand and identify which operators are requiring significantly more time or resources than expected and take appropriate measures. Further, they introduce the new Live Query Statistics (LQS) feature in Microsoft SQL Server 2016, which includes the display of overall query progress as well as progress of individual operators in the query execution plan. Other relevant papers that focus on progress estimation are [17, 18].

In [19], Dziedzic et al. focus on the importance of hybrid database physical designs, which consist both of B+ tree indexes and column-store indexes. The authors argue that this hybrid design can yield better performance in several orders of magnitude. They also extend the Microsoft SQL Server Database Engine Tuning Advisor to recommend an appropriate combination of column-store and B^+ indexes for a given workload.

In [20], the authors similarly discuss the trend of using specialized systems that are optimized for either fast ACID

transaction workloads or complex analytical query workloads, but not both (thus inducing additional storage and administration overhead by keeping two separate copies of the database). The paper then introduces a hybrid DBMS architecture that efficiently supports varied workloads on the same database, thus obviating the need to maintain separate copies of the same database in independent systems.

3 EXPERIMENTAL SET-UP AND IMPLEMENTATION

3.1 Turning Point in Index Usage

Let us consider a table (Clients) (Fig. 1), which is populated with 80,000 rows and has column ClientID defined as its primary key. Microsoft SQL Server automatically adds a clustered index based on the primary key column(s) of the table.

```
CREATETABLE Clients(
   ClientIdINTIDENTITY (1,1)PRIMARYKEY,
   TaxCodeCHAR(11),
   Name NCHAR(60),
   Surname NCHAR(60),
   IsActiveTINYINT,
   RegistrationDateDATETIME,
   Address NCHAR (57),
   Phone CHAR (15))
```

Figure 1 The (Clients) table

This table also contains a non-clustered index TaxCodeNC on column TaxCode. To examine the clustered and non-clustered indexes on table Clients in terms of their depth and number of rows and pages in each level, the dm_db_index_physical_stats dynamic management view (DMV) is used, yielding the results given in Tab. 1 below:

Table 1 Indexes on table Clients

Index	Depth	Level	Rows	Pages
Clustered Index ClientPK	3	0	80000	4000
		1	4000	14
		2	14	1
Non-clustered Index TaxCodeNC	2	0	80000	212
		1	212	1

Now let us consider the following range query (Fig. 2) against table Clients:

```
SELECT c.*
FROM Clientsc
WHEREc.TaxCodeBETWEEN'01153701453'
      AND'01164967899'
```

Figure 2 A range query searching by TaxCode

This query performs a SELECT * against the Clients table for the particular client with the specified TaxCode. However, the non-clustered index contains only the non-clustered key TaxCode and the base table row lookup id (in our case, the ClientID) in its leaf pages. This means that the rest of the columns requested in the SELECT need to be retrieved from the base table-in our case the clustered index.

Indeed, the query execution plan demonstrates this (Fig. 3).The non-clustered index TaxCodeNC defined on columnTaxCode is used in an index seek, but there is also a key lookup operation performed in the clustered index to retrieve the rest of the columns.

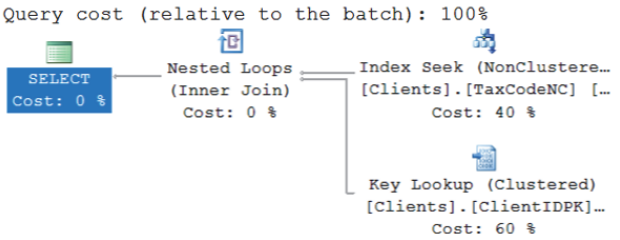

Figure 3 Execution plan of the range query searching by TaxCode

The following message is displayed for the IO statistics for this query:

Table 'Clients'. Scan count 1, logical reads 62, physical reads 34, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

There are 62 logical reads performed:

- two logical reads from the non-clustered index (the root page plus the leaf page containing the 20 records. At most, these 20 records can be scattered across two adjacent leaf pages, in which case there would be a total of three logical reads instead of two)
- $20 \times 3 = 60$ logical reads from the clustered index (for each of the 20 client IDs, the clustered index must be traversed (from root to the corresponding intermediate page, and then to the corresponding leaf page, so a total of 3 reads) in order to retrieve the rest of the columns of that client record

This sequence of operations is shown figuratively in Fig. 4 below.

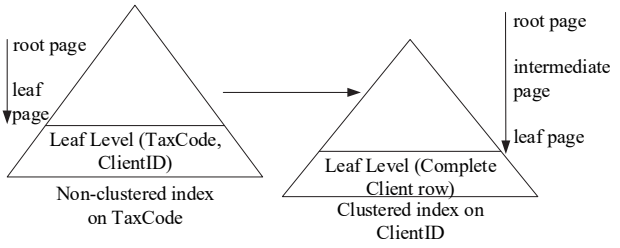

Figure 4 An index seeks followed by a key lookup

This number of logical reads is still more efficient than performing a table (clustered index) scan, which in our case would produce 4,016 logical reads (the total number of pages in the clustered index). The usage of the non-clustered index in an index seek, followed by a key lookup in the clustered index is efficient enough for the SQL Server engine to use this execution plan instead of a table scan, at least for our example query.

However, the question arises: Does this execution plan remain effective when the number of records in the result set increases even more? Does an index seek followed by a key lookup operation ever become too expensive? This will be examined next.

Let us consider a more generalized form of our range query (Fig. 5), with unknown (min) and (max) range boundaries (let us call them x and y):

```
SELECT c.*
FROM Clients c
WHEREc.TaxCodeBETWEENx ANDy
```

Figure 5 Generalized range query searching by TaxCode

The number of pages reads that need to be performed by the non-clustered index seek, followed by the clustered index key lookup, changes depending on the number of rows that fall into the (x, y) range requested by the query. As this number of resulting rows increases, the number of page reads made by the two operations described above also increases. The question arises: When this number of reads approaches the number of page reads that would be necessary if a clustered index scan was to be performed instead, does the SQL Server engine still decide to go through the two separate operations (non-clustered index seek and key lookup in the clustered index) and join their results to get all the necessary columns for the resulting records? One element that affects this decision is also the nature of these operations. In a clustered index scan, the logical reads are sequential, whereas the key lookup in the clustered index might be very random although the resulting records have sequential TaxCode values, their corresponding ClientID are probably not sequential at all and reside on different pages.

Let us next examine this by looking at the execution plan and IO statistics of the same query as before, but this time within a range for the TaxCode column which returns 1,000 rows (Fig. 6):

```
SELECT c.*
FROM Clients c
WHEREc.TaxCodeBETWEEN'60904989375'
AND'62115722473'
```

Figure 6 Range query searching by TaxCode with 1.000 resulting rows

The execution plan (Fig. 7) and the IO statistics message are given below:

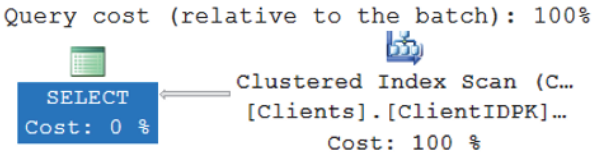

Figure 7 Execution plan of the range query searching by TaxCode with 1.000 resulting rows (1000 rows affected)

Table 'Clients'. Scan count 1, logical reads 4016, physical reads 0, read-ahead reads 3994, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

In this case, SQL Server chose to perform a clustered index scan. Although the same T-SQL query as in the previous case was used, this time the execution plan is different. Albeit this last query returned only 1,000 out of the 80,000 rows in the table (1.25% of the data in the table), still SQL Server estimated that the non-clustered index is more expensive to be used. For our sample range query searching by TaxCode, table 2 summarizes the type of operation(s) used when a different number of resulting records are returned.

SQL Server makes use of the clustered index only when the query is highly selective. It looks at the table size (in our case the clustered index size) to compare the cost of key lookups (which are random) to the cost of a table scan (which can be performed sequentially). Consequently, non-clustered indexes are only useful when a very selective set of data needs to be retrieved.

This explains why some queries sometimes have one plan, and at other times a completely different plan a situation that often puts at unease many DBAs. It also makes clear why some of the non-clustered indexes simply are not as useful as they were expected to be, and the habitual recommendation of just adding indexes for the columns referenced in the WHERE clause does not give the expected results. In fact, the addition of indexes might work well during testing, but in a production environment where the same queries provide different result sets, the indexes might just not be used. This implies that in terms of query tuning and performance optimization, other strategies should be considered than just adding an index on a column that is in the WHERE clause of queries that are most critical to performance and for which more consistent plans are needed.

Table 2 Operation used according to number of records in the result set

Query	No. of records in the result set	Operation used
SELECT c.* FROM Clients c WHERE c.TaxCode BETWEEN x AND y	1	Non-clustered Index Seek and Clustered Index Key Lookup
	20	Non-clustered Index Seek and Clustered Index Key Lookup
	350	Non-clustered Index Seek and Clustered Index Key Lookup
	800	Non-clustered Index Seek and Clustered Index Key Lookup
	1000	Clustered Index Scan
	1500	Clustered Index Scan
	10000	Clustered Index Scan
	80000	Clustered Index Scan

3.2 Index Fusion

Following our objective of proposing techniques that increase query performance in Microsoft SQL Server, we will next examine ways how to remove the limitations and variance on index usage presented in the previous section. First, we will look at the concept of covering queries and examine if and how it can impact index usage in queries.

In order to understand covering, a short re-statement of our non-clustered index structure should be made. The non-clustered index on TaxCode was made of two levels the root and leaf levels. The leaf level records contained the non clustered index key TaxCode as well as the key of the clustered index ClientId. The latter was then used in key lookup operations into the clustered index when it was necessary to retrieve the rest of the columns of the record. But what if going to the clustered index to perform the expensive random lookups was to become unnecessary? Then it would suffice for the non-clustered index to be traversed in a seek operation, and potentially SQL Server would not switch to the clustered index scan that reads the complete table.

Our existing non-clustering index covers queries that

require only TaxCode and ClientId to be returned as these fields are stored in the non-clustered index itself. But SQL Server offers a technique to include other columns of the record in the page leaves of the non-clustered index as well. This is done via the keyword INCLUDE, after which the columns that we wish to be included in the index are specified. With this, we can ensure that for critical queries that require a specific collection of record columns to be returned in the SELECT part, those columns are included in the index itself. When they are included in the non-clustered index, there is no need to go to the base table and do a key lookup operation to retrieve those necessary columns (hence the term "covering index" or "covered query").

Covering indexes is an excellent technique made available by SQL Server for ensuring better index usability and performance improvements. However, even though this technique might seem very attractive and simple to implement, it is not a good approach to use it for every single query in the workload. The reason is obvious: too many indexes become costly during data modifications as well as during index maintenance. They also waste memory and disk space (including back-up space).

So, what techniques should be followed then, in order to achieve better, more predictable usage of indexes in queries and thus improve performance?

```
--Search only active clients by name and
surname
        SELECT c.*,dt.DocumentTypeId
FROM Clients cJOINClientDocuments
cdONc.ClientId=ClientIdJOINDocumentTypes
dtONcd.DocumentTypeId=dt.DocumentTypeId
WHEREc.Surname=N'Haxhijaha'ANDc.Name=N'Blerta'A
NDc.IsActive= 1;
--Search clients by surname only and retrieve
their contact details
SELECT c.*, cc.*
FROM Clients
cJOINClientContactsccONc.ClientId=cc.ClientIdJO
INContactTypesctONcc.ContactTypeId=ct.ContactTy
peId
WHEREc.Surname=N'Aliu';
--Search clients by surname in a specific
range; retrieve name, surname and phone
SELECTc.Surname,c.Name,c.Phone
FROM Clients c
WHEREc.SurnameLIKE'[A-E]%'
ORDERBYc.Surname,c.Name;
--Search clients by surname in a specific
range; retrieve name, surname and tax code
SELECTc.Surname,c.Name,c.TaxCode
FROM Clientsc
WHEREc.SurnameLIKE'[A-E]%'
ORDERBYc.Surname,c.Name;
```

Figure 8 Most critical queries identified

We already discussed what is NOT the best technique attempting to add indexes for every column in the WHERE clause. Depending on the result set, they might not even get used, even though the query is written in the same way as in the testing stages, when those indexes were shown to be used. We then presented the concept of covering indexes, which although powerful and relatively simple to implement, is not a good solution if for every single query a new covering index is introduced. If this practice is followed, there will soon be too many indexes which are costly to maintain during inserts, updates, deletes, as well as during index maintenance. Those indexes also cost in terms of disk space, in caching, and in backups.

So, let us try to answer our question by turning to the analysis of the following queries, which we assume were defined asmost critical during the workload analysis on the database (Fig. 8):

If all the database tables used in these critical queries are clustered structures without any non-clustered indexes defined, then the queries would all perform clustered index scans against the relevant tables. To avoid this, we could simply add an index for each of the different column combinations used in these critical queries. When these indexes are added, we can check from the execution plan of the critical queries that they are indeed utilized (Fig. 9). Tab. 3 summarizes the details of the four new indexes added, including the index depth and the total page numbers occupied by them, as returned by the dm_db_index_physical_stats DMV.

The total number of pages used by the four indexes is 9,365 pages, each at 8 KB. In megabytes, this is 74.92 MB of storage. This number might not look too problematic in today's systems with a lot of memory available, but this number becomes much higher when dealing with tables that have millions or hundreds of millions of records stored in them.

Table 3 Columns, depth, and page numbers of the newly created indexes

Index Name	Index Columns	Depth	Pages
SurnameNC	Surname	3	1320
FullNameStatusNC	Surname, Name, IsActive	4	2597
FullNameInclude PhoneNC	Surname, Name (INCLUDE Phone)	4	2768
FullNameInclude TaxCodeNC	Surname, Name (INCLUDE TaxCode)	4	2680

Even though we managed to tune the individual queries in terms of index usage, this was not an adequate tuning at server level. As already discussed, adding indexes for all WHERE clause columns in queries costs in terms of maintenance, storage space, cache, backups, etc. Additionally, it can be observed that the redundancy of certain columns has increased: now there exist four copies of the Surname column and three copies of the Name column in these indexes. So, is there a better technique?

This paper argues that there indeed is, and this technique revolves around "consolidating" indexes, i.e. fusing them together so that one or a few indexes do the work of many. These consolidated indexes might be slightly larger in structure, but as we will show in the next section, eventually they will take less space than the combined space used by the indexes they substitute.

Index consolidation as a process should take place after the process of tuning on query level. In our paper so far, we have performed performance tuning on query level. We did this by focusing on how to improve the execution of the individual queries, assuming that those queries were identified as critical in our workload. We created indexes that benefited these individual queries, whereby we also hinted that this might potentially create redundancy, in terms of certain index columns appearing on several indexes. Now we are ready to move on to the next level-

performance tuning at server level, or as we call it the index fusion process. We will aim to show how this process can also address the redundancy in repeating columns and the high cost of maintenance when numerous indexes are created in the server.

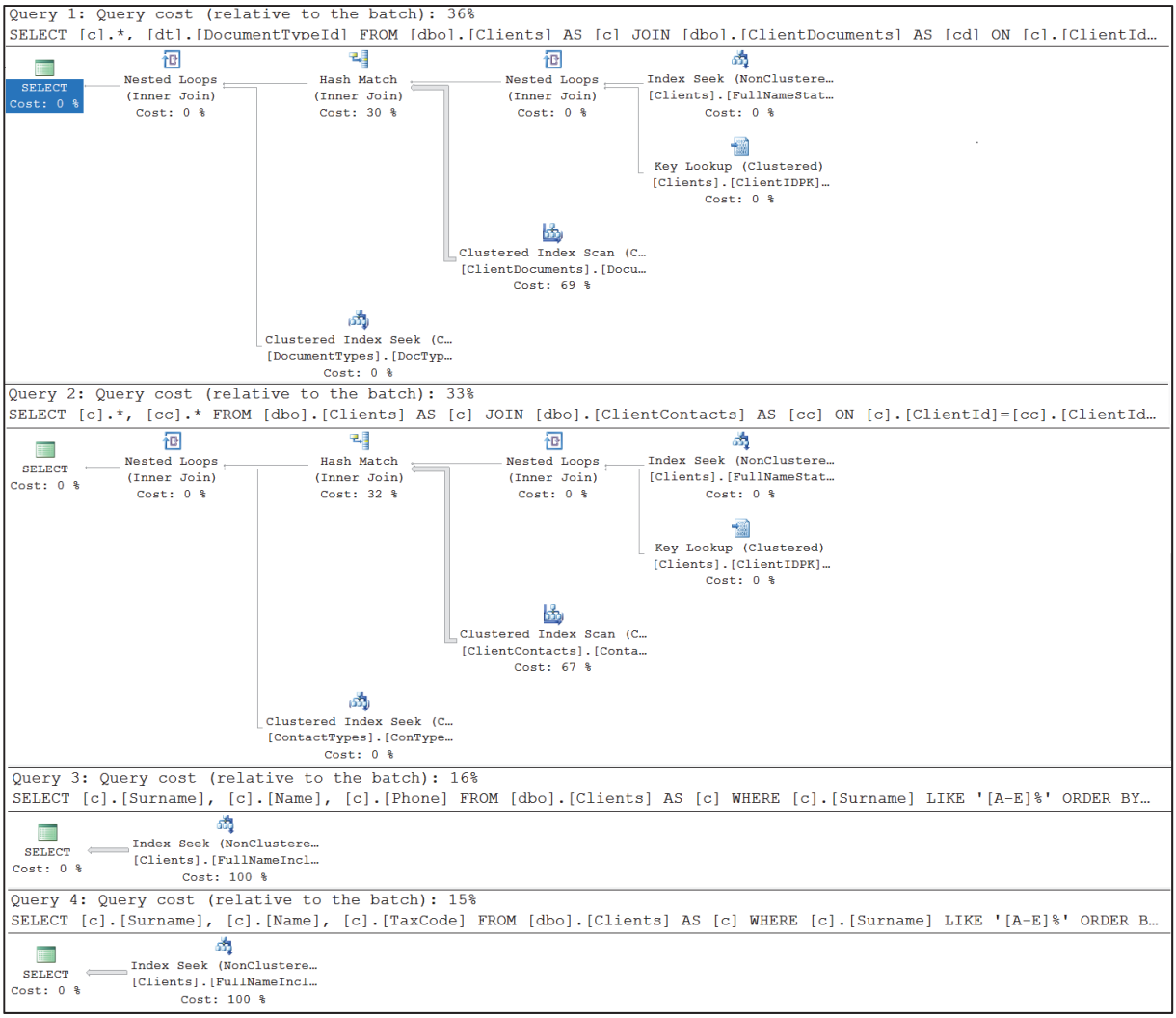

Figure 9 Execution plan of the critical queries after the creation of the four new indexes

In order to describe the index fusion process, we will continue to use our existing example, where the indexes for the individual critical queries were identified as displayed in Tab. 3. We will try to define a new, "super" index that will substitute the existing ones. Looking at the four indexes from Tab. 3, we can observe that column Surname is a left based subset of all the index keys. This suggests that Surname should be the first (i.e. left-most) column of our new index. The key columns that remain to be included are the Name and IsActive columns, because they are used in index FullNameStatusNC.

Therefore, up to this point, we have identified the key column of our new index to be (Surname, Name, IsActive). Looking at the list of included columns of the separate indexes, we observe that in our new index, we must include columns Phone and TaxCode because they are used in indexes FullNameIncludePhoneNC and FullNameInclude-TaxCodeNC. Unlike the key columns which must retain the left-based order, the order of the included columns in the new index is not important. Finally, our new consolidated index can be created as follows (Fig. 10):

```
CREATEINDEXConsolidatedIndexNC
ON Clients(SurName, Name,IsActive)
INCLUDE (Phone,TaxCode);
```

Figure 10 Consolidated index creation

In order to test our new consolidated index, we disable the four existing indexes from Tab. 3. Disabling is recommended over deletion during the testing stage. Then, if after thorough testing we conclude that the new index is utilized as expected, the "old" indexes it substitutes can be deleted from the server.

We execute the same set of critical queries given in Fig. 8 and observe their execution plan, which is given in Fig. 11 below and shows that indeed, the new index ConsolidatedIndexNC is used in all four critical queries. So now the server uses a single non-clustered index instead of the four different non-clustered indexes it was utilizing previously. Although those non-clustered indexes were all suitable, having only one index to maintain is highly preferred because of all the different costs associated with indexes even though this new index is bigger and may require a few more IOs. Furthermore, we also resolved the issue of redundancy: we do not have several copies of the

same columns present in different indexes any more.

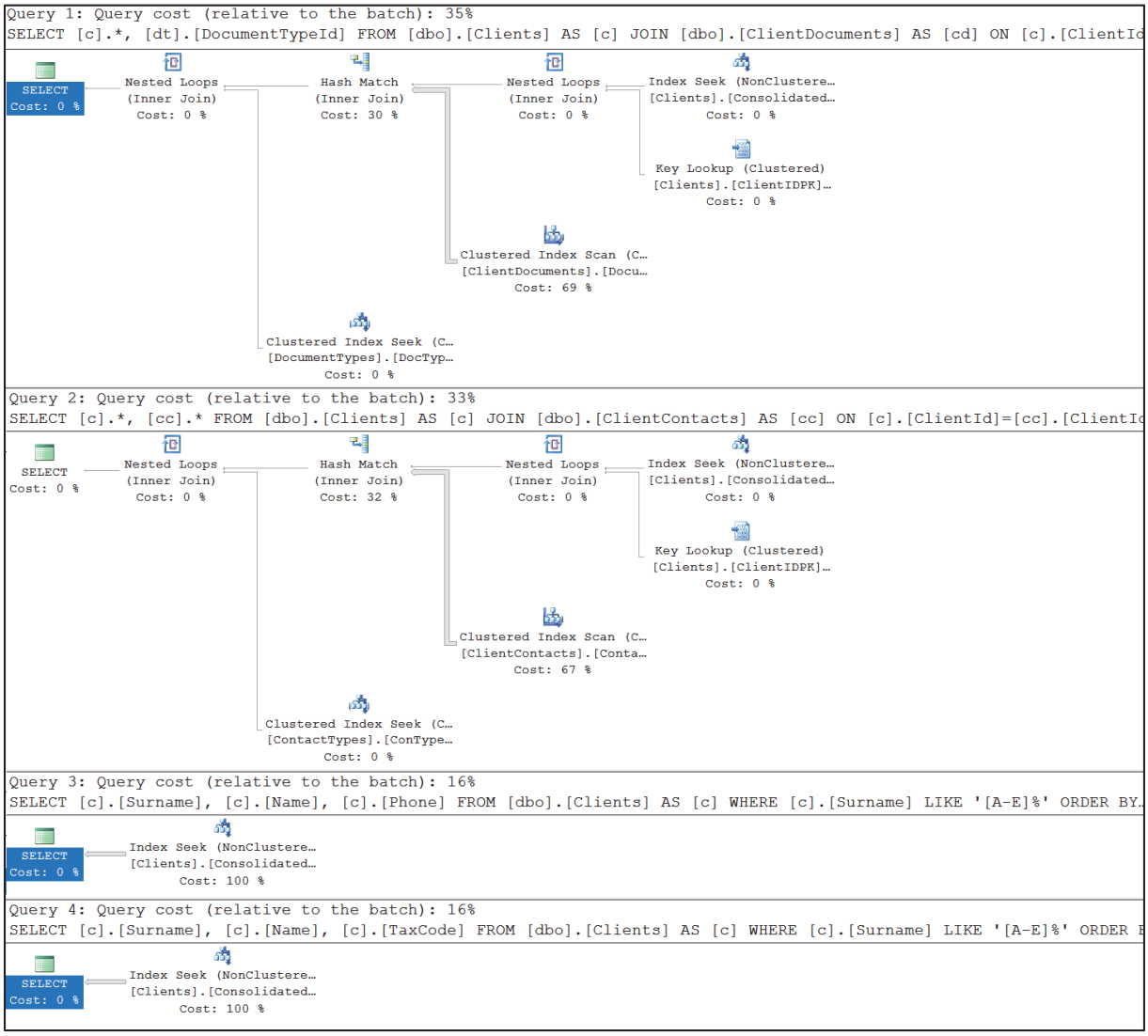

Figure 11 Execution plan of critical queries after creation of the consolidated index

The dm_db_index_physical_stats DMV for the ConsolidatedIndexNC reveals that this index contains a total of 2,863 pages, which at 8K each, calculate to 22.9 MB of storage. The storage space occupied by the four previous indexes was 74.92 MB, so the storage space was reduced to less than a third of the initial space (Fig. 12). The numbers in this example might not seem to make a big difference, but they do if we consider larger environments, with tables that are in the hundreds of gigabytes or even terabytes in size. Reducing indexes in such environments by a third of their size, could mean great savings in terms of disk space, memory, logging, fragmentation, and index maintenance in general. On the other hand, the IO statistics for our test queries reveal that the usage of our single consolidated index continues to ensure as low logical page reads of table Clients as in the case of the dedicated indexes, and certainly much lower than when no index is present (Tab. 4).

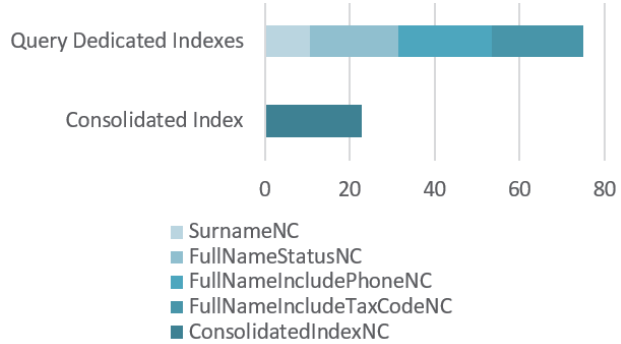

Figure 12 Total size of indexes in MB

A generalized approach for applying the index fusion technique can be summarized as follows:

1. The critical workload is identified (i.e., the set of SELECT queries that are most often executed on the server, and/or the performance of which is most crucial to be improved).
2. The critical workload is identified (i.e., the set of SELECT queries that are most often executed on the server, and/or the performance of which is most crucial to be improved).

3. The critical workload is identified (i.e., the set of SELECT queries that are most often executed on the server, and/or the performance of which is most crucial to be improved).
4. Optimal indexes for each query in the critical workload are identified (i.e. one index is identified as most suitable for each query).
5. The identified optimal indexes are then "fused" into one or more indexes (depending on the number of tables), by using the left-based subset approach for the key, and adding the columns which appear in the SELECT statement via the INCLUDE keyword.
6. The newly fused index(es) are tested to check if they are consistently used regardless of the number of rows in the resulting set of the SELECT queries.
7. If testing for usage consistency is satisfactory, the individual indexes with keys already present in the fused index(es) can be removed from the server.

Table 4 Number of logical page reads of table Clients per query-index combination

	No Index	Respective Dedicated Index	Consolidated Index
Query 1	4016	7	7
Query 2	4016	16	16
Query 3	4406	1050	1086
Query 4	4406	1017	1086

4 CONCLUSIONS

As the reliance on digitally stored data is becoming ever more pervasive in today's technology-driven world, and the amount of data available is also increasing at a fast pace, the task of efficiently retrieving and manipulating data is becoming ever more important and challenging.

This paper focused on indexes, which although being one of the most common measures used for performance improvement in data retrieval operations, still pose challenges to DBAs because of their not always predictable use on production environments.

This paper looked at that variance in index usage for several test queries, which, depending on the result set, were either using the available indexes or were opting for a table scan. The variance in index usage was observed even when nothing was changed in the way the investigated T-SQL query was written. The reason behind this variance lies in the increased cost of the usage of non-clustered index seek operations, when they need to be followed by a key lookup on the clustered table as well. These two operations together are sometimes higher in cost than performing a table scan in the first place, depending on the number of rows returned by the query. The results of these experiments can shed some light in understanding why indexes, while tested successfully in testing environments, often fail to be utilized in production.

Next, the paper proposed the index fusion technique in order to overcome the unpredictability and variance in index usage during query executions. The index fusion technique ensures better index usage consistency on server level. The experiments also demonstrated that using the index fusion technique, the overall number of indexes on a server can be reduced, thus also saving on disk space, memory, logging, fragmentation issues; making the overall index maintenance easier. In order to perform index fusion and have these gains at server level, the process to follow must be based on a thorough analysis of the current indexes and of the critical workload on that server. Analysing the two in conjunction with the knowledge on the index internals ensures successful index consolidation at server level.

5 REFERENCES

[1] Spolaôr, N., Lee, H. D., Takaki, W. S., & Wu, F. C. (2015). Feature Selection for Multi-label Learning: A Systematic Literature Review and Some Experimental Evaluations. *International Journal of Computational Intelligence Systems*, *8*(2), 3-15. https://doi.org/10.1080/18756891.2015.1129587

[2] Pereira, D. A., Morais, W. O., & Freitas, E. P. (2017). NoSQL real-time database performance comparison. *International Journal of Parallel, Emergent and Distributed Systems*, *33*(2), 144-156. https://doi.org/10.1080/17445760.2017.1307367

[3] Corlăţan, C. G., Lazar, M., Luca, V. D., & Petricică, O. T. (2014). Query Optimization Techniques in Microsoft SQL Server. *Database Systems Journal*, *5*, 33-48.

[4] Habimana, J. D. (2015). Query Optimization Techniques - Tips for Writing Efficient and Faster SQL Queries. *International Journal of Scientific & Technology Research*, *4*, 22-26.

[5] Lungu, I. C., Mercioiu, N., & Vlăducu, V. (2010). Optimizing Queries InSql Server 2008. *Scientific Bulletin Economic Sciences*, *9*, 103-108.

[6] Ayanso, A., Goes, P. B., & Mehta, K. (2009). A Cost-Based Range Estimation for Mapping Top-k Selection Queries over Relational Databases. *Journal of Database Management*, *20*(4), 1-25.

[7] Malki, A., Benslimane, S., Malki, M., Barhamgi, M., & Benslimane, D. (2020). Top-k query optimization over data services. *Future Generation Computer Systems*, *113*, 1-12. https://doi.org/10.1016/j.future.2020.06.052

[8] Chaudhuri, S. & Narasayya, V. (2007). Self-Tuning Database Systems: A Decade of Progress. *Proceedings of the 33rd International Conference on Very Large Data Bases*, 3-14. VLDB Endowment.

[9] Bruno, N., Chaudhuri, S., König, A., Narasayya, V., Ramamurthy, R., & Syamala, M. (2011). AutoAdmin Project at Microsoft Research: Lessons Learned. *IEEE Data Eng. Bull.*, *34*, 12-19.

[10] Bagui, S. & Loggins, A. (2009). Generating Join Queries for Large Databases and Web Services. *International Journal of Information Technology and Web Engineering* (*IJITWE*), *4*(2), 45-60. https://doi.org/10.4018/jitwe.2009092203

[11] Azharul, H. K. M., Tsuji, T., & Higuchi, K. (2006). A Parallel Implementation Scheme of Relational Tables Based on Multidimensional Extendible Array. *International Journal of Data Warehousing and Mining* (*IJDWM*), *2*(4), 66-85. https://doi.org/10.4018/jdwm.2006100104

[12] Rahman, N. (2016). SQL Scorecard for Improved Stability and Performance of Data Warehouses. *International Journal of Software Innovation* (*IJSI*), *4*(3), 22-37. https://doi.org/10.4018/IJSI.2016070102

[13] Zhou, W., Lu, J., Luan, Z., Wang, S., Xue, G., & Yao, S. (2013). SNB-index: A SkipNet and B tree based auxiliary Cloud index. *Cluster Computing*, *17*(2), 453-462. https://doi.org/10.1007/s10586-013-0246-y

[14] He, J., Wu, Y., Dong, Y., Zhang, Y., & Zhou, W. (2016). Dynamic multidimensional index for large-scale cloud data. *Journal of Cloud Computing*, *5*(1). https://doi.org/10.1186/s13677-016-0060-1

[15] Narasayya, V. & Syamala, M. (2010). Workload driven index defragmentation. *2010 IEEE 26th International Conference on Data Engineering* (*ICDE 2010*), 497-508. https://doi.org/10.1109/ICDE.2010.5447889

[16] Lee, K., König, A., Narasayya, C. V., Ding, B., Chaudhuri, S., Ellwein, B., Eksarevskiy, A., Kohli, M., Wyant, J., Prakash, P. et al. (2016). Operator and query progress estimation in Microsoft sql server live query statistics, *Proceedings of the 2016 International Conference on Management of Data*, 1753-1764. https://doi.org/10.1145/2882903.2903728

[17] Chaudhuri, S., Kaushik, R., & Ramamurthy, R. (2005). When Can We Trust Progress Estimators for SQL Queries? *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data. New York, NY, USA: ACM Press*, 575-586. https://doi.org/10.1145/1066157.1066223

[18] Mishra, C. & Koudas, N. (2007). A lightweight online framework for query progress indicators. *Proceedings of the 23rd ICDE Conference*. https://doi.org/10.1109/ICDE.2007.368996

[19] Dziedzic, A., Wang, J., Das, S., Ding, B., Narasayya, V., & Syamala, M. (2018). Columnstore and B+ tree - Are Hybrid Physical Designs Important? *Proceedings of the 2018 International Conference on Management of Data.* https://doi.org/10.1145/3183713.3190660

[20] Arulraj, J., Pavlo, A., & Menon, P. (2016). Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads. *Proceedings of the 2016 International Conference on Management of Data*. https://doi.org/10.1145/2882903.2915231

Contact information:

Blerta HAXHIJAHA EMINI, MSc in Software and Application Development
(Corresponding author)
South East European University,
Ilindenska 335, Tetovo, 1220, Republic of North Macedonia
E-mail: bh27486@seeu.edu.mk

Jaumin AJDARI, Assoc. Prof. Dr. in Computer Sciences
South East European University,
Ilindenska 335, Tetovo, 1220, Republic of North Macedonia,
E-mail: j.ajdari@seeu.edu.mk

Bujar RAUFI, Assoc. Prof. Dr. in Computer Sciences
South East European University,
Ilindenska 335, Tetovo, 1220, Republic of North Macedonia
E-mail: b.raufi@seeu.edu.mk

Besnik SELIMI, Assoc. Prof. Dr. in Computer Sciences
South East European University,
Ilindenska 335, Tetovo, 1220, Republic of North Macedonia
E-mail: b.selimi@seeu.edu.mk