UNIVERSITETI I EVROPËS JUGLINDORE
УНИВЕРЗИТЕТ НА ЈУГОИСТОЧНА ЕВРОПА
SOUTH EAST EUROPEAN UNIVERSITY

FAKULTETI I SHKENCAVE DHE TEKNOLOGJIVE BASHKËKOHORE
ФАКУЛТЕТ ЗА СОВРЕМЕНИ НАУКИ И ТЕХНОЛОГИИ
FACULTY OF CONTEMPORARY SCIENCES AND TECHNOLOGIES

POSTGRADUATE STUDIES – SECOND CYCLE

THESIS:

Techniques for improving query performance in Microsoft SQL Server

CANDIDATE:
Blerta Haxhijaha Emini

MENTOR:
Prof. Dr. sc. Jaumin Ajdari

Tetovo, August 2019

Contents

Declaration of original work

I hereby declare in my honor that I am the original author of this thesis. I have not copied from any other students' work and the thesis does not contain other people's work apart from reviewed references in accordance with the rules of referencing.

Blerta Haxhijaha Emini

Acknowledgements

I would like to express my sincere gratitude to my husband and family who have continuously motivated and supported me on this journey. Their encouragement has kept pushing me forward towards the completion of this work.

I would also like to express my gratitude to my mentor, whose suggestions, directions and continuous support has been invaluable during all the stages of preparation of this thesis.

List of Figures

List of Tables

Abstract

The demand for doing 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.

The principal objective of this master thesis is to research techniques and practices that improve the performance of common data operations written in T-SQL and executed in Microsoft SQL Server. Being that T-SQL is a declarative language that specifies what should be produced as result, instead of how to achieve that result, this master thesis will investigate the internals of SQL Server that affect the "how" of queries and data operations, in order to leverage this knowledge in proposing techniques that ensure performance gains. The internals of indexes are examined in order to shed light on their limitations, and to answer the question why indexes are sometimes used, and other times not, in the exact same query. The thesis then proposes techniques to overcome these limitations. Lastly, the power of wait statistics in identifying query execution issues is described experientially through different scenarios.

1. Introduction

What is acknowledged as fast enough by computer users is changing constantly. That which was considered as very fast during the advent of computers is far from what is deemed even acceptable now, a few decades later. Depending on the actual operation or process in question, generally, seconds are not acceptable anymore, and users are expecting that computer-related processes complete in matters of milliseconds or even faster.

Database operations are often responsible for a substantial portion of the delays associated with completing a computer-related action on some 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, so performance continues to be of critical importance [1].

The Microsoft SQL Server CSS and Development team have announced that after taking a deep dive into scalability and performance improvements, Microsoft SQL Server 2016 has been shipped with as much as 25% performance improvement [2]. This commitment and investment in performance improvement by one of the leading rDBMS vendors clearly shows that customer businesses and enterprises worldwide are looking for faster and faster database engines in order to fulfill their growing needs of responsive solutions that perform more rapidly than ever.

Due to the ever-growing need for performance enhancements in data operations in today's data-driven world, this master thesis will focus on the performance of T-SQL queries executed on Microsoft SQL Server.

However, this thesis will look at performance from another angle. It will attempt to provide answers on why in certain circumstances, indexes as the classical go – to solution for performance boosting, do not give the expected results. Often times DBAs and/or database developers struggle in understanding why their indexes, while tested successfully in testing environments, fail to be utilized on production. It will then aim to provide suggestions on how

to remove these limitations so that indexes serve their purpose in increasing the execution speed of queries that use them. Finally, it will look at the waits stored by SQL Server which track what has been waited for (in terms of contention for resources and being blocked) during query executions and use this information as the starting point in fixing the potential performance issues. Being that SQL Server is a complex system with memory, I/O, space and network consideration, to name just a few, it is difficult and time consuming to pinpoint exactly what the issue is when there is a performance downgrading observed. Wait statistics will be investigated in order to assess their potential in becoming the first point to look at while doing this troubleshooting.

The rest of this master thesis is organized as follows: *chapter Internals of the Microsoft SQL Server rDBMS* provides an outline of the key building blocks of SQL Server that are of interest to this thesis, followed by chapter *Related Work* which provides a summary of relevant research in the area of performance improvement. Chapter *Methodology* describes the methodology used in the *Experimental Set-up and Implementation* chapter, which in turn constitutes the bulk portion of this thesis. Finally, chapter *Conclusions* summarizes and concludes this work.

1.2. Internals of the Microsoft SQL Server rDBMS

1.2.1. Indexes and storage internals

The primary structure of a table in SQL Server can be either a heap or a B-tree, whereas indexes in SQL Server are B-trees [3]. If a table is organized as a clustered index (B-tree), then the column(s) that define the clustered index determine the order in which the table is physically stored [4]. On the other hand, the data rows of a table that does not have a clustered index are not organized in any particular order and these tables are referred to as heaps.

The name “clustered index” implies that the said structure is an index, but in Microsoft SQL Server, a clustered index also contains the table data in its leaf level. So, a clustered index in SQL Server is a B-tree where the leaf nodes are the actual data pages and the non-leaf nodes are index pages [5].

A table in Microsoft SQL Server can have at most one clustered index, because the clustering key(s) determine the physical order of the data records. A non-clustered index on the other hand does not affect the physical order of the data records, and hence their number is not limited to one per table. A non-clustered index is simply an additional database structure with leaf levels that point to the actual data records of the table.

A table can either be stored as an unordered structure, which is referred to as a heap, or the table can have its data ordered. The data can be ordered through the creation of a clustered index. Hence, if a table has a clustered index, it is no longer a heap and it is referred to as a clustered table or clustered structure.

Heaps are comprised of data pages only. There is no guarantee on any particular order of the data records in a heap, and there is no linkage between adjacent pages [6].

When a clustered index is created on a table, the table becomes ordered by the clustering key(s) and is no longer an unordered heap. The ordered data becomes the leaf level of the clustered index and then the index tree is built for navigation. The number of pages on the leaf level, as well as the number of index pages in the non-leaf portion of the clustered index depends on the amount of data records on the table and the record size.

Next in this section, an overview will be given on the storage internals of database files in Microsoft SQL Server. This will help clarify how the two different structures that exist for tables (heaps and clustered structures) are organized and stored in SQL Server, as well as how non-clustered indexes compare to the former. The internals of the storage engine and the organization of indexes will form the theoretical foundation upon which the thesis questions on indexes will be examined.

Microsoft SQL Server organizes data in memory chunks of 8KB, called pages. Pages are the fundamental unit of data storage in SQL Server. The data files that SQL Server uses are divided into pages of 8KB, and these pages are numbered from 0 up to the number of pages that fit into that data file. Hence, when specifying a page in SQL server, a 2-part specification is given, comprised of the file number and the page number within that file.

Pages in SQL Server begin with a 96-byte header, followed by data rows which are then placed serially on the page. One entry for each row found on the page is placed in the row offset

table, pointing to the first byte of that row. The row offset table is placed at the end of the page and its entries are in reverse sequence from the sequence of the page rows (figure 2.1).

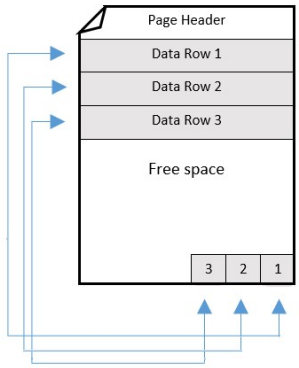

Figure 2.1. A page in Microsoft SQL Server

The amount of space that is available for storing data on a page, after subtracting the page header, row offset and some reserved space, is 8,060 bytes.

Next in this section, an overview of the row structure in Microsoft SQL Server will be given, in order to have a more complete understanding of how they fill the 8 KB pages discussed above. The row structure is given in Figure 2.2 below, for an IN_ROW_DATA allocation unit (described next in *Allocation Units*) without any sparse columns or compression enabled.

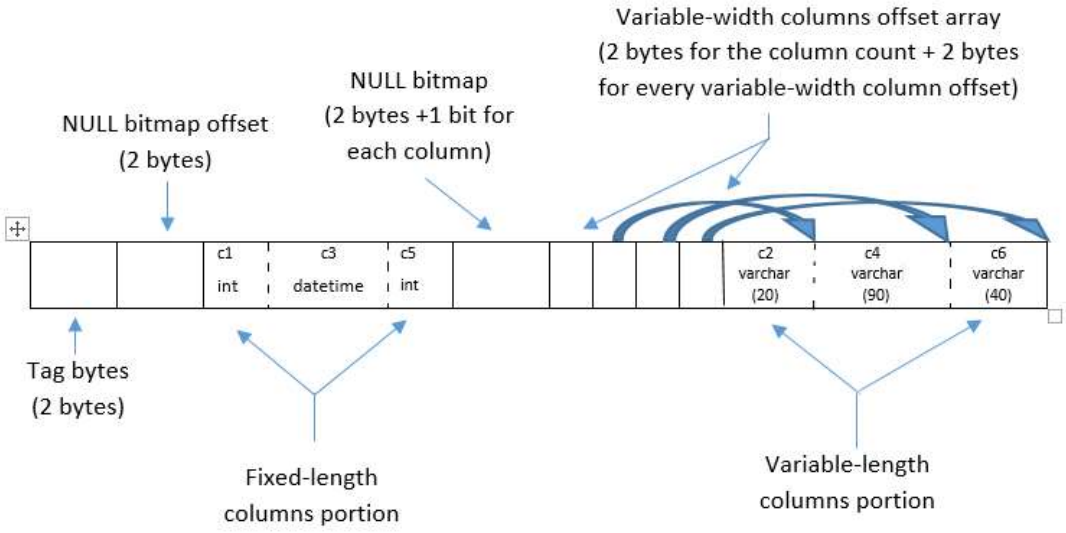

Figure 2.2. Record structure in Microsoft SQL Server

The record header is 4 bytes long. It consists of the tag bytes which contain information about the record type (2 bytes), and the null bitmap offset (also 2 bytes). This offset points to the null bitmap found further in the record and is explained later in this section. Next comes the fixed-length columns portion, which contains the data of the columns which are fixed-length. The NULL bitmap follows next, and it helps manage the nullability of the row columns. It contains 2 bytes for the row column count, and one bit for every column to denote if it is NULL or not (so the NULL bitmap is at least 3 bytes in size). Finally, the variable-length column offset array contains 2 bytes for the count of columns in the row which are of variable length, and 2 bytes per variable-length column, giving the offset to the end of the column value [7].

Another important unit of data storage in SQL Server is the extent. An extent is a collection of 8 consecutive pages. There are 2 types of extents: uniform extents, where all 8 pages of the extent are used by the same object, and mixed extents, where each of the pages of the extent can be used by a different object.

1.2.2. Allocation units

In SQL Server, there are three types of allocation units available, depending on what type of data is being stored into pages. Allocation units are collection of pages within a heap or B-tree used to store data depending on the data type and characteristics. The three different allocation units in SQL Server are the following [8]:

- IN_ROW_DATA
- ROW_OVERFLOW_DATA
- LOB_DATA

The first allocation unit IN_ROW_DATA contains pages that are used to store data or index rows that contain all data types, except for large object (LOB) data types. This allocation unit is for data rows that fit into the 8,060 bytes limit of the page. There is one IN_ROW_DATA allocation unit for every partition used by a table, index, or indexed view.

The ROW_OVERFLOW_DATA allocation unit is a collection of pages that is used to store variable length data stored in nvarchar, varchar, sql_variant or varbinary columns that exceed the 8,060 byte row size limit. When this limit is reached, the column with the largest width

from that row is moved to a page in the ROW_OVERFLOW_DATA allocation unit, and the original page keeps a 24-byte pointer to this new location.

The third allocation unit LOB_DATA is a collection of pages that store data in text, ntext, xml, image, varbinary(max), varchar(max), nvarchar(max) or CLR user-defined types. One LOB_DATA allocation unit per partition is allocated when a table or index has one or more LOB data types to store. The LOB_DATA and ROW_OVERFLOW allocation units are what give the possibility to define very large rows in SQL Server.

For SQL Server to determine where to store any data that is being inserted into tables, it makes use of special data structures that track allocation and deallocation of pages, as well as the free space available in them [9]. These special structures are in essence special page types that SQL Server keeps in order to manage the available space in database file(s). The types of storage tracking data structures that are of interest to our topic are the GAM (Global Allocation Map), SGAM (Shared Global Allocation Map), Page Free Space (PFS) and IAM (Index Allocation Map). A short overview of them is provided below.

A GAM page, just like any other type of page in SQL Server is 8KB in size. 8,000 bytes of the 8KB GAM are used to store information about availability of extents, with one bit representing one extent. The bit has value 0 if the extent is being used (is allocated), or 1 if the extent is free. 8,000 bytes with 8 bits each equals 64,000 bits, so that's the number of extents that are covered by a single GAM page. The third page in the first data file is a GAM page, as shown in figure 2.3 below.

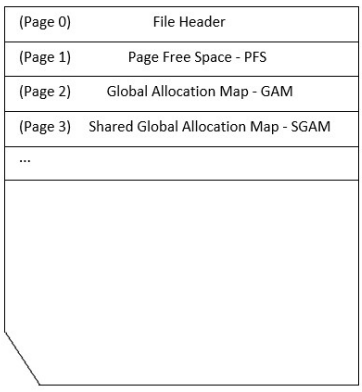

Figure 2.3. Pages in a Microsoft SQL Server file

Since one GAM page tracks the availability of 64,000 extents (which is 512.000 pages), the first GAM page covers almost 4GB of storage space, and this storage space is known as one GAM interval. All the calculations related to GAM pages are detailed in table 2.1. Since most databases today are larger than 4GB or one GAM interval, multiple GAM pages exist in order to track the availability of all the data pages in the database.

Unit	Size
Page size	8 KB
Extent size	64 KB (8 pages * 8 KB)
GAM portion that tracks extent allocation	8,000 B
Number of bits in a GAM that track extent allocation	64,000 bits (8,000 B * 8 bits)
Number of extents covered by a single GAM page	64,000 extents (one bit per extent)
Number of pages covered by a single GAM page	512,000 pages (64,000 extents * 8 pages)
Storage space covered by a single GAM page	~ 4 GB (512,000 pages * 8 KB = 4,096,000 KB)

Table 2.1. GAM related calculations

When a GAM page marks an extent with bit 0 as used, that extent can be in two different states – it can be partially used or completely used. In order to track this additional information, the SGAM page is used. Like the GAM, the SGAM page also uses one bit per extent and covers 64,000 extents or 512,000 pages. The GAM and SGAM work in parallel, and the following combinations are possible (table 2.2):

Current use of extent	GAM bit setting	SGAM bit setting
Free, not being used	1	0
Uniform extent, or full mixed extent	0	0
Mixed extent with free pages	0	1

Table 2.2. Bit setting combinations for pages in GAM and SGAM

An IAM page is also a type of bitmap, which, like the GAM and SGAM, covers extents that span approximately 4GB of space. The IAM page tracks which extents within that specific interval belong to a single allocation unit of a table or index. So IAM pages belong to allocation units and as such, they are created every time a new allocation unit is created, or when the allocation unit grows to span more than one GAM interval. When multiple IAM pages exist for a single allocation unit, they are linked into what is called an IAM chain.

Finally, a PFS page looks different from the GAM, SGAM and IAM pages which were bitmaps with one bit in the page per extent. The PFS page is broken down into bytes, and each byte represents a page. Each byte tracks if the page is allocated or not, and if it is allocated, to what

percentage it is full: 1 to 50 percent full, 51 to 80 percent full, 81 to 95 percent full, or 96 to 100 percent full. SQL Server uses the PFS page to check which pages have been allocated and if there is enough space to enter a new row on some allocated page.

1.2.3. Wait statistics

Wait statistics were introduced in SQL Server 2008, and they have already grown to include around a thousand different wait types. These wait types tell us what is delaying queries, and this is contention for various resources. It could be that a query is waiting for a lock, it could be waiting for a page from disk, it could be waiting for some memory to be available so that it can start running. SQL Server keeps track of all these waits happening due to contention for various resources, and this data provides a reliable foundation in order to start performance troubleshooting.

Microsoft SQL Server comes with its own mini operating system, called the SQL Server Operating System or SQL OS. Among other things, the SQL OS performs memory management and scheduling as its two main functions [10]. It gets memory from Windows and then parcels it out to threads within SQL Server for its own needs.

A thread is the smallest unit of execution within a process. There are many threads that can be running within a single process. Threading works is such a way that each thread is given a small amount of processing time to run until it needs to wait for a resource or until it's run for a certain amount of time. Then it relinquishes the processor so that other threads can continue. Eventually, it will get back on the processor and do more work. This process of running for a while, having to wait and then running again is called scheduling. Scheduling gives the user the impression that lots of things are happening at the same time, whereas, in fact, on each individual processor, only one thread is executing at once. But because each executes for a very small amount of time and they relinquish the processor to another thread, it seems like there are lots of things happening concurrently.

SQL Server uses operating system threads, as worker threads, to perform the tasks necessary to complete a given process. There are threads that are dedicated for a particular task (a

dedicated thread for the CHECKPOINT process, for example) but SQL Server maintains the others in a thread pool and uses them as necessary to process user requests [11].

Many performance gains in SQL Server come from the ability to execute in parallel. SQL Server can introduce parallelism by using several processors during the execution of an expensive query so that it runs faster [12]. For example, in a machine with 4 processors, all of which are configured to be allowed to be used by SQL Server, the query execution portion of the engine may decide to use a degree of parallelism 4. In that case, the threads involved would be the following (figure 2.4):

- a single control thread,
- four threads that produce streams which go into some type of exchange operator to decide where the record should go on the output side,
- four threads consuming those streams on the output side of the operation

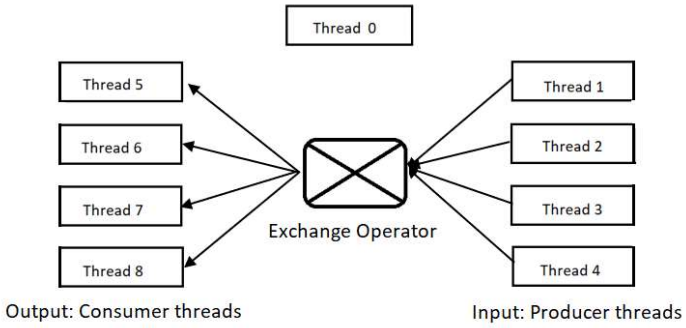

Figure 2.4. Threads in a parallel execution

A concrete example for this operation would be a large table scan of four million rows. If there are four CPUs available, 1 thread can be running on each of the CPUs, scanning 1/4th of those four million rows. This process would run in less amount of time than 1 thread having to scan all four million rows. When the query optimizer decides that an operation can run in parallel, it will produce a query plan that can be parallelized at the time the query executes. If it decides that there are no operations that could be parallelized, it will produce a plan that can only be run single threaded, or serially. Then, the query execution portion of the query processor decides what level of parallelism to actually use, based on the resources there are available at that time and various configuration options on server or query level.

SQL server performs its own thread scheduling in non-preemptive mode. Windows (which does pre-emptive scheduling) does not control when a given thread has to get off the processor so that the other threads can execute.

Normally, there is one SQL OS scheduler per CPU “core”, regardless if it's a logical core or a physical core. A scheduler can be said to compose of three parts (figure 2.5): the processor itself where only one thread can be running at a time; the list of threads that are waiting for resources (the waiter list), and the queue of threads that have all the resources they need and they're waiting for their turn to get back on the processor (the runnable queue). Scheduling works by having threads switch from the processor to the waiter list, then to the runnable queue and back to the processor, up until they finish doing all their work.

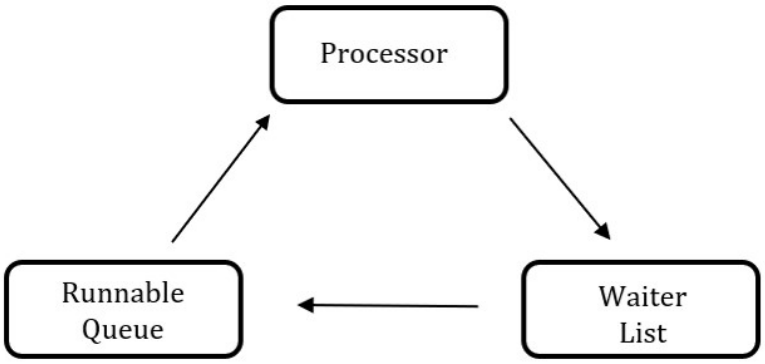

Figure 2.5. SQL OS scheduler components

While on the scheduler, the thread can be in one of three states:

- **Running** - when the thread is executing on the processor. Only 1 thread per scheduler can have the state running.
- **Suspended** - when the thread must wait for a resource to become available, it can't continue running and its state changes to suspended. The thread moves to the waiter list and simply waits for its resource to become available (ex. a page to be fetched to memory).
- **Runnable** - when the thread is signalled that its resource is available, its state changes to runnable and it moves to the runnable queue (which is a FIFO queue) to wait for its turn. When it makes its way up to the top of the runnable queue, it goes back to state running again.

The threads that are being used for query execution transition between these states until their

work is done. There is a limit to the processing time a thread is allowed to use the processor in state running, and that is 4 milliseconds. A thread must yield the processor after the exhaustion of this 4-millisecond quantum, even if it does not need to wait for any kind of resource to become available. In that case, the thread bypasses the waiter list and goes directly to the bottom of the runnable queue. If, however, the running thread needs a resource before its allotted quantum is complete, it moves to the waiter list. Once the resource is available, the thread moves to the runnable queue and then, when its time comes, back onto the CPU [11].

The analysis of threads waiting for a resource to become available, and the analysis of how long they remain suspended and how long they wait on the runnable queue, is the basis of the wait statistics analysis. This analysis is also the basis of the waits and queues performance tuning methodology.

A **wait** in SQL Server is what occurs when a thread which is running on the processor cannot proceed because it needs a resource and this resource is not available. SQL Server keeps track of all the different resources being waited for - how often they have been waited, as well as how long they have been waited for by the various threads. Each of the resources maps to a wait type and these wait types can be retrieved from various DMVs. An example of a resource that a thread might have to wait for because it isn't available is a lock. This shows up as a LCK_M_S or LCK_M_S wait type, depending if a shared or exclusive lock is needed for the resource, respectively. Another common wait type is the CXPACKET, which is accumulated by a thread which is involved in a parallel operation and is waiting for another thread in that operation to finish. An overview of the common wait types defined in SQL Server, as well as information on how to leverage them in order to increase performance can be found in [13] and [14].

The total time spent waiting by a thread (known as *wait time*) is the sum of the resource wait time and signal wait time (figure 2.6), where:

- Resource wait time - Time spent waiting for the resource to be available, i.e. time spent on the Waiter List with state SUSPENDED
- Signal wait time - Time spent waiting to get back to the processor after the resource is available, i.e. time spent on the Runnable Queue with state RUNNABLE

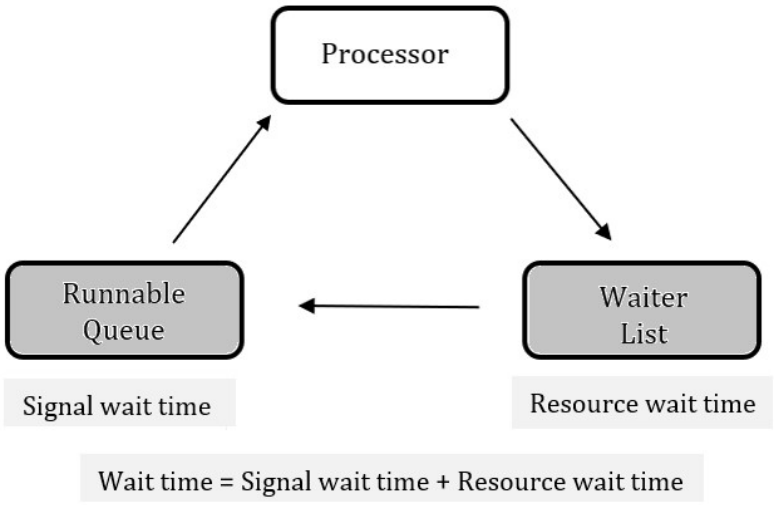

Figure 2.6. Signal, resource and total wait times

In the next section, a short description of the wait types relevant to the experiments performed in section *Troubleshooting with wait statistics* of this thesis will be given below.

In the first experiment, latch-related wait types are encountered. Latches are short-term lightweight synchronization primitives used to protect memory structures for concurrent access [9]. Unlike locks which protect transactional consistency and isolation and are held during the whole duration of the transaction, latches are shorter term and are used to guarantee the consistency of in-memory objects. However, they do have similar modes like locks, including shared and exclusive modes [13]. Latches are taken any time there needs to be a page modification – either moving the page from disk to memory or vice versa, writing a record onto a page or changing the page's metadata.

There are several wait types that map to latch-related waits, as described below ('XX' stands for the mode abbreviation):

- PAGEIOLATCH_XX waits - latches waiting for data pages to be read from disk into memory
- PAGELATCH_XX waits - latches for access to in-memory data pages
- LATCH_XX waits - Latches for access to other data structures

In the second experiment, CXPACKET waits are encountered. CXPACKET waits occur during parallel query executions. Whenever there is parallelism occurring, there will always be

associated CXPACKET waits recorded [13], since the controller thread (thread 0) will always produce CXPACKET waits while waiting for the threads to finish. Related to the producer threads, according to [15], there are two main scenarios when CXPACKET waits are produced by them. The first scenario is when a thread from the parallel query is blocked and cannot continue while waiting for a resource. The second scenario is when one of the threads from the parallel query takes longer to execute that the rest of the threads and the rest of the threads have to wait for the slower thread to complete. During the waits in both scenarios, CXPACKET waits are produced.

2. Literature Review

There is an abundance of available literature 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 maximise execution speed. In [16], 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 rare queries and reports, and are then dropped. They 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_executesql system stored procedure for running ad-hoc queries.

In [17], 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 [18], 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 the IN or OR conditions. The reason behind this is that SQL Server 2008, in case of using the IN condition, will access the index for a number of 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 it into a pair of >= <= conditions.

In [19], several principles for proper usage of indexes are outlined, suggesting that table scans should be avoided since seeks have better efficiency in most cases. The paper concludes that

functions and calculations should be avoided or replaced, or be used as little as possible in queries, in order to make indexes effective.

Other papers have focused directly on indexes, recognizing them as crucial in performance improvement strategies. Ferrar et al. in [20] have proposed a methodology for automated determination and selection of optimal indexes. Their method consists of capturing a workload representative of queries executed during system use, computing cost benefits for different combinations of indexes, and recommending the best indexes to be created by selecting those that have the most favourable cost on the captured workload. A similar index selection mechanism allowing for efficient generation of index recommendations for a given workload is also proposed by Brown et al. in [21].

Monteiro et al. in [22] have used heuristics to enable indexes creation and destruction for DBMSs. Their engine is based on an integration between software agents and the components of the database management system. The proposed non-intrusive architecture is claimed to allow the complete automation of the index choice, creation and destruction, during normal operation of the database management system.

This research incentive to explore methods that automate the physical design of relational databases, based on a workload of SQL statements, gained lots of prominence in scope of the AutoAdmin project. The principles however were not limited to the Microsoft SQL Server RDBMs alone but were applicable to other relational database management systems as well.

In [23], Agrawal et al. go beyond the underlying assumption that in order to provide automation of the database physical design based on a workload of SQL statements, that workload must be a set of SQL statements. The authors instead look at the possibility of treating the SQL statements workload as a sequence in order to exploit the ordering of statements, and present scenarios where the sequence information is crucial for performance improvement.

Chaudhuri and Bruno in [24] discuss the limitation present in most of the work on automated database physical design, which is that the tuning tools are invoked offline and depend on DBAs to select representative workloads. They propose an always-on, low-overhead technique that continuously modifies the current physical design, by reacting to changes in

the query workload.

In [25], Narasayya and Chen argue that traditional RDBMs do not provide an adequate solution to the common scenario in data warehousing that operate on large amounts of data, whereby many GROUP BY queries are executed in order to analyse and understand the data. Claiming that numerous GROUP BY queries are expensive, the paper proposes an optimization technique for GROUPING SETS queries for common data analysis scenarios.

An ambitious undertake 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 [26] makes a summary of the progress from a decade of research in self-tuning database systems, while [27] reviews the lessons learned from the AutoAdmin project at Microsoft Research up to year 2011. Publications from the AutoAdmin project pertinent to performance enhancements are also referenced and described in this section.

Bruno and Chaudhuri in [28] argue that although there has been a considerable amount of recent research on automated selection of physical design in database systems, the proposed techniques have become increasingly complex. In their paper, the authors critically examine the architecture of current solutions, and then move on to design a new framework that reduces the heuristics and assumptions used in previous approaches.

The same authors in [29], argue that although current techniques for automating the physical design in database systems give good recommendations, they are quite resource intensive, making DBAs often reluctant in deciding to start a tuning session. In their paper, they introduce an alerter that helps determine when a physical design tool should be invoked, claiming also that their mechanism is lightweight and is able to handle large workloads with little overhead.

In [30], the authors discuss the limitations of query hints that try to address situations when optimizers choose a poor plan for a given query. Claiming that they are not flexible enough to handle a multitude of non-trivial scenarios, the authors introduce a hinting framework that offers rich constraints that influence the optimizer to pick more optimal plans for execution.

Lee et al. in [31] 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 [32] and [33].

In [34], Dziedzic et al. focus on the importance of hybrid database physical designs, which consist both of B+ tree indexes and columnstore 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 columnstore and B+ indexes in a given workload.

In [35], 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.

Narasayya and Syamala in [36] 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 also study the question of which indexes is it most appropriate to defragment for a given workload.

3. Methodology

This thesis will follow an experimental approach in investigating 3 research questions related to query performance in SQL Server.

First, the thesis will investigate at which point non-clustered indexes stop being used in queries, even-though previously the same queries, written in the same way, did make use of those indexes. This is a source of confusion for many T-SQL developers and database administrators, who suddenly find out that their queries no longer use the intended indexes, even-though previous tests demonstrated that the indexes were used.

In order to provide an answer to the question of why for the same query, sometimes non-clustered indexes are used, and at other times not, this thesis will create a test table with several indexes on it, and look at the query execution plan and IO statistics for several queries, in order to identify and provide an explanation for this turning point on index usage.

Next, the thesis will investigate how these limitations can be removed, and queries fine-tuned in order to utilize the intended indexes, by proposing a different technique in choosing what kind of indexes to create. Again, query execution plans will be used in order to confirm if the goal of making the indexes be utilized in the queries is achieved.

Finally, this thesis will show how wait statistics can point to potential performance issues, particularly involving indexes, by looking at what resources SQL Server has been waiting on while executing queries.

All the experiments defined in this thesis are performed on a machine with Microsoft SQL Server 2014 Developer edition with Service Pack 2 installed. This machine has 4 cores (8 logical processors), and 8GB of RAM installed.

4. Experimental set-up and Implementation

4.1. Turning point on index usage

Let's consider a table [Clients], which contains 8 columns that add up to 393 bytes in size (table 5.1). However, as described in section *Indexes and storage internals*, SQL Server also stores some overhead in the data records. For our table structure, the extra overhead is as follows:

- 2 Bytes for the record header
- 2 Bytes for the NULL bitmap pointer
- 3 Bytes for the NULL bitmap (2 Bytes column count, and one bit for each column)

Since the table contains no variable-length columns, the record structure does not include the variable-length specific portion of the records. This makes the total length of the data rows in our table 400 bytes.

Column Name	Column Type	Column size (B)
ClientID	INT	4
TaxCode	CHAR(11)	11
Name	NCHAR(60)	120
Surname	NCAHR(60)	120
IsActive	TINYINT	1
RegistrationDate	DATETIME	16
Address	NCAHR(57)	114
Phone	CHAR(15)	15

Table 5.1. Columns of [Clients] table

Table [Clients] 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. This means that our table is a clustered index, not a heap.

In order to calculate the number of rows in each leaf level page of the clustered index, we divide the available page size for storing records (8,060 B) by the total length of the data row (393B data columns length plus 7B row overhead). Fixed-length column records are part of the IN_ROW_DATA allocation unit and as explained in section *Allocation Units*, they cannot span

pages:

$$\frac{8{,}060\ bytes/page}{400\ bytes/row} = 20 \text{ rows/page (+2 bytes row offset)}$$

Since each data page can hold a maximum of 20 rows, and our table contains a total of 80,000 rows, we can calculate the number of data pages in the leaf level of the index:

$$\frac{80{,}000\ rows}{20\ rows/page} = 4{,}000 \text{ pages}$$

We can check these calculations from the dm_db_index_physical_stats dynamic view, using the *Detailed* mode (figure 5.1):

```
SELECT [index_depth] AS [Depth],
       [index_level] AS [Level],
       [record_count] AS [Rows],
       [page_count] AS [Pages],
       [min_record_size_in_bytes] AS [RowMinLen],
       [max_record_size_in_bytes] AS [RowMaxLen]
FROM   [sys].[dm_db_index_physical_stats]
       (DB_ID (N'ThesisDB'),
       OBJECT_ID (N'ThesisDB.dbo.Clients'),
       1,
       NULL,
       'DETAILED');
       GO
```

Figure 5.1. Investigating the clustered index on [Clients table]

The results of this query are given in table 5.2, and it can be observed that indeed there are 4,000 pages in the leaf level of the index.

Depth	Level	Rows	Pages	RowMinLen	RowMaxLen
3	0	80000	4000	400	400
3	1	4000	14	11	11
3	2	14	1	11	11

Table 5.2. Clustered index on [Clients] table

Table 5.2 further shows that the clustered index is of depth 3. It contains a leaf level (Level 0) with the 80,000 data rows spread over 4,000 data pages. As expected, the row length is 400 bytes because 7 bytes of overhead were added to the 393 bytes of data. The minimum and

maximum row length in the leaf level are both 400 Bytes since our table does not contain variable-length columns.

The next level in the index is the intermediate level (Level 1), which contains 4000 index rows , spread over 14 pages. There are 4000 index records at this level because at the leaf level there were 4000 pages, and according to the index structure in Microsoft SQL Server, there is one index record per page in the level below, and that index record points to the first record of the page in the level below. This explains also why there are 14 index records in the root level (Level 2), pointing to the 14 pages of the intermediate level.

To list the pages allocated to our clustered index, the DBCC IND command can be used, which returns a row for every page allocated to the requested object [15]. The resulting dataset of this command will be inserted in a helper table [ClusteredIndexPages], which can be then manipulated with an ORDER BY clause to better see the ordering of the pages of the index.

```
insert into [ClusteredIndexPages]
exec ('DBCC IND (ThesisDB, Clients, 1)')
```

Figure 5.2. DBCC IND for the clustered index on [Clients] table

Table [ClusteredIndexPages] contains 4,016 pages, which is 1 page more than the total number of pages of the clustered index (table 5.2). The extra page is for the IAM page of the index (IAM pages were discussed in the Introduction section). A portion of the [ClusteredIndexPages] table (10 out of the 4,016 result pages) is given in table 5.3 below, with information for the following pages:

- the IAM page (1:146)
- first three pages of the leaf level (1:256), (1:280), (1:281)
- last three pages of the leaf level (1:4380),(1:4381),(1:4382)
- first page of the intermediate level (1:264)
- last page of the intermediate level (1:1149)
- the root page (1:1136)

IndexLevel	PageFID	PagePID	PrevPageFID	PrevPagePID	NextPageFID	NextPagePID
NULL	1	146	0	0	0	0
0	1	256	0	0	1	280
0	1	280	1	256	1	281
0	1	281	1	280	1	282
…	…	…	…	…	…	…
0	1	4380	1	4379	1	4381
0	1	4381	1	4380	1	4382
0	1	4382	1	4381	0	0
1	1	264	0	0	1	1137
…	…	…	…	…	…	…
1	1	1149	1	1148	0	0
2	1	1136	0	0	0	0

Table 5.3. Pages of the clustered index on table [Clients]

A visual representation of the clustered index, with the file and page numbers as returned from the DBCC IND command, is given in the figure 5.3. below.

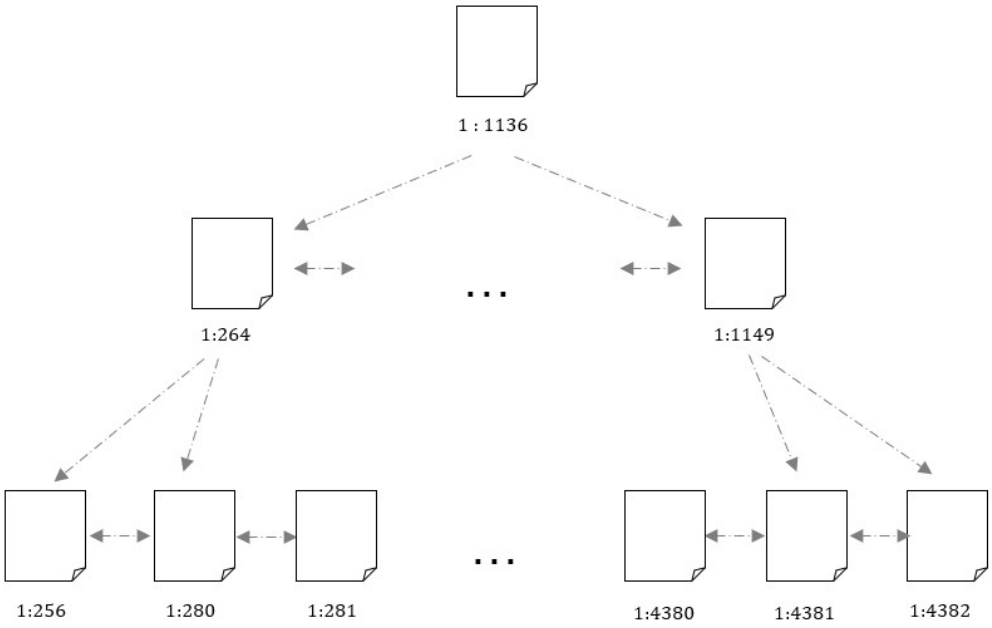

Figure 5.3. Visual Representation of the clustered index on table [Clients]

Next, a non-clustered index on column [TaxCode] is added for the [Clients] table (figure 5.4). A non-clustered index in Microsoft SQL Server is a separate structure (unlike the clustered index which contained the table itself in its leaf level), so the leaf level pages of the non-clustered

indexes point to the base data table's row lookup ID. If the base data table is a clustered structure, then this row lookup ID is the clustered key, whereas if the data table is a heap, then the RID of the heap rows is used.

```
create unique nonclustered index TaxCodeNC
on [Clients] (TaxCode)
```

Figure 5.4. Creation of the [TaxCodeNC] non-clustered index on table [Clients]

To examine the non-clustered index, similarly as with the clustered index, the dm_db_index_physical_stats dynamic view will be used. For the non-clustered index, it yields the following results given in table 5.4 below:

Depth	Level	Rows	Pages	RowMinLen	RowMaxLen
2	0	80000	212	19	19
2	1	212	1	21	21

Table 5.4. Non-clustered index on [Clients] table

The non-clustered index is of depth 2, meaning it contains 2 levels - the leaf level and the root level. The leaf level contains the 80,000 records spread over 212 pages. The number of pages is smaller when compared to the clustered index because the records themselves are smaller (19 B in the leaf level). This is so because these records do not contain all the columns of the actual data record, but only the non-clustered key ([TaxCode] in our case) and the base table row lookup id (the [ClientID] in our case) and some overhead.

Again, the output of the DBCC IND command for the non-clustered index gives the pages allocated for this object. A portion of this table containing some of the leaf pages, as well as the root and IAM page, is given in table 5.5.

Next, we will examine several queries where the clustered and non-clustered indexes are used effectively, and then move on to scenarios where these indexes are not deemed as effective by SQL Server anymore.

IndexLevel	PageFID	PagePID	PrevPageFID	PrevPagePID	NextPageFID	NextPagePID
NULL	1	150	0	0	0	0
0	1	4628	1	4627	1	4629
0	1	4629	1	4628	1	4630
0	1	4630	1	4629	1	4631
…	…	…	…	…	…	…
0	1	4851	1	4850	1	4852
0	1	4852	1	4851	1	4853
0	1	4853	1	4852	1	4854
1	1	5168	0	0	0	0

Table 5.5. DBCC IND command for the non-clustered index [TaxCodeNC]

The structure of the non-clustered index is given in figure 5.5 below:

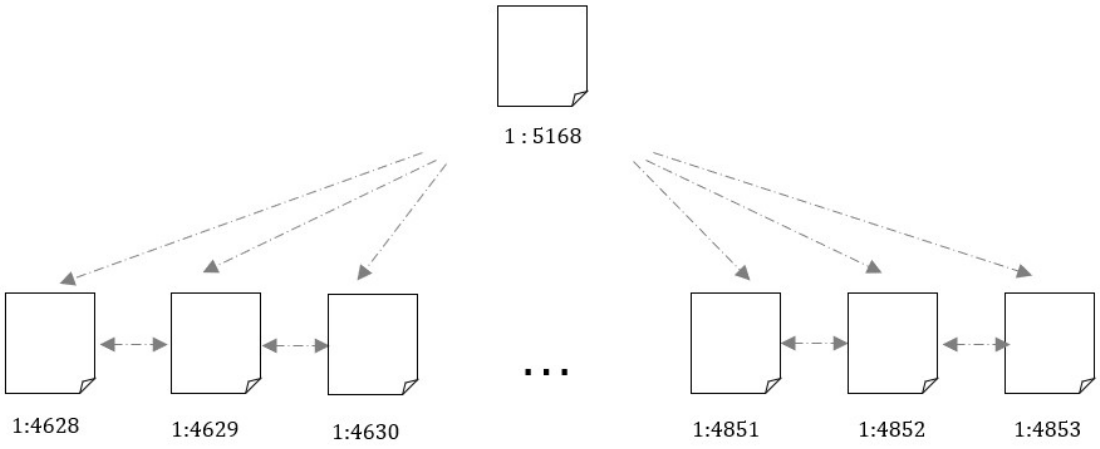

Figure 5.5. Visual representation of the non-clustered index

Consider the following query (figure 5.6) which makes for a rather straightforward example where the clustered index would be used:

```
SET STATISTICS IO ON;
GO

SELECT [c].*
FROM [dbo].[Clients] AS [c]
WHERE [c].[ClientID] = 2438;
GO
```

Figure 5.6. A simple equality query

This query references column [ClientID] in its WHERE clause in an equality comparison. [ClientID] is the clustering key of the clustered index [ClientsPK] in table [Clients], and we would

expect that the SQL Server engine would make use of this clustered index when evaluating the result. By examining the actual execution plan, it is shown that this is exactly the case (figure 5.7).

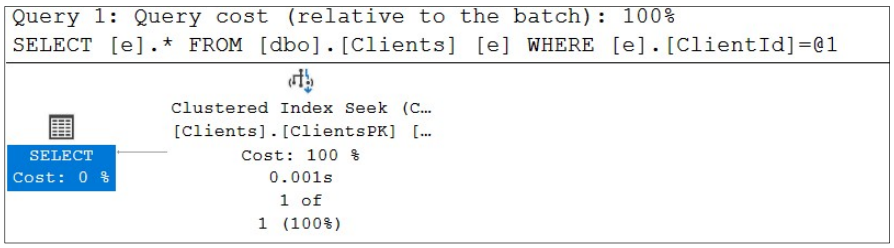

Figure 5.7. Execution plan of the equality query

Regarding the IO statistics, the following message is displayed:

```
Table 'Clients'. Scan count 0, logical reads 3, physical reads 2, read-ahead reads 0,
lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
```

As expected, there are 3 logical reads, because 3 different pages of the [ClientsPK] clustered index structure have to be read: first the root of the index, then the corresponding intermediate page, and finally the data page containing the client record with [ClientId] = 2348.

The next query to be examined will reference the [TaxCode] column of the [Clients] table in the WHERE clause, hinting this to be a case where the non-clustered index defined on [TaxCode] could be used.

```
SET STATISTICS IO ON;
GO

SELECT [c].*
FROM [dbo].[Clients] AS [c]
WHERE [c].[TaxCode] = '87704182846';
       GO
```

Figure 5.8. A query searching by [TaxCode]

It should be noted that the query does 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 [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. The non-clustered index [TaxCodeNC]

defined on column [Clients].[TaxCode] is used in an *index seek*, but there is also a *key lookup* operation performed in the clustered index [ClientsPK], to retrieve the rest of the columns.

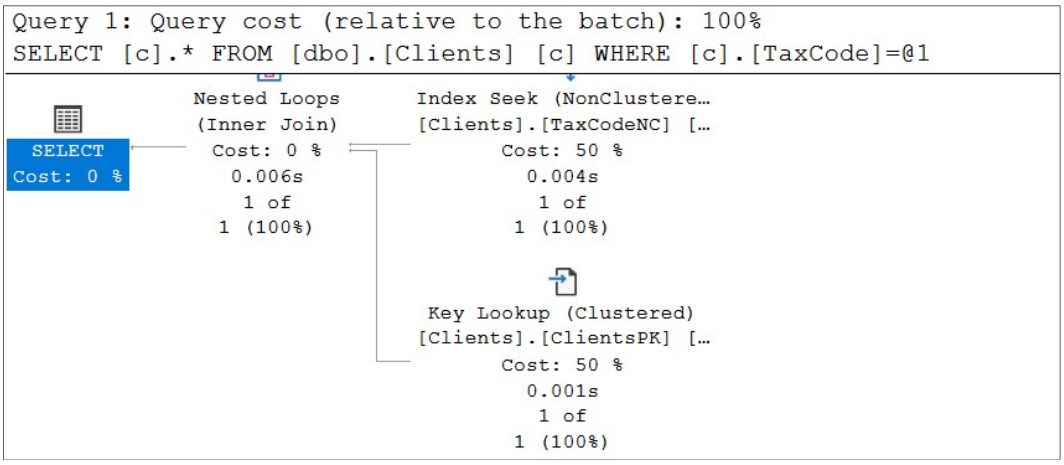

Figure 5.9. Execution plan of the query searching by [TaxCode]

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

```
Table 'Clients'. Scan count 0, logical reads 5, physical reads 4, read-ahead
reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
```

There are 5 logical reads performed, and this matches our expectation: in order to evaluate the query, the non-clustered index seek operation needs to take place, by accessing the root page and the corresponding leaf page of the non-clustered index, thus making for 2 logical reads. From the leaf page, the clustered key is read and this enables the key lookup operation in the clustered index, where again the root page of this index, the corresponding intermediate page, and finally the leaf page containing the record are retrieved. This contributes to additional 3 logical reads, for a total of 5, as displayed by the IO statistics message.

This sequence of operations is shown figuratively in fig. 5.10 below:

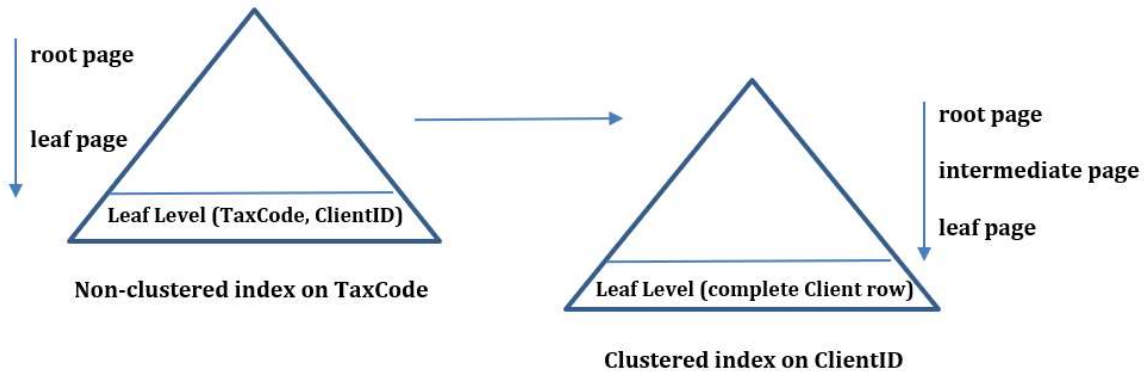

Figure 5.10. An index seek followed by a key lookup

The number of logical reads of course increases when there are more records returned by the query. Let us consider a result set of 20 client records; in order to retrieve all the columns for these 20 records, according to Figure 5.10 above, there should be:

- 2 or 3 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 2 adjacent leaf pages, in which case there would be a total of 3 logical reads instead of 2)
- 20 x 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) in order to retrieve the rest of the columns of that client record.

In order to check this assumption, the following query (figure 5.11) which returns 20 rows from the Clients table will be examined through its execution plan and IO statistics:

```
SET STATISTICS IO ON;
GO

SELECT [c].*
FROM [dbo].[Clients] AS [c]
WHERE [c].[TaxCode] BETWEEN '01153701453' AND '01164967899'
GO
```

Figure 5.11. A range query

The query returns the following information about the IO statistics:

```
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.
```

The execution plan of the query is as follows (figure 5.12., again a non-clustered index seek

followed by key lookup in the clustered index):

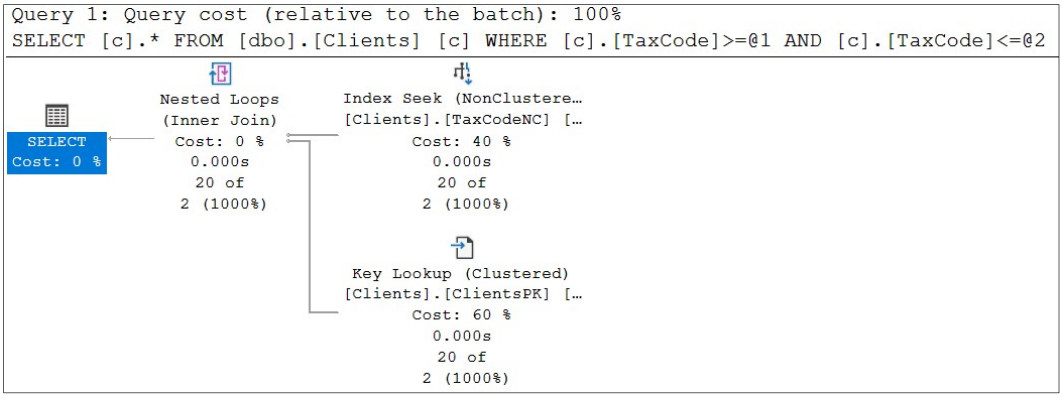

Figure 5.12. Execution plan of the range query

This number of logical reads is still more efficient then doing 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 two sample queries so far.

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's consider a more generalized form of our range query, with unknown min and max range boundaries (let's call them *x* and *y*):

```
SELECT [c].*
FROM [dbo].[Clients] AS [c]
WHERE [c].[TaxCode] BETWEEN x AND y
```

Figure 5.13. Generalized range query

The number of page 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. 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 then decide to switch to using a

clustered index scan directly, instead of going through the two separate operations (non-clustered index seek and key lookup in the clustered index) and joining 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's next examine the execution plan and IO statistics of the following query, which returns 1,000 rows:

```
SET STATISTICS IO ON;
GO

SELECT [c].*
FROM [dbo].[Clients] AS [c]
WHERE [c].[TaxCode] BETWEEN '60904989375' AND '62115722473'
GO
```

Figure 5.14. Range query with 1.000 resulting rows

The execution plan (figure 5.15) and the IO statistics message are given below:

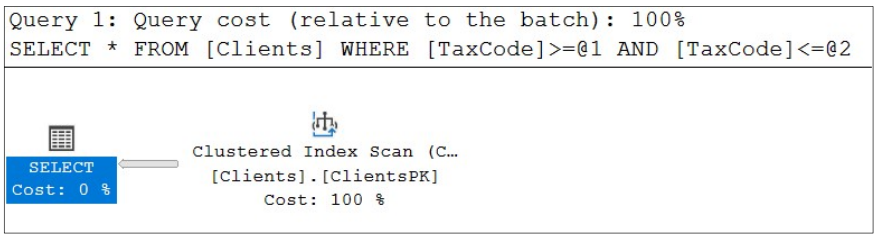

Figure 5.15. Execution plan of the range query with 1.000 resulting rows

```
(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 do a clustered index scan. Although the structure of this query is identical compared to the previous test queries, this time the execution plan is different. Even though 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 test case, table 5.6 shows information on the type of operation(s) used during query execution for several queries with different number of resulting records.

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 also 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 aren't as useful as they were expected to be, and the habitual recommendation of simply 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 on 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's in the where clause of queries that are most critical to performance and for which more consistent performance and consistent plans are needed.

Table 5.6. Operation used according to number of records in the result set

Query	No. of records in the result set	Operation used
SELECT [c].* FROM [dbo].[Clients] WHERE [c].[TaxCode] BETWEEN x AND y	1 record	Non-clustered Index Seek and Clustered Index Key Lookup
	20 records	Non-clustered Index Seek and Clustered Index Key Lookup
	350 records	Non-clustered Index Seek and Clustered Index Key Lookup
	800 records	Non-clustered Index Seek and Clustered Index Key Lookup
	1,000 records	Clustered Index Scan
	1,500 records	Clustered Index Scan
	10,000 records	Clustered Index Scan
	80,000 records	Clustered Index Scan

4.2. Index fusion

Following our thesis objective of proposing techniques that increase query performance in Microsoft SQL Server, next we will examine ways on how to remove the limitations on index usage presented in the previous section.

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 the non-clustered index structure examined in the previous section should be made. The non-clustered index on [Clients].[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 key was then used in key lookup operations into the clustered index when it was necessary to retrieve the rest of the columns of the client record requested in the SELECT statement.

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 read the complete table. Consider the following query (figure 6.1):

```
SET STATISTICS IO ON;
GO

SELECT [c].[ClientId], [c].[TaxCode]
FROM [dbo].[Clients] AS [c]
WHERE [c].[TaxCode] BETWEEN '60904989375' AND '99733993541'
GO
```

Figure 6.1. Range query on table [Clients]

The query requests that only [ClientID] and [TaxCode] columns be retrieved, both of which are found in the leaf page of the non-clustered index. Thus, in order to retrieve this information, the non-clustered index is sufficient, and hence the query execution plan (figure 6.2) shows that only an index seek operation has been performed:

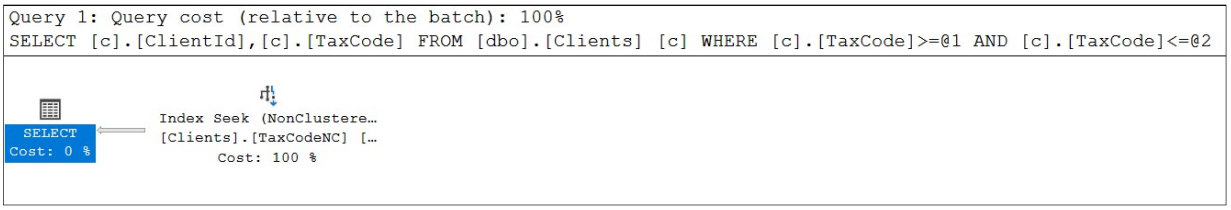

Figure 6.2. Execution plan of the range query on table [Clients]

The IO statistics message of the query is given below:

```
(31000 rows affected)
Table 'Clients'. Scan count 1, logical reads 86, physical reads 1, read-ahead reads
105, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
```

Even-though this query returned 31,000 rows, it made use only of the non-clustered index as this index was essentially "covering" the needs of our query.

This scenario is a good introduction to the concept of covering indexes. 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 (hence the term "covering index" or "covered query"), there is no need to go to the base table and do a key lookup operation to retrieve those necessary columns.

Let's consider the following query (figure 6.3):

```
SET STATISTICS IO ON;
GO

SELECT [c].[ClientId], [c].[TaxCode], [c].[Name]
FROM [dbo].[Clients] AS [c]
WHERE [c].[TaxCode] BETWEEN '60904989375' AND '99733993541'
GO
```

Figure 6.3. Range query on table [Clients] with column [Name] in the SELECT list

This is the same as the previous query in figure 6.1, with the difference that here also the client name is requested to be retrieved. Now SQL Server must go to the clustered index in order to retrieve the [Name] column, as it is not located in the non-clustered index. Since the index seek together with the key lookup operation is too expensive, SQL Server opts for a table scan instead. Below are given the execution plan (figure 6.4) and IO statistics message, respectively:

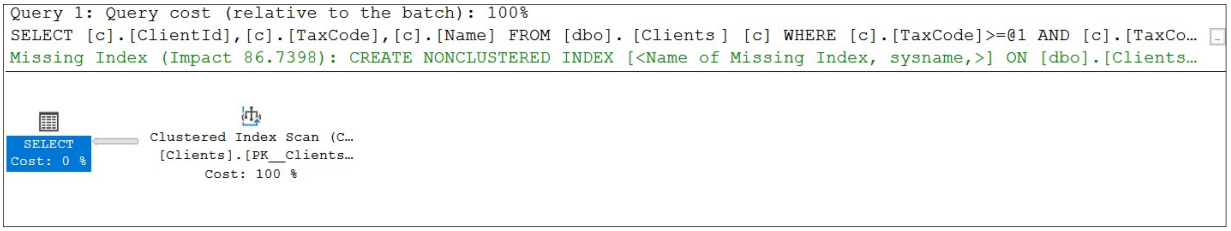

Figure 6.4. Execution plan of the range query on table [Clients] with [Name] column included in the SELECT list

```
(31000 rows affected)
Table 'Clients'. Scan count 1, logical reads 4016, physical reads 2, read-ahead
reads 4015, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
```

Let's try to add a new, covering index for this query, that will contain also the [Name] column in its leaf level (figure 6.5):

```
CREATE NONCLUSTERED INDEX TaxCodeNCNameIncl
on [dbo].[Clients] (TaxCode)
INCLUDE ([Name])
GO
```

Figure 6.5. Creation of the [TaxCodeNCNameIncl] non-clustered index

The execution plan (figure 6.6.) for the same query now uses this new non-clustered index in an index seek operation to retrieve all the necessary information:

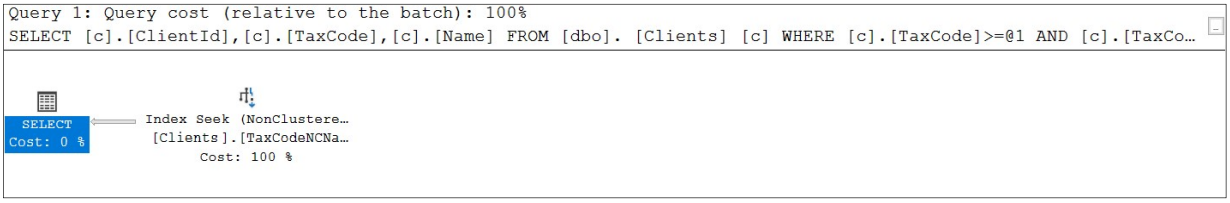

Figure 6.6. Execution plan of the range query after the creation of the [TaxCodeNCNameIncl] non-clustered index

```
(31000 rows affected)
Table 'Clients'. Scan count 1, logical reads 713, physical reads 2, read-ahead reads
635, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
```

Covering indexes are 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).

Additionally, our last example also shows another drawback in the creation of numerous indexes. Namely, index [TaxCodeNCNameIncl] now contains another copy of the [TaxCode] and [ClientID] columns. If there appears another query to tune, similar to the one in figure 6.3 but requesting column [Phone] in the SELECT list instead of [Name], a new covering index [TaxCodeNCPhoneIncl] might be introduced for this query. But this would increase the redundancy for columns [TaxCode] and [ClientId] even more. In that case, there would be four copies of these two columns: in the clustered index, in the [TaxCodeNC] index, in the [TaxCodeNCNameIncl] index, and in the last index introduced [TaxCodeNCPhoneIncl].

After having examined the internals of indexes in Microsoft SQL Server, their limitations in queries that return larger record sets, as well as briefly looking at the powerful concept of covering indexes, the next question we will attempt to answer in this research work is: what techniques should be followed in order to achieve better, more predictable usage of indexes in queries and thus improve performance?

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 practise 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.

Having too many indexes on a server is a quite common scenario that happens when a variety of different sources all propose different indexes for one or few particular queries, and those are executed without a more thorough analysis of what indexes are already present in the server, and if the different indexes proposed introduce too much redundancy in terms of containing many similar columns . For example, if different people work on performance tuning of different parts of the same application, proposing indexes that benefit their own siloed part of the system; if there are tools that suggest indexes for a particular query in question like the hints in SHOWPLAN or other third-party tools; if the missing index DMVs are checked and

abided to without deeper analysis – all of these sources may contribute to there being too many indexes on the server. Even-though these sources may all suggest indexes that benefit the particular queries analysed by them, if there is no deeper analysis on server level, rather than just on query level, then all this added overhead in the form of indexes will need to be maintained, stored, logged, to the point that it might hurt the server's overall performance.

Let's turn to the analysis of the following queries, which we assume were defined a most critical during the workload analysis on the database (figure 6.7 and figure 6.8):

```
--Search clients by name and surname
SELECT [c].*
FROM [dbo].[Clients] AS [c]
WHERE [c].[Surname] = N'Haxhijaha' AND [c].[Name] = N'Blerta';
GO

-- Search only active clients by name and surname
SELECT [c].*
FROM [dbo].[Clients] AS [c]
WHERE [c].[Surname] = N'Aliu' AND [c].[Name] = N'Sara' AND [c].[IsActive] = 1;
GO

--Search clients by surname only
SELECT [c].*
FROM [dbo].[Clients] AS [c]
WHERE [c].[Surname] = N'Aliu';
GO
```

Figure 6.7. Most critical queries identified – Part I

```
--Search clients by surname in a specific range; retrieve name, surname and phone
SELECT [c].[Surname], [c].[Name], [c].[Phone]
FROM [dbo].[Clients] AS [c]
WHERE [c].[Surname] LIKE '[A-E]%'
ORDER BY [c].[Surname], [c].[Name];
GO

--Search clients by surname in a specific range; retrieve name, surname and tax code
SELECT [c].[Surname], [c].[Name], [c].[TaxCode]
FROM [dbo].[Clients] AS [c]
WHERE [c].[Surname] LIKE '[A-E]%'
ORDER BY [c].[Surname], [c].[Name];
GO

-- Search clients by surname in a specific range, retrieve name, surname and status
SELECT [c].[Surname], [c].[Name], [c].[IsActive]
FROM [dbo].[Clients] AS [c]
WHERE [c].[Surname] LIKE '[S-Z]%'
ORDER BY [c].[Surname], [c].[Name], [c].[IsActive];
GO

-- Number of clients in a specific range
SELECT COUNT(*) AS [Total Number of Clients]
FROM [dbo].[Clients] AS [c]
WHERE [c].[Surname] LIKE '[F-M]%';
GO
```

Figure 6.8. Most critical queries identified – Part II

If the only indexes present in the database for table [Clients] are those given in table 6.1 (retrieved from sys.indexes), then the queries have the following execution plans (figure 6.9 and figure 6.10):

Name	IndexID	Type	TypeDescription	IsUnique	IsPrimaryKey
ClientIDPK	1	1	Clustered	1	1
TaxCodeNC	2	2	Nonclustered	1	0

Table 6.1. Indexes present on table [Clients]

As expected, all the queries make a clustered index scan because the only currently available non-clustered index on [TaxCode] is not usable. In order to fix this, it is easy to just follow the advice to put an index on the WHERE clause columns of these critical queries.

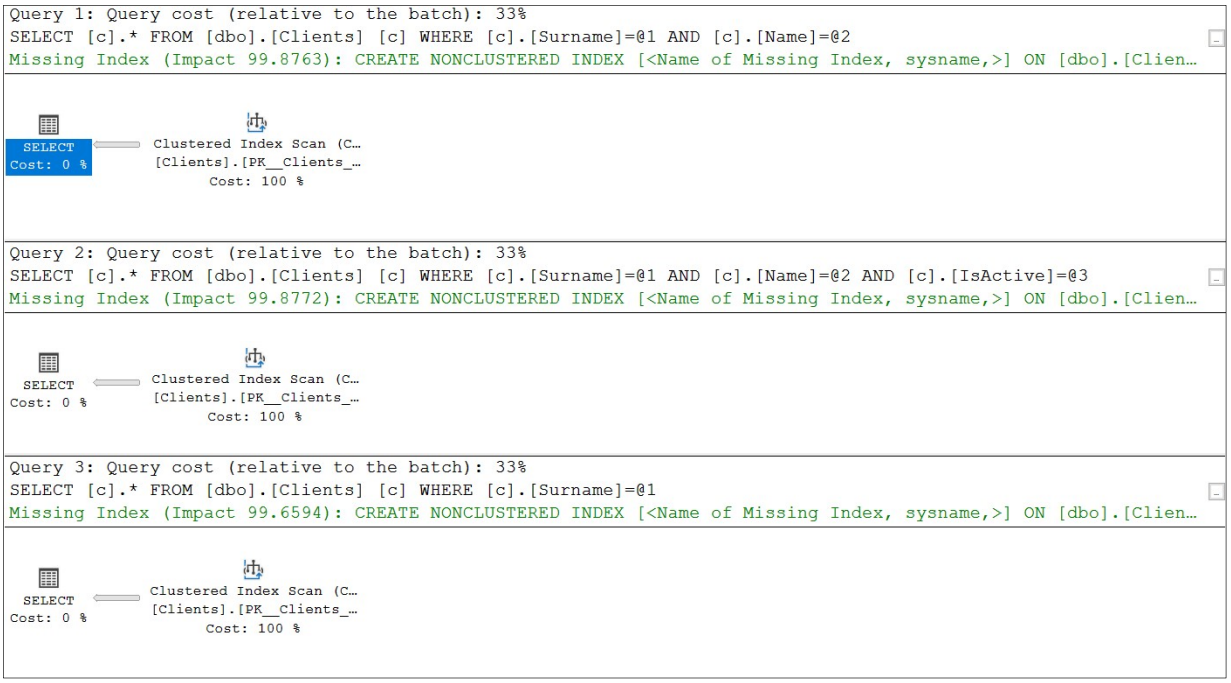

Figure 6.9. Execution plan of critical queries - Part I

Given that we have critical queries that reference different columns/set of columns in the WHERE clause, the list of indexes to create would become the following:

- (Surname)
- (Surname, Name, IsActive)

- (Surname, Name) INCLUDE (TaxCode)
- (Surname, Name) INCLUDE (Phone)

When the queries from figure 6.7 and 6.8 are executed again after creating these indexes, the execution plans displayed are also different (figure 6.11 and figure 6.12). They now make use of the newly created indexes.

Let's have a look at what was the cost in terms of storage after adding the new indexes. Table 6.2 summarizes the number of levels and pages for each of the four indexes, obtained through the *dm_db_index_physical_stats* DMV that we have used earlier in this paper when examining the clustered index and the non-clustered index on [TaxCode].

Figure 6.10. Execution plan of critical queries - Part II

Index	Depth	Level	Rows	Pages
[Surname]	3	0	80000	1295
	3	1	1295	24
	3	2	24	1
[Surname],	4	0	80000	2504
[Name],	4	1	2504	85
[IsActive]	4	2	85	7
	4	3	7	1
[Surname],	4	0	80000	2670
[Name]	4	1	2670	90
INCLUDE	4	2	90	7
[Phone]	4	3	7	1
[Surname],	4	0	80000	2585
[Name]	4	1	2585	87
INCLUDE	4	2	87	7
[TaxCode]	4	3	7	1

Table 6.2. Page, row, level and depth number of the newly created indexes

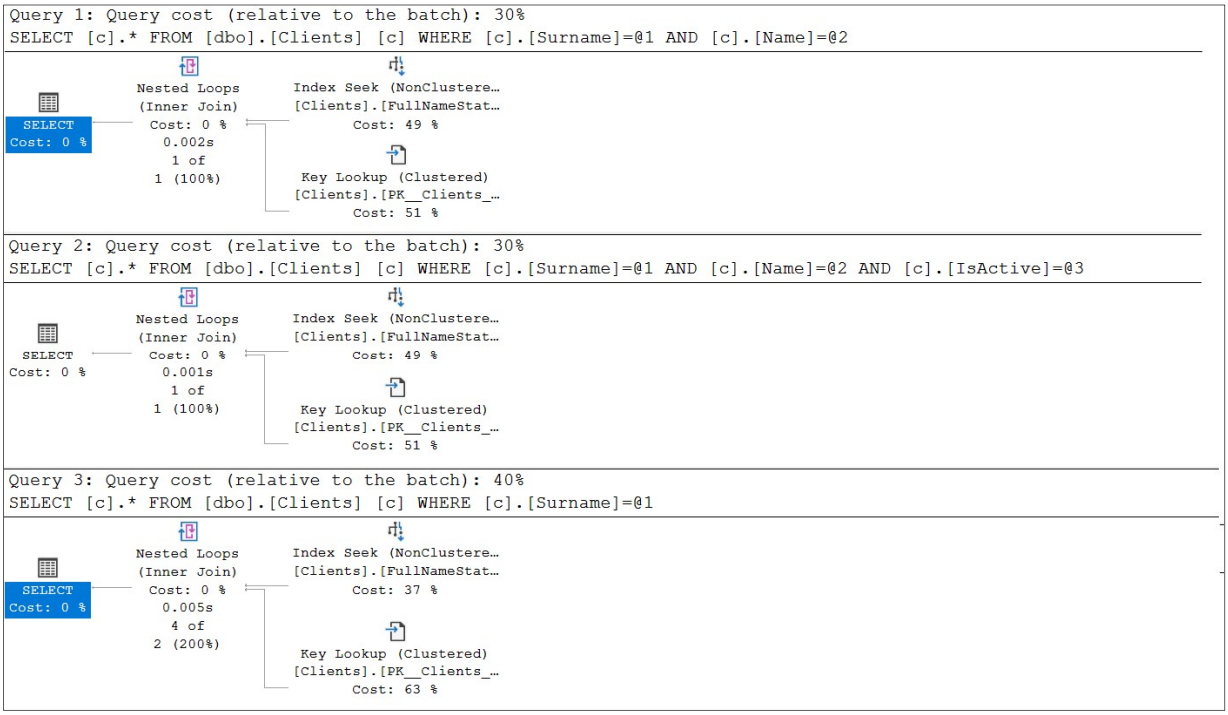

Figure 6.11. Execution plan of critical queries after the creation of new indexes - Part I

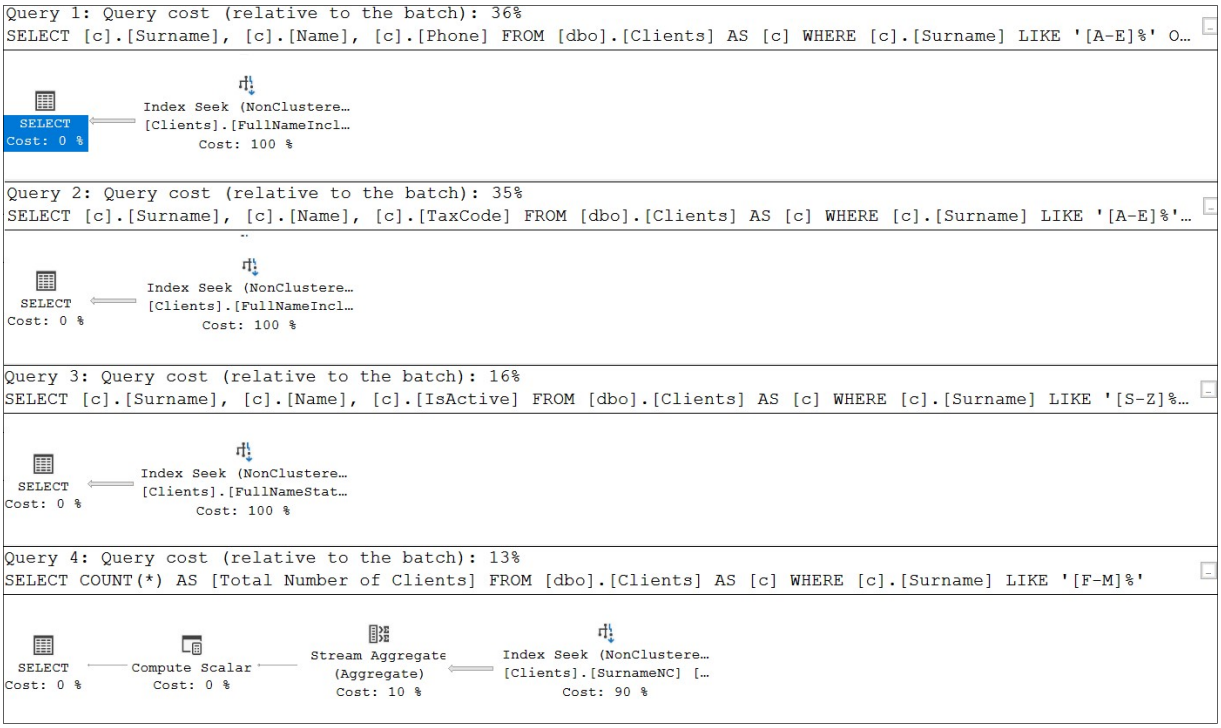

Figure 6.12. Execution plan of critical queries after the creation of new indexes - Part II

The total number of pages used by the four indexes is 9,365 pages, each at 8KB. In megabytes, that's around 73MB 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.

Even-though we managed to tune the individual queries in terms of index usage, this might not be an adequate tuning on server level. As already discussed, adding indexes for all WHERE clause columns in queries costs in terms of maintenance, storage space, cache, backups etc. 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 follow after the process of tuning on query level. In our thesis 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 on server level, or as we call it index fusion or index consolidation. We will aim to show how this process can also address the issues we have raised in this paper so far regarding indexes: redundancy in repeating columns and the high cost of maintenance when numerous indexes are created in the server.

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 table 6.3. Looking at the indexes together, we next identify 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].

Name	Key column(s)	Included column(s)
SurnameNC	Surname	/
FullNameStatusNC	Surname, Name, IsActive	/
FullNameIncludePhoneNC	Surname, Name	Phone
FullNameIncludeTaxCodeNC	Surname, Name	TaxCode

Table 6.3. Indexes for the individual critical queries

So 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 [FullNameIncludeTaxCodeNC]. 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 (figure 6.13):

```
CREATE INDEX [ConsolidatedIndexNC]
ON [dbo].[Clients] ([SurName], [Name], [IsActive])
INCLUDE ([Phone], [TaxCode]);
GO
```

Figure 6.13. Consolidated index creation

In order to test our new consolidated index, we disable the four existing indexes from table 6.3. 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 queries given in figure 6.7 and 6.8 and observe their execution plans given in figures 6.14 and 6.15 below. They indeed show that the new index [ConsolidatedIndexNC] is used in all five critical queries. So now the server uses a single non-clustered index instead of the four different non-clustered indexes it was utilizing previously. Those non-clustered indexes were all suitable for query level performance tuning, but on server level, having only one index to maintain is highly preferred in terms of all the different costs associated with indexes, even-though this new index is bigger and may require a few more IOs.

The number of levels and pages of index [ConsolidatedIndexNC] is given in table 6.4, as returned by the dm_db_index_physical_stats DMV. In order to calculate how much storage space was saved, again the total number of pages is summed up and the result multiplied by 8K. This calculates to around 22.4 MB of storage. The storage space occupied by the four previous indexes was 73MB, so the storage space was reduced to less than a third of the initial space. The numbers in this example might not seem to make a big difference, but let us 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, of maintenance in general.

Index	Depth	Level	Rows	Pages
[ConsolidatedIndexNC]	4	0	80000	2762
	4	1	2762	93
	4	2	93	7
	4	3	7	1

Table 6.4. Page, row, level and depth number of the consolidated index

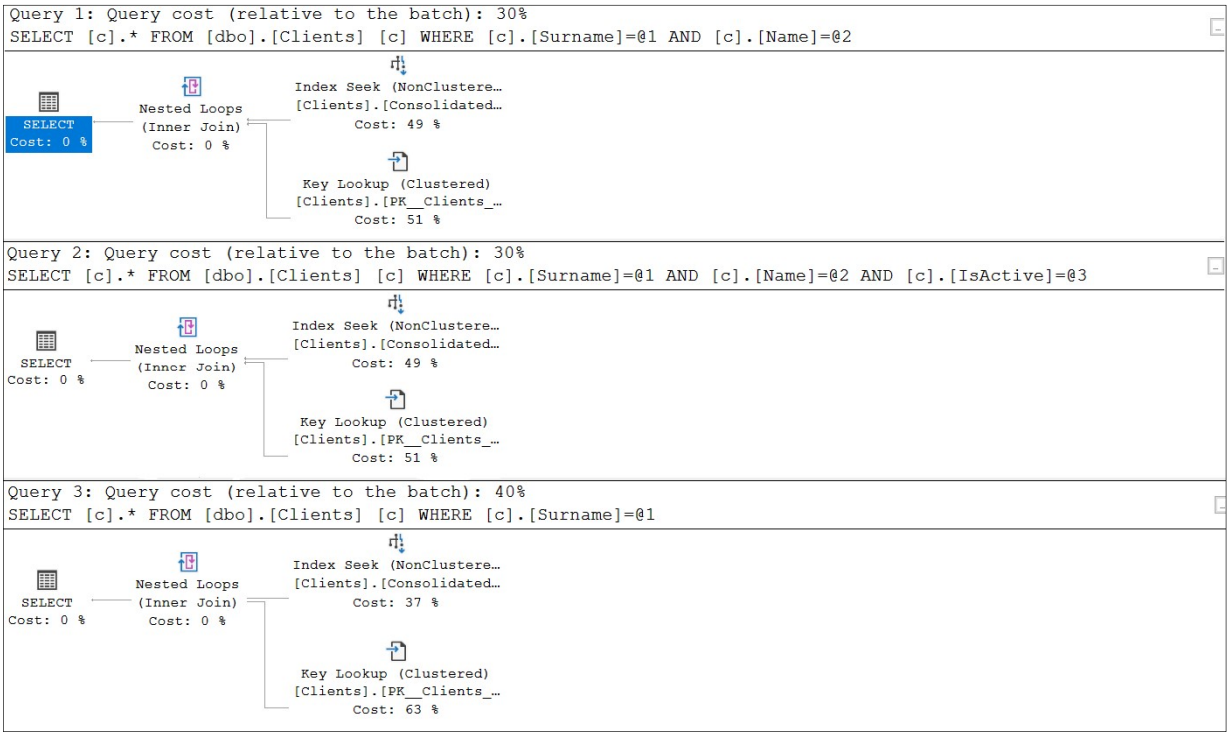

Figure 6.14. Execution plan of critical queries after creation of the consolidated index - Part I

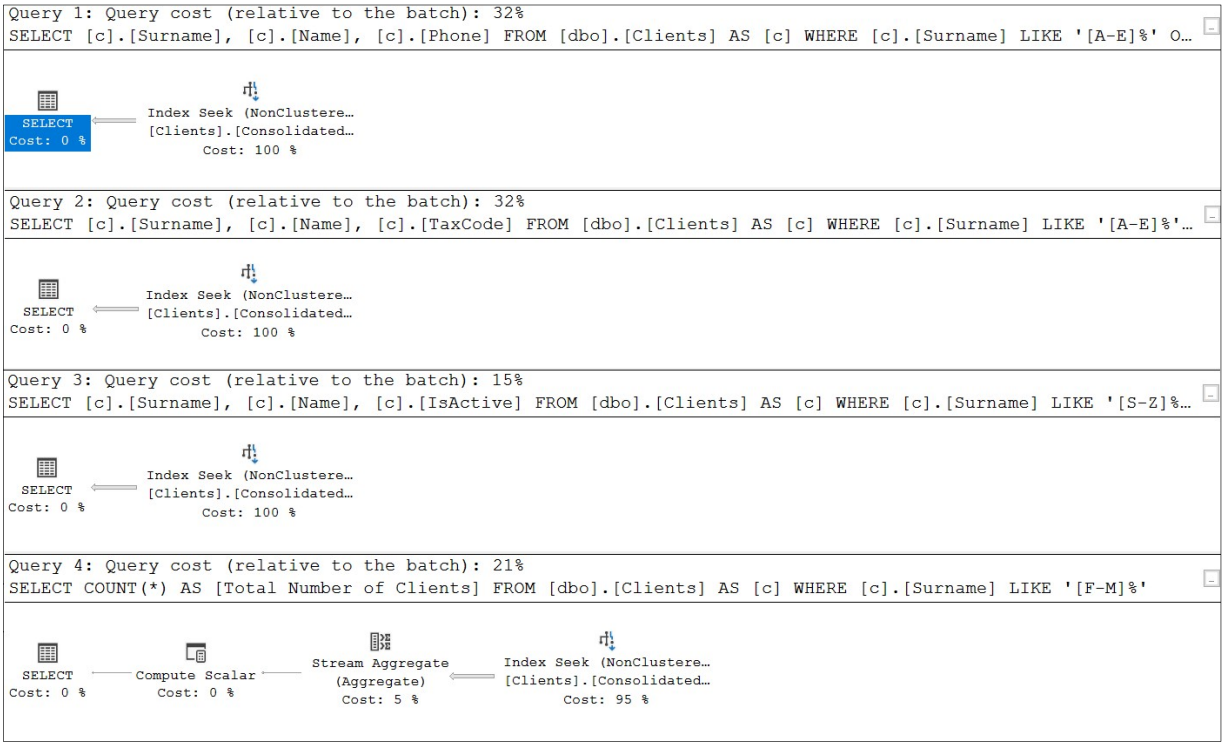

Figure 6.15. Execution plan of critical queries after creation of the consolidated index - Part II

4.3. Identifying performance issues with wait statistics

In the previous sections we have looked at indexes and indexing strategies, claiming them to be an important aspect to focus on in order to secure good database performance. In this next section, we will attempt to tackle the query performance subject from a more general level. Namely, we will attempt to answer the question on how to identify the bottlenecks of a SQL Server query execution, when we observe that they are taking longer than usual.

It is a common scenario that DBAs or maybe even database developers are asked to perform some troubleshooting of the server, when there are observations or complaints that different database operations are running very slow or seem blocked altogether. Where to start troubleshooting in these cases? There are many different components of SQL Server that troubleshooting can start at - from checking different hardware configurations, the memory, the I/O subsystem, the networking; the indexing strategies, the fragmentation in indexes. Or perhaps there is something wrong with the application code, or the way that users are running queries. Without a clear direction on where to start looking, troubleshooting can easily become a lengthy process with time lost on examining or even fixing components that are not the actual source of the problem.

This master thesis will propose the usage of Wait statistics in identifying performance bottlenecks, because they point to what SQL Server has been waiting on while executing queries. Only when there is proper information on where the bottleneck resides, measures can be taken to address that particular bottleneck, amend it and increase performance.

A short introduction on how wait statistics work, as well as a brief description of the wait types encountered in our experiments were provided in the *Wait Statistics* section of this paper.

Next, let's consider a data table [WaitStatsTest], composed of only two columns: an identity column [ID], on which a clustered index is created, and a char(20) column [Description], as in figure 7.1:

```
CREATE TABLE [WaitStatsTest](
[ID] int IDENTITY(1,1) NOT NULL
, [Description] CHAR(20) NOT NULL)
GO

CREATE UNIQUE CLUSTERED INDEX WaitStatsTest_CLIX
```

```
ON dbo.WaitStatsTest (ID)
GO
```

Figure 7.1. Test table [WaitStatsTest] with a clustered index

Next, let's assume that there are 100 concurrent connections, each trying to insert 10.000 rows in this table (figure 7.2):

```
SET NOCOUNT ON;

INSERT INTO dbo.WaitStatsTest ([Description])
VALUES ('testing wait stats')
GO 10000
```

Figure 7.2. Inserting default values into [WaitStatsTest]

Since the records for this table are very small, many can fit into a single 8K page. Being that the clustering key is monotonically increasing, records will be inserted consecutively at the end of a page until that page is filled up. There will not be a problem with locks, since locking will be on row level, but there will be contention on latches.

To examine what types of waits are being generated by the workload, the sys.dm_os_waiting_tasks DMV will be used. This DMV retrieves information about every thread on the server that is currently suspended. All threads, no matter what scheduler they are on, if they are on the waiter list in state suspended, will show up in the output of this DMV. sys.dm_os_waiting_tasks can be joined with some other useful DMVs in order to get some other useful information, as in the script in figure 7.3 below:

```
SELECT
 [wt].[session_id],
 [wt].[exec_context_id],
 [wt].[wait_duration_ms],
 [wt].[wait_type],
 [wt].[blocking_session_id],
 [wt].[resource_description],
 [es].[program_name],
 [sql].[text]
 FROM sys.dm_os_waiting_tasks [wt]
 INNER JOIN sys.dm_exec_sessions [es] ON
 [wt].[session_id] = [es].[session_id]
 INNER JOIN sys.dm_exec_requests [er] ON
 [es].[session_id] = [er].[session_id]
 OUTER APPLY sys.dm_exec_sql_text ([er].[sql_handle]) [sql]
 WHERE [es].[is_user_process] = 1
 ORDER BY [wt].[session_id], [wt].[exec_context_id]
 GO
```

Figure 7.3. Diagnostic query for waiting tasks in SQL Server

When executing this script on the server where only the workload described above is running, the following result set is obtained (fig 7.4, abbreviated to show only the first 10 rows, for space considerations):

	session_id	exec_context_id	wait_duration_ms	wait_type	blocking_session_id	resource_description	program_name	text
1	59	0	9	PAGELATCH_EX	NULL	8:1:3954	SQLCMD	(@1 varchar(8000))INSERT INTO [dbo].[WaitStatsTest]([Descriptio
2	60	0	8	PAGELATCH_EX	NULL	8:1:3954	SQLCMD	(@1 varchar(8000))INSERT INTO [dbo].[WaitStatsTest]([Descriptio
3	61	0	9	PAGELATCH_EX	NULL	8:1:3954	SQLCMD	(@1 varchar(8000))INSERT INTO [dbo].[WaitStatsTest]([Descriptio
4	62	0	7	PAGELATCH_EX	NULL	8:1:3954	SQLCMD	(@1 varchar(8000))INSERT INTO [dbo].[WaitStatsTest]([Descriptio
5	63	0	8	PAGELATCH_EX	NULL	8:1:3954	SQLCMD	(@1 varchar(8000))INSERT INTO [dbo].[WaitStatsTest]([Descriptio
6	64	0	8	PAGELATCH_EX	NULL	8:1:3954	SQLCMD	(@1 varchar(8000))INSERT INTO [dbo].[WaitStatsTest]([Descriptio
7	65	0	8	PAGELATCH_EX	NULL	8:1:3954	SQLCMD	(@1 varchar(8000))INSERT INTO [dbo].[WaitStatsTest]([Descriptio
8	66	0	8	PAGELATCH_EX	NULL	8:1:3954	SQLCMD	(@1 varchar(8000))INSERT INTO [dbo].[WaitStatsTest]([Descriptio
9	67	0	8	PAGELATCH_EX	NULL	8:1:3954	SQLCMD	(@1 varchar(8000))INSERT INTO [dbo].[WaitStatsTest]([Descriptio
10	68	0	7	PAGELATCH_EX	NULL	8:1:3954	SQLCMD	(@1 varchar(8000))INSERT INTO [dbo].[WaitStatsTest]([Descriptio

Figure 7.4. Waiting tasks diagnostic query results

It can be observed that there are several PAGELATCH_EX wait types occurring at the moment the script was executed. The resource_description column shows that the resource being waited for is page (8:1:3954), where 8 is the database id, 1 is the file number, and 3954 is the page number on that file. In order to check what type of page is this, the DBCC PAGE command [15] can be used (figure 7.5), which gives the output given in figure 7.6 for the page header:

```
DBCC TRACEON(3604)
GO
DBCC PAGE(8,1,3954,0)
GO
```

Figure 7.5. DBCC PAGE command for the resource being waited on

```
m_pageId = (1:3954)                    m_headerVersion = 1                    m_type = 2
m_typeFlagBits = 0x0                   m_level = 1                            m_flagBits = 0x8200
m_objId (AllocUnitId.idObj) = 129      m_indexId (AllocUnitId.idInd) = 256
Metadata: AllocUnitId = 72057594046382080
Metadata: PartitionId = 72057594041139200                                     Metadata: IndexId = 1
Metadata: ObjectId = 309576141         m_prevPage = (1:3684)                  m_nextPage = (1:4224)
pminlen = 11                           m_slotCnt = 269                        m_freeCnt = 4599
m_freeData = 6938                      m_reservedCnt = 0                      m_lsn = (29414:400:28)
m_xactReserved = 0                     m_xdesId = (0:0)                       m_ghostRecCnt = 0
m_tornBits = -324851023                DB Frag ID = 1
```

Figure 7.6. Header from DBCC PAGE command for page (1:3954)

Investigating [ObjectID]=309576141 via the object_name() function in SQL Server, we confirm that this page belongs to table [WaitStatsTest], which is a clustered index. Since the records being inserted are very small, many inserts are happening on the same page and only one thread at a time can have an exclusive latch on the page, whereas all the rest of them are

queuing up.

In order to examine the wait stats that have accumulated during the whole duration of our workload, the following diagnostic query will be used (taken from [15], pp. 27-28). It uses the sys. dm_os_wait_stats DMV, which keeps track of all the aggregated waits that have occurred on the server since the SQL Server was last restarted, or the DMV manually cleared. Notice that this version of the query filters out the benign waits, so that they don't clutter up the result set. In order to get the wait stats from our workload only, the sys. dm_os_wait_stats DMV is cleared at the beginning of the script:

```
DBCC SQLPERF('sys.dm_os_wait_stats', CLEAR)
SELECT
wait_type,
wait_time_ms,
wait_time_ms * 100.0 / SUM(wait_time_ms) OVER() AS percentage,
signal_wait_time_ms * 100.0  / wait_time_ms as signal_pct
FROM sys.dm_os_wait_stats
WHERE wait_time_ms > 0
AND wait_type NOT IN (
'BROKER_DISPATCHER',          'BROKER_EVENTHANDLER',
'BROKER RECEIVE WAITFOR',    'BROKER TASK STOP',
'BROKERTOFLUSH',                    'BROKER TRANSMITTER',
'CHECKPOINT_QUEUE',                 'CHKPT',
'CLR_AUTO_EVENT',                   'CLR_MANUAL_EVENT',
'CLR_SEMAPHORE',                    'DBMIRROR_DBM_EVENT',
'DBMIRROR_DBM_MUTEX',         'DBMIRROR_EVENTS_QUEUE',
'DBMIRROR_WORKER_QUEUE',      'DBMIRRORING_CMD',
'DIRTY_PAGE POLL',                  'DISPATCHER_QUEUE_SEMAPHORE', 'EXECSYNC',
'FST4GENT',
'FT_IFTS_SCHEDULER_IDLE_WAIT',  'FT_IFTSHC_MUTEX',
'HADRCLUSAPI_CALL',                 'HADRFILESTREAMIOMGRIOCOMPLETION',
'HADR_LOGCAPTURE_WAIT',             'HADR_NOTIFICATION_DEQUEUE',
'HADR TIMER TASK',                  'HADR WORK QUEUE',
'KSOURCEWAKEUP',                    'LAZYWRITER_SLEEP',
'LOGMGR_QUEUE',                     'ONDEMAND_TASK_QUEUE',
'PWAIT_ALL_COMPONENTS_INITIALIZED',  'ODS_ASYNC_QUEUE',
'OPS_PERSIST_TASK_MAIN_LOOP_SLEEP',  'QDS_SHUTDOWN_ENEUE',
'QDS_CLEANUP_STALE_QUERIES_TASK_MAIN_LOOP_SLEEP',  'REWEST_FOR_DEADLOCK_SEARCH',
'RESOURCE_QUEUE',                   'SERVER_IDLE_CHECK',
'SLEEPBPOOLFLUSH',                  'SLEEP_BUFFERPOOL_HELPLW',
'SLEEP_DBSTARTUP',                  'SLEEP_DCOMSTARTUP',
'SLEEP MASTERDBREADY',        'SLEEP MASTERMDREADY',
'SLEEP_MASTERUPGRADED',             'SLEEP_MSDBSTARTBP',
'SLEEP_SYSTEMTASK',                 'SLEEP_TASK',
'SLEEP_TEMPDBSTARTUP',        'SLEEP_WORKSPACE_ALLOCATEPAGE',
'SNI_HTTP_ACCEPT',                  'SP_SERVER_DIAGNOSTICS_SLEEP',
'SO.LTRACE_BUFFER_FLUSW',    'SQLTRACE_INCREMENTAL_FLUSH_SLEEP',
'NLTRACE_WAIT_ENTRIES' ,      'WAIT_FOR_RESULTS',
'WAITFOR',                                'WAITFOR TASKSHUTDOWN',
'WAITXTP_HOSTWAIT',                 'WAITXTP_OFFLINECKPT_NEW_LOC',
'WAIT_XTP_CKPT_CLOSE',        'XE_DISPATCHER_JOIN',
'XE_DISPATCHER WAIT',         'XE_TIMER_EVENT')
ORDER BY percentage DESC
```

Figure 7.7. Diagnostic query for the wait statistics accumulated in SQL Server

The result of this query after the workload finished executing are given in figure 7.8 (the top five wait types highest in percentage). The top two occurring wait types are the PAGELATCH_EX and PAGELATCH_SH wait types, together contributing in more than 80% of the total waits aggregated.

	wait_type	wait_time_ms	percentage	signal_pct
1	PAGELATCH_EX	19853880	66.925067169017828	1.266951346537805
2	PAGELATCH_SH	4426351	14.920702552279415	3.991888578199062
3	WRITELOG	972296	3.277494127503911	38.677419222129886
4	LATCH_SH	862542	2.907526452567406	7.050207410189880
5	LATCH_EX	746488	2.516322227235471	1.904384263377308

Figure 7.8. Accumulated wait statistics

In high throughput/high concurrency OLTP workloads, where the possibility for concurrency increases due to the higher number of CPU cores on modern servers, contention points arise on memory structures which must be accessed serially (in exclusive modes). This is especially true, as our example shows, for clustered indexes with clustering keys of sequentially increasing pattern, found on tables with narrow rows that can fit in a single page in a large number.

Via SQL Server's wait statistics, contention problems which might lead to performance issues can thus be discovered, and appropriate action taken in order to alleviate them. In the case of high PAGELATCH_XX waits slowing down performance, alternatives such as splitting the insertion points in the table through partitioning, or maybe even charting the inserts to multiple tables can help reduce the contention issue.

Another example where wait statistics can show potential problems in the query execution will be described next.

Let's consider a table [WaitStatsTest2] with 3.000 rows in it (figure 7.9), but with incorrect statistics on its clustering key column, where the optimizer assumes there are 5.000.000 records in the table (for test purposes, achieved via the undocumented SQL Server command UPDATE STATISTICS … WITH ROWCOUNT, PAGECOUNT):

```
CREATE TABLE [WaitStatsTest2] (
[ID] INT IDENTITY,
```

```
[SomeInteger] INT,
[SomeText] NVARCHAR (100)
)
GO

CREATE UNIQUE CLUSTERED INDEX WaitStatsTest2_CLIX
ON WaitStatsTest2 (ID)
GO


INSERT INTO [WaitStatsTest2] ([SomeInteger], [SomeText])
SELECT top 3000 [ClientId], [Name]
FROM [Clients]
GO

UPDATE STATISTICS [WaitStatsTest2] ([WaitStatsTest2_CLIX])
WITH ROWCOUNT = 5000000, PAGECOUNT = 500000;
GO
```

Figure 7.9. Table [WaitStatsTest2] with incorrect statistics

The workload that will be executed against table [WaitStats2] is given in figure 7.10:

```
SET NOCOUNT ON;
GO

DECLARE @SomeText NVARCHAR (100);

SELECT TOP (500)
    @SomeText = [SomeText]
FROM [WaitStatsTest2]
ORDER BY NEWID() DESC;
GO 50000
```

Figure 7.10. Workload for [WaitStatsTest2] table

Executing the query from figure 7.3 for waiting tasks during the execution of this workload, we observe that there are CXPACKET waits occuring on the server.

	session_id	exec_context_id	wait_duration_ms	wait_type	blocking_session_id	resource_description	program_name	text
1	52	0	6	CXPACKET	52	exchangeEvent id=Port14...	Microsoft SQL Server Management Studio - Query	DECLARE @SomeText NVAR
2	52	2	6	CXPACKET	52	exchangeEvent id=Port14...	Microsoft SQL Server Management Studio - Query	DECLARE @SomeText NVAR
3	52	3	6	CXPACKET	52	exchangeEvent id=Port14...	Microsoft SQL Server Management Studio - Query	DECLARE @SomeText NVAR
4	52	4	5	CXPACKET	52	exchangeEvent id=Port14...	Microsoft SQL Server Management Studio - Query	DECLARE @SomeText NVAR
5	52	5	5	CXPACKET	52	exchangeEvent id=Port14...	Microsoft SQL Server Management Studio - Query	DECLARE @SomeText NVAR
6	52	6	5	CXPACKET	52	exchangeEvent id=Port14...	Microsoft SQL Server Management Studio - Query	DECLARE @SomeText NVAR
7	52	7	5	CXPACKET	52	exchangeEvent id=Port14...	Microsoft SQL Server Management Studio - Query	DECLARE @SomeText NVAR
8	52	8	5	CXPACKET	52	exchangeEvent id=Port14...	Microsoft SQL Server Management Studio - Query	DECLARE @SomeText NVAR

Figure 7.11. Waiting tasks diagnostic query results when incorrect statistics are present

After the workload execution is finished, the accumulated wait stats are also retrieved via the query from figure 7.7 and presented below in figure 7.12. The top wait type accumulated for this workload, CXPACKET, contributes to more than 50% of the total waits. The CXPACKET wait type was described in section *Wait Statistics* of this thesis.

	wait_type	wait_time_ms	percentage	signal_pct
1	CXPACKET	1077285	51.296623175273283	2.057208630956524
2	REQUEST_FOR_DEADLOCK_SEARCH	215207	10.247420491031655	100.000000000000000
3	HADR_FILESTREAM_IOMGR_IOCOMPLETION	214947	10.235040181247735	0.023261548195601
4	LOGMGR_QUEUE	214885	10.232087953530031	0.001861460781348
5	DIRTY_PAGE_POLL	214819	10.228945259507958	0.002793049031975

Figure 7.12. Accumulated wait stats on a workload using incorrect statistics

Let's now look at the query execution plan for our query (figure 7.13). As expected, the SQL Server engine opted for a parallel execution, because it assumed that there were 5.000.000 rows in the table. Since we also identified CXPACKET waits occurring, we were indeed expecting that there was a parallel execution taking place. Figure 7.13 shows that SQL Server was estimating 5.000.000 rows in the table, when the actual number was only 3.000 rows.

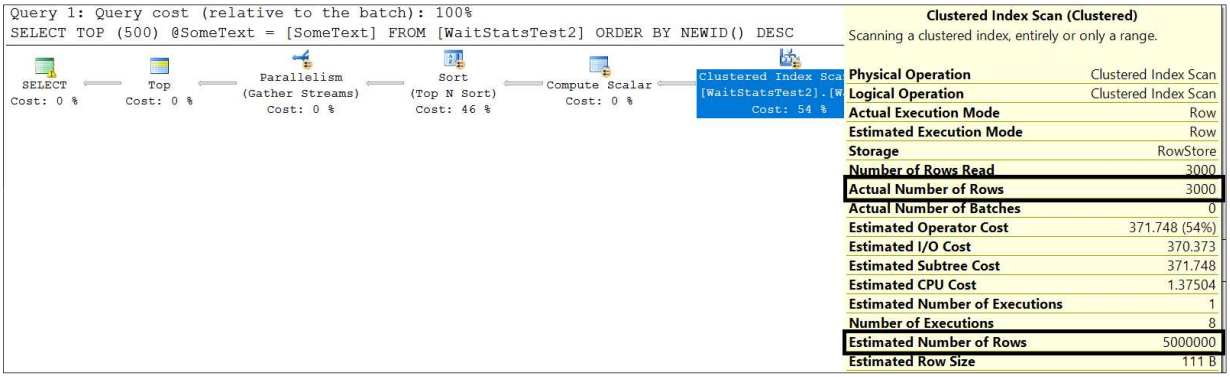

Figure 7.13. Execution plan of the query that uses incorrect statistics

Looking at the properties of the execution plan (figure 7.14), it can be observed that only one thread was doing the work, and that is thread 1. And looking back at the results of the sys.dm_os_waiting_tasks DMV in figure 7.11, it can be observed that only this thread did not appear in the result set, while all the other threads were producing CXPACKET waits while waiting for thread 1 to finish.

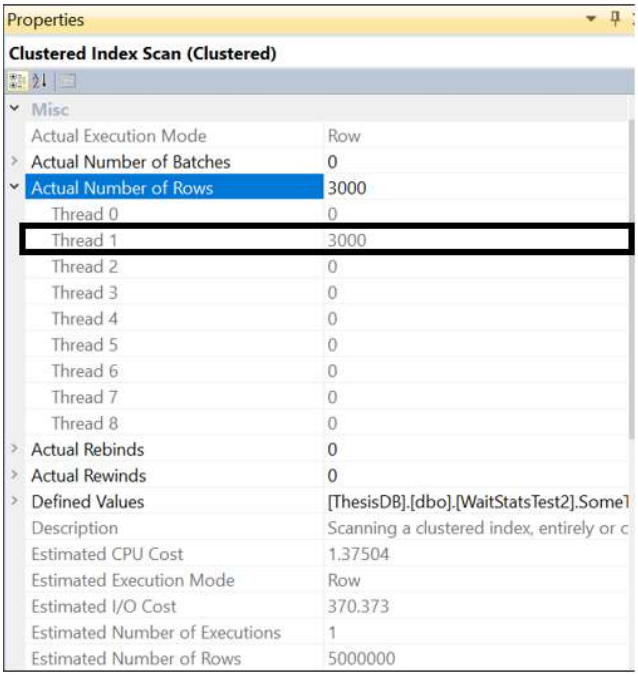

Figure 7.14. Properties of the execution plan

Because this parallel execution was against incorrect statistics, there was a skewed distribution of work to the threads, and actually all threads were waiting idly waiting for thread 1 to complete. So, only one thread was doing all the work, and the parallelism introduced only caused an overhead to the system, without any gains. Figure 7.15 graphically explains the situation of unequal distribution of work amoung threads, making the waiting threads produce CXPACKET waits:

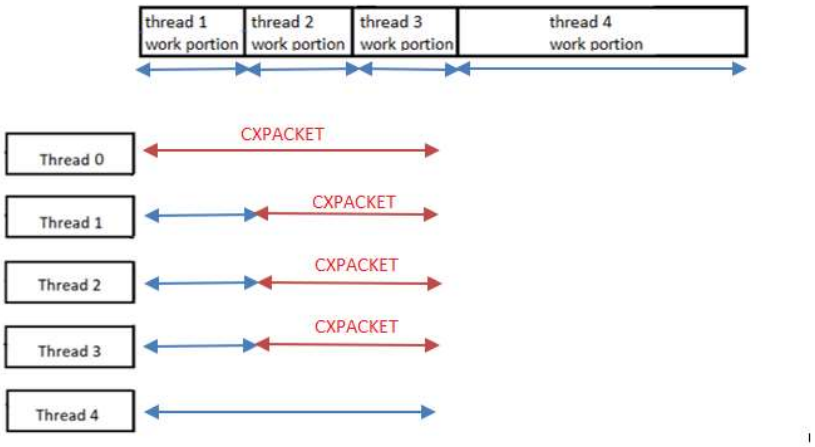

Figure 7.15. Distribution of work among threads when incorrect statistics are present

Now that the wait statistics identified the problem with high CXPACKET waits, which prompted us to look at the execution plan of the query and find out about the incorrect statistics on the [ID] column, let's next investigate the behaviour of the same workload when the statistics are not off, and see if the CXPACKET waits still persist. Table [WaitStatsTest2] is recreated and populated with 3.000 rows, and the statistics are not updated with the [UPDATE STATISTICS WITH ROWCOUNT, PAGECOUNT] command which caused the incorrect statistics in the initial workload. Now when running the query from figure 7.10, the query execution plan (figure 7.16) does not show a parallel operation, and both the estimated and actual number of rows are equal to 3.000 rows, which is the correct number of rows in the table.

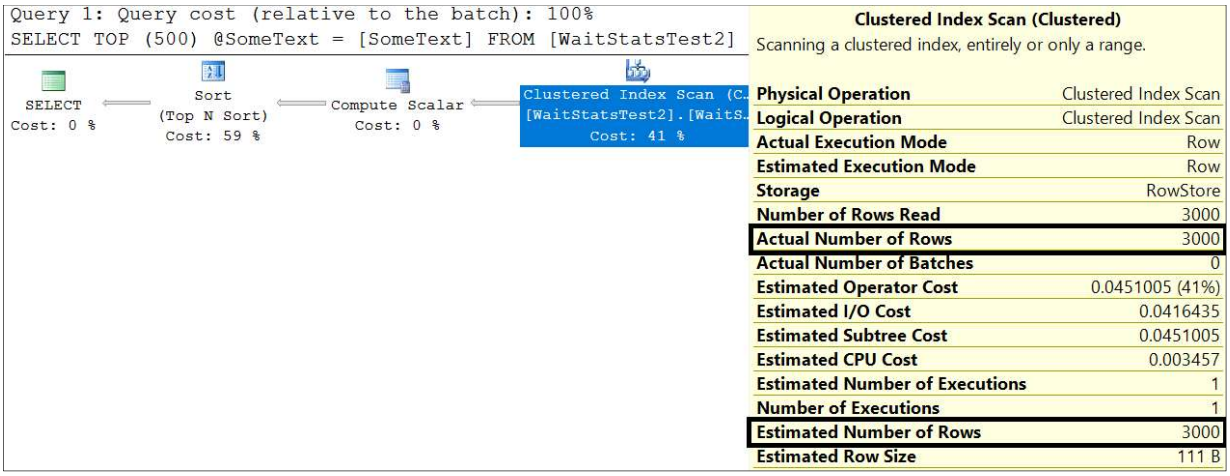

Figure 7.16. Execution plan of the query when statistics are correct

The sys.dm_os_waiting_tasks diagnostic query from fig. 7.3 now gives an empty result set during the whole execution of the workload (the query was executed repeatedly while the workload was running, and no waits were returned).

The dm_os_wait_stats DMV query from figure 7.7 executed after the workload finished, also revealed that there were no CXPACKET waits accumulated (figure 7.17):

	wait_type	wait_time_ms	percentage	signal_pct
1	REQUEST_FOR_DEADLOCK_SEARCH	245394	22.350808802098513	100.000000000000000
2	HADR_FILESTREAM_IOMGR_IOCOMPLETION	243706	22.197063538327018	0.002461982881012
3	LOGMGR_QUEUE	243409	22.170012387059166	0.002875818067532
4	DIRTY_PAGE_POLL	243359	22.165458321189157	0.003698240048652
5	BROKER_TO_FLUSH	121865	11.099624744972311	0.000000000000000

Figure 7.17. Accumulated wait stats when the statistics are correct

The wait statistics analysis once again showed how the waits SQL Server keeps track of in various DMVs, can be a reliable source of information in order to troubleshoot potential issues. In our case, we were observing high CXPACKET waits accumulated after our workload execution. Looking at the execution plan of the query involved, we observed that it had chosen a parallel plan where only one thread was doing all the work, making all the other threads produce CXPACKET waits. This was caused by the incorrect statistics, and repeating the workload with correct statistics in place, had SQL Server choose a serial plan and complete the workload without associated waits of CXPACKET type.

In order to utilize the true power of wait statistics, a good practice to follow is to additionally track wait statistics over time, in order to reveal if there are any trends happening. By watching the trends and wait statistics over time, different patterns may emerge that need to be solved so that there is no performance downgrading.

5. 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. Relational database management systems, as one of the traditional choices for storing data and performing data-related operations, also participate in this trend and are continuously focused on providing solutions that include performance enhancements.

This thesis also focused on performance improvement techniques for data operations, and particularly investigated techniques for ensuring good performance of queries executed in Microsoft SQL Server. It primarily addressed indexes, which although one of the most common measures aimed at performance improvement, still pose challenges to DBAs because of their not always predictable use on production environments.

This thesis attempted to answer the question of why these limitations on index usage occur, by looking at the internals of indexes in SQL Server, and investigating the turning point after which these indexes are no longer used in queries. It demonstrated that there are cases when using an index becomes too expensive, even-though nothing was changed in the way the T-SQL query was written, which previously did make use of the index.

Next, the thesis proposed the index fusion technique in order to overcome this challenge. This ensured a more consistent usage of the indexes on the server, and reduced their overall number, thus also saving on disk space, memory, logging, fragmentation issues, and making the overall maintenance easier. In order to perform index fusion and have these gains on 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 a successful index consolidation on server level.

Finally, this thesis looked at using the wait statistics stored by SQL Server as the starting point for troubleshooting query performance. Since SQL Server performs its own thread scheduling and keeps track of what resources the various threads are waiting on while executing, the wait statistics can be used as a starting point to analyse contention for resources and then apply

appropriate measures to provide for a smother query execution. Since there is a variety of factors that might be responsible for delays during query execution, including but not limited to network latencies, disk I/O pressure, CPU pressure as well as bad indexing strategies., using the wait statistics in the performance tuning methodology is suitable as it pinpoints exactly what is causing the delays. This helps avoid spending futile efforts in trying to discover where the issue is, or even worse, in fixing what is not broken. Except for providing a peek at what is going on in the server in terms of contention for resources and accumulated wait types, it is also possible to proactively track wait statistics over time as this might point to when changes and problems have started to occur, making it possible to map this back to some rollout of new code or some other change in the system.

References

[1] T. Lahdenmaki, M. Leach. Relational Database Index Design and the Optimizers: DB2, Oracle, SQL Server, et al. 1st edition, Wiley-Interscience, 2007

[2] B. Dorr, B. Ward, R. Stonecipher, "SQL 2016 - It Just Runs Faster Announcement", *https://blogs.msdn.microsoft.com/bobsql/2016/06/03/sql-2016-it-just-runs-faster-announcement/*, June 3rd, 2016

[3] PA. Larson, C.Clinciu, E.N. Hanson, A. Oks, S.L. Price, S. Rangarajan, A. Surna, Q. Zhou. "SQL Server Column Store Indexes", Proceedings of the 2011 ACM SIGMOD International Conference on Management of data, June 12-16, 2011, Athens, Greece

[4] J. M. Monteiro, S. Lifschitz, A. Brayner. "An architecture for automated index tuning", SBBD, 2006.

[5] B. Patel, S. Mishra, "Comparing Tables Organized with Clustered Indexes versus Heaps", Microsoft Documentation, 2010

[6] G. Fritchey, S. Dam, "SQL Server 2008 Query Performance Tuning Distilled", Apress, 2009

[7] "Estimate the Size of a Clustered Index", Microsoft Documentation, *https://docs.microsoft.com/en-us/sql/relational-databases/databases/estimate-the-size-of-a-clustered-index?view=sql-server-2017*, 2017

[8] Ch. Shaw, G. Fritchey, C. Bossy, J. Lowell, G. Shaw, J. Johansen, M. Prajdi, W.Pastrick, K. Pot'Vin, J. Strate, H. Roggero, T. Belt, J. Gardner, G. Berry, B. Ball, J. Borland, B. DeBow, L. Davidson. "Pro SQL Server 2012 Practices", 1st edition, Apress, 2012

[9] B. Nevarez, "High Performance SQL Server: The Go Faster Book", 1st edition, Apress, 2016

[10] K. Delaney, C. Freeman. "Microsoft SQL Server 2012 Internals", 1st edition, Microsoft Press, 2013

[11] J. Kehayias, E. Stellato. "SQL Server Performance Tuning Using Wait Statistics: A Beginner's Guide", Simple Talk, 2014

[12] B. Nevarez. "Inside the SQL Server Query Optimizer", Red Gate, 2011

[13] E. Van de Laar. "Pro SQL Server Wait Statistics", 1st edition, Apress, 2015

[14] P.Dave, R. Morelan. "SQL Wait Stats Joes 2 Pros: SQL Performance Tuning Techniques Using Wait Statistics, Types & Queues", CreateSpace Independent Publishing Platform, 2011

[15] J. Strate, T. Krueger. “Expert Performance Indexing for SQL Server 2012”, 1st edition, Apress, 2012

[16] C. G. Corlatan, M.M. Lazar, V. Luca, O. T. Petricica. “Query Optimization Techniques in Microsoft SQLServer”. Database System Journal, 2014, Vol. 5, No. 2, pp. 33-48.

[17] J. Habimana. “Query Optimization Techniques – Tips for Writing Efficient and Faster SQL Queries” International Journal of Scientific and Technology Research, vol. 4, issue 10, October 2015, pp. 22 – 26.

[18] I. Lungu, N. Mercioiu, and V. Vladucu. “Optimizing Queries in SQL Server 2008” Sci-entific Bulletin – Economic Sciences, Vol. 9, No. 15, 2010, pp. 103-108.

[19] Q. Chunxia. "On index-based query in SQL Server database." Control Conference (CCC), 2016 35th Chinese. IEEE, 2016.

[20] D. J. Farrar, A. Nica. Database systems with methodology for automated determination of indexes, Patent No. US 7,406,477 B2, 2008

[21] D. P. Brown, J. Chaware, A.Pradesh, M. Koppuravuri, A. Pradesh. Index Selection in a Database Systems, Patent No. US 7,499,907 B2, 2009

[22] J.M. Monteiro, S. Lifschitz, A.Brayner. “An Architecture for Automated Index Tuning”, In V Ph.D. and M.S. SBBD, 2006

[23] S. Agrawal, E. Chu, V. Narasayya. “Automatic physical design tuning: workload as a sequence”, Proceedings of the 2006 ACM SIGMOD international conference on Management of data, June 27-29, 2006

[24] N. Bruno, S. Chaudhuri. “An Online Approach to Physical Design Tuning”. Proceedings of the 2007 ICDE Conference

[25] Z. Chen, V. Narasayya. “Efficient computation of multiple group by queries”. In SIGMOD, 2005

[26] S. Chaudhuri , V. Narasayya “Self-tuning database systems: a decade of progress”, Proceedings of the 33rd international conference on Very large data bases, September 23-27, 2007, Vienna, Austria

[27] N. Bruno, S. Chaudhuri, A. C, König, V. Narasayya, R. Ramamurthy, M. Syamala. “AutoAdmin Project at Microsoft Research: Lessons Learned”. Bulletin of the Technical Committee on Data Engineering, Vol. 34, №. 4, December 2011, pp. 12-19

[28] N. Bruno, S. Chaudhuri. “Automatic physical database tuning: A relaxation-based approach”. In Proceedings of the ACM International Conference on Management of Data

(SIGMOD), 2005

[29] N Bruno , S Chaudhuri. “To tune or not to tune?: a lightweight physical design alerter”, Proceedings of the 32nd international conference on Very large data bases, September 12-15, 2006, Seoul, Korea

[30] N. Bruno, S. Chaudhuri, R. Ramamurthy, “Power Hints for Query Optimization”, IEEE International Conference on Data Engineering, 2009

[31] K. Lee, A. C. König, V. Narasayya, B. Ding, S. Chaudhuri, B. Ellwein, A. Eksarevskiy, M. Kohli, J. Wyant, P. Prakash et al., "Operator and query progress estimation in microsoft sql server live query statistics", Proceedings of the 2016 International Conference on Management of Data, pp. 1753-1764, 2016

[32] S. Chaudhuri, R. Kaushik, R. Ramamurthy. "When Can We Trust Progress Estimators for SQL Queries?" in SIGMOD '05: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data. New York, NY, USA: ACM Press, 2005, pp. 575-586

[33] C. Mishra, N. Koudas. “A lightweight online framework for query progress indicators”. In Proceedings of the 23rd ICDE Conference, 2007

[34] A Dziedzic, J. Wang, S. Das, B. Ding, V. Narasayya, M. Syamala. “Columnstore and B+ tree - Are Hybrid Physical Designs Important?” Proceedings of the 2018 International Conference on Management of Data, June 10-15, 2018, Houston, TX, USA

[35] J. Arulraj , A. Pavlo , P. Menon, “Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads”, Proceedings of the 2016 International Conference on Management of Data, June 26-July 01, 2016, San Francisco, CA, USA

[36] V. Narasayya, M. Syamala. “Workload driven index defragmentation”. In Proceedings of the IEEE International Conference on Data Engineering, pages 497–508. IEEE Computer Society, 2010