Improving Query Processing Performance in Database Management Systems

Suren Dilanka Gamage

149210D

Faculty of Information Technology

University of Moratuwa

May 2018

Improving Query Processing Performance in Database Management Systems

Suren Dilanka Gamage

Index no: 149210D

Dissertation submitted to the Faculty of Information Technology, University of Moratuwa, Sri Lanka for the partial fulfillment of the requirements of the Master of Science in Information Technology.

May 2018

Declaration

I declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institution of tertiary education. Information derived from the published or unpublished work of others has been acknowledged in the text and a list of references is given.

……………………………….....   ……………………………
Name of Student (s)   Signature of Student (s)

Date:

Supervised by

……………………………………   …………………………...
Name of Supervisor(s)   Signature of Supervisor(s)

Date:

Dedication

This thesis is dedicated to my wife, Mrs. U. Kumarapeli for her endless love, encouragement, and support.

Acknowledgments

First and foremost I would like to offer my sincere gratitude to my research supervisor, lecturer Mr.Chaman Wijesiriwardana, for his guidance, supervision, encouragement, and support throughout this study.

I would not know what research is and how to do research if the lecture series of thesis writing and research methodologies are not offered. So I would like to offer my sincere gratitude to the Prof. Asoka S. Karunananda for feeding the knowledge and guidance for doing researches in the proper way.

I would also like to thank my lecturer S. Premaratne and all the lecturers of Faculty of Information Technology – University of Moratuwa, for their guidance and encouragement to get maximum use of knowledge and capabilities.

I am grateful to the management and Softlogic Holdings PLC for their kind support and understanding during this work.

Finally, I would like to extend my deepest gratitude to my parents and family, for their continuous support given in every possible way to make this project successful.

Abstract

Improving Query Processing Performance in Database Management Systems has been a research challenge. This is the most important and is a real problem, this happens to be very crucial in large organizations with heterogeneous data, online system, billing systems and so on. Among other issues in the query optimization problem, faced by everyday query optimizers, get more and more complex with the server increasing complexity of user queries. During the last decade, database management systems have become important information processing system supporting business activities of geographically decentralized organizations.
The Performance monitoring has been evaluated and used by various tools. Most DBA's agreed that these tools are valuable. Our research also tried to identify how performance problems could be reduced and which methods were used in practice. Besides hardware upgrades, the following areas in tuning are known to have major impacts.
The main aim of this thesis is to produce flexible database monitoring tool and query optimization techniques that is capable of get basic idea of database server, database log, missing indexes, graphical user interface of currently running queries, optimizing large queries in a complex database. Among other issues in a database, such as deadlock, expensive query, primary key missing places, badly design quarries can be simply identified.
This database monitoring tool and proposed new optimization techniques will more helpful to identify database performance issues and provide better solutions. During the evaluation, it was shown that system was successful more than 70%.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Prolegomena

This chapter presents the background and motivation of the research, hypothesis, objectives, problem statement, our database performance improvements approach and the structure of the rest of the thesis. Here we describe some of the key problem areas currently exist in the world. Especially, database performance problems will be identified and solutions will be proposed to address those problems.

1.2 Background and Motivation

Although the Database Management System has become a de-facto standard, its main strengths have to be found in its ease-of-use and querying capabilities, rather than its efficiency in terms of hardware and system overhead. With the constantly growing amount of data being accumulated and processed by companies' information systems, database performance issues become more likely. In the meantime, user requirements and expectations are constantly arising, and delay in response time could considerably affect the company's operations. Database performance tuning, or database performance optimization, is the activity of making a database system run faster. SQL optimization attempts to optimize the SQL queries at the application Level, and typically offers the biggest potential for database performance optimization.

The query optimizer is widely considered to be the most important part of a database system. The main aim of the optimizer is to take a user query and provide a detailed plan called a Query Execution Plan (QEP), then indicate to the executer exactly how the query should be executed. The problem that the optimizer faces is, for a given user query, a large amount of different equivalent QEPs exists, and each of them has a corresponding execution cost.

1.3 Problem statement

In the modern era, digital data are considered as the most valuable asset of an organization, and the organizations assign more significance to it than the software and hardware assets. Database systems are computer-based record keeping systems, which have been developed to store data for efficient retrieval and processing. Since data is produced and shared every day, data volumes could be large enough for the database performance to become an issue. In order to maintain database performance, identification, and diagnosis of the root cause that may cause delayed queries is done. Poor query design can be one of the major causes of delayed queries. There are various methods available to deal with the performance issues. Database administrator decides the method or a combination of methods that work best.

1.4 Hypothesis

We hypothesize that simple and cost-effective Database monitoring app and proposed database performance techniques would address problems in database performance issues, can be developed using Microsoft Visual Studio, SQL, and best practices.

1.5 Objectives

(i) To study the current database performance issues related to SQL server applications.

(ii) To critically review the available methods for query performance analysis and Fine - tuning large complex SQL queries.

(iii) To do an in-depth study of database behaviors with a large amount of data.

(iv) To develop a new Database monitoring application.

(v) To provide a solution to database performance issues.

(vi) To allow third parties to use the system and techniques.

(vii) To evaluate the performance of the system.

1.6 Structure of the Thesis.

The rest of the thesis is organized as follows. Chapter 2 critically reviews the literature on Improving Query Processing Performance in Database Management Systems and identify the research problems. Chapter 3 includes the technology for Improving Query Processing Performance in Database Management Systems. Chapter 4 presents our new approach to use Improving Query Processing Performance in Database Management Systems. Chapter 5 and Chapter 6 describe the design and implementation respectively. Chapter 7 is an evaluation of the new method. Chapter 8 concludes the research with a note on further work.

1.7 Summary.

This chapter gave an overall picture of the entire project presented in this thesis. Further, we described the background/motivation, problem definition, hypothesis, objectives, and a brief overview of the solution. Next chapter presents a critical review of the literature on improving the query processing performance in database management systems.

Chapter 2

Developments and Challenges in Improving Query Processing Performance in Database Management Systems.

2.1 Introduction

Chapter 1 gave a comprehensive description of the overall project described in this thesis. This chapter provides a critical review of the literature in relation to Improve Query Processing Performance in Database Management Systems developments and its challenges.

For this purpose, the review of the past researchers has been presented under five major sections, such as Database optimization techniques, tools used to find database performance, hardware configurations, proper database designing techniques and future challenges and find out unsolved problems.

2.2 Early developments

With the rapid development of science and technology, information systems have become a necessity in people's day to day life. Optimization of database systems plays an important role and runs throughout the entire lifecycle of the database application, however, performance for most database systems is only assessed after the completion of the entire system at an early stage. Mostly, performance assessment for some database systems is performed after system deployment [1]. The earlier optimization work starts, the less cost. Database system performance tuning should be taken into consideration in design stage [2]. Andrew proposed a novel approach to database performance optimization meeting the requirements of the query process [3].

A distributed relational database is a distributed database consisting of multiple physical locations or sites and a number of relations. Relations may be replicated and/or fragmented at different sites in the system. The placement of data in the system is determined by factors such as local ownership and availability concerns. A distributed database management system should not be confused with a parallel Database management system, which is a system where the distribution of data is determined entirely by performance considerations. When every site in the system runs the same DBMS software, the system is called ‘homogenous’. Otherwise, the system is called a ’heterogeneous’ system or a multi-database system [4].

To achieve database tuning, it is important to understand the causes of the problems and find the current bottlenecks as there can be various possible factors affecting the database performance. A first category targets ‘hardware’ factors which include memory, processor, and disk and network performance. Other category includes the database related factors such as the database design, indexing, partitioning or locking. And sometimes application level problems can also be the cause of performance degradation. [2]

D. Kossmann et al[5] presented four different architectures based on classic multi-tier database application architecture which includes partitioning, replication, distributed control and caching architecture. It is clear that alternative providers have different business models and target different kinds of applications: Google seems to be more interested in small applications with light workload whereas Azure is currently the most affordable service for medium to large services. Most of recent cloud service providers are utilizing hybrid architecture which is capable of satisfying their actual service requirements. In this section, we mainly discuss big data architecture from four key aspects such as; big data service models, distributed file system, non-structural and semi-structured data storage and data virtualization platform.

2.3 Modern trends in improving Query Processing

Performance tuning of database management system means enhancing the performance of the database, i.e., minimizing the response time at a very optimum cost. As query response time is the number one metrics when it comes to database performance, query optimization is one of the important aspects of performance tuning. [15]

Column-oriented database systems (column-stores) have attracted a lot of attention in the past few years. Column-stores, in a nutshell, store each database table column separately, with attribute values belonging to the same column stored contiguously, compressed, and densely packed, as opposed to traditional database systems that store entire records (rows) one after the other. Reading a subset of a table's columns becomes faster, at the potential expense of excessive disk-head seeking from column to column for scattered reads or updates. After several dozens of research papers and at least a dozen of new column-store startups, several questions remained. Is there a new breed of systems or simply old wine in new bottles? How easily can a major row-based system achieve column-store performance? Is column-stores the answer to effortlessly support large-scale data-intensive applications? What are the new, exciting system research problems to tackle? What are the new applications that can be potentially enabled by column-stores? In this tutorial, we present an overview of column-oriented database system technology and address these and other related questions.

2.4.1 Future challenges of Improving Query Processing

Data processing is a common part of the processes inside every organization. Critical challenges of these days come with well-known characters defined mostly for big data – velocity, variety, and volume. Even new technologies exist, traditional data sources and processes require a variety of different approaches. Current research and development in the field of data processing accommodates knowledge from different areas including algorithms, hardware, software, engineering, and social issues. Applications usually combine high-performance computers for computation, high-performance databases and cloud servers for data storage and management, and desktop computers for human-computer interaction source for processing often comes from models or observations based on different scientific, engineering, social, and cyber applications.

Massive sets of data in pet bytes (1015) or terabytes (1012) are available for analytical and transactional processing. Main application areas are medicine, large sensor networks, social networks, and other industrial-based sources of data. The common factor is the existence of connections between data which on the other hand leads to increased complexity of data sets. In our paper, we will define some of our observations and selected experimental results to describe basic challenges of data processing. We are dealing with three different approaches such as relational, semantic, and graph based. All of these require accommodation of different techniques.

2.4.2 Big Data Management System

According to a recent survey by Gartner in 2010g, 47% of survey respondents ranked data growth in their top three challenges, followed by system performance and scalability at 37%, and network congestion and connectivity architecture at 36%. Many researchers have suggested that commercial Data Base Management Systems (DBMSs) are not suitable for processing extremely big data. Classic architecture's potential bottleneck is the database server while facing peak workloads. One database server has the restriction of scalability and cost [22], which are two important goals of big data processing in order to adapt various large data processing models.

2.4.3 Big Data Service Model

As we all know, cloud computing is a kind of information and communication [29] technology, which delivers valuable resources to people as a service, such as Software as a Service (SaaS), Infrastructure as a Service (IaaS) and Platform as a Service (PaaS) [24]. There are several leading Information Technology (IT) solution providers, who offer these services to the customers. Now, as the concept of the big data came up, the cloud computing service model is gradually transferring into big data service models, which are DaaS (Database as a Service), AaaS (Analysis as a Service) and BDaaS (Big data as a Service). The detailed descriptions are as follows: Database as a Service means that database services are available applications deployed in any execution [28] environment, including on a PaaS. But in the big data context, these would optimally be scale-out architectures such as No SQL data stress and in-memory databases.

Analysis as a Service would be more familiar with interacting with an analytics platform on a higher abstraction level. They would typically execute scripts and queries that data scientists or programmers developed for them.

Big data as a Service coupled with Big Data platforms are for users that need to customize or create new big data stacks, however, readily available solutions do not yet exist. Users must first acquire the necessary cloud computing infrastructure, and manually install the big data processing software. For complex distributed services, this can be a daunting challenge.

2.4.4 Non-structural and Semi-structured Data Storage

With the success of the Web 2.0, most IT companies increasingly need to store and analyze the ever-growing data, such as search logs, crawled web content and click streams collected from a variety of web services, which are usually in the range of petabytes. However, web datasets are usually non-relational or less structured and processing such semi-structured data sets at scale poses another challenge. Moreover, simple distributed file systems mentioned above cannot satisfy service providers like Google, Yahoo!, Microsoft and Amazon. All providers have their purpose to serve potential users and own their relevant state-of-the-art of big data management systems in the cloud environment. Big table [7] is a distributed storage system of Google for managing structured data that is designed to a scale of a very large size (petabytes of data) across thousands of commodity servers. Big table does not support a full relational data model. However, it provides clients with a simple data model that supports dynamic control over data layout and format. PNUTS [8] is a massive scale hosted database system designed to support Yahoo! web applications.

2.4.5 Data Virtualization Platform

Data virtualization describes the process of abstracting disparate systems. It can be described as conceptual building of abstract layers of resources. In short, big data and cloud computing refer to a convergence of technologies and trends that are making IT infrastructures and applications more dynamic, more modular and more consumable. Currently, the technology of constructing virtualization platform is just in the primary phase, which mainly depends on the cloud data center integration technology.

2.4.6 Distributed Applications

In this age of data explosion, parallel processing is essential to perform a massive volume of data in a timely manner. In contrast, the use of distributed techniques and algorithms is the key to achieve better scalability and performance in processing big data. At present, there are a lot of popular parallel and distributed processing models, including MPI, General Purpose GPU (GPGPU), Map Reduce and Map Reduce-like. We will focus on the last two processing models.

2.4.7 Map Reduce

Map Reduce proposed by Google, is a very popular big data processing model that has rapidly been studied and applied by both industry and academia [9]. Map Reduce has two major advantages: it hides details related to the data storage, distribution, replication, load balancing and so on. Furthermore, it is so simple that programmers only specify two functions, such as ‘map function’ and ‘reduce function’. We divide existing Map-Reduce applications into three categories as, partitioning sub-space, decomposing sub-processes and approximate overlapping calculations. While Map Reduce is referred to as a new approach of processing big data in cloud computing environments, it is also criticized as a “major step backwards” compared with DBMS. As the debate continues, the final result shows that neither of them is good at what the other does well, and the two technologies are complementary [19]. Recently, some DBMS vendors have integrated Map Reduce front-ends into their systems including Aster, HadoopDB[14], Greenplum[15]. Most of those are still databases, which simply provide a MapReduce front-end to a DBMS. HadoopDB is a hybrid system which efficiently takes the best features from the scalability of Map Reduce and the performance of DBMS. Lately, J. Dittrich et al. proposed a new type of system named Hadoop++ which indicates that HadoopDB also has severe drawbacks, including forcing user to use DBMS, changing the interface to SQL and so on.

2.4.8 Map Reduce Optimization

Previous works have shown that Map-Reduce systems are inefficient in utilizing computing resources. In this section, we present details of approaches for improving the performance of processing big data with Map Reduce.

2.4.9 Data Transfer Bottlenecks

It is a big challenge that cloud users must consider how to minimize the cost of data transmission. Consequently, researchers have begun to propose variety of approaches. Map-Reduce-Merge[17] is a new model that adds a Merge phase after Reduce phase that combines two reduce outputs from two different MapReduce jobs into one, which can efficiently merge data that is already partitioned and sorted (or hashed) by Map and Reduce modules. Map-Join-Reduce [18] is a system that extends and improves MapReduce runtime framework by adding Join stage before Reduce stage to perform complex data analysis [19] tasks on large clusters. The authors presented a new data processing strategy which runs filtering-join aggregation tasks with two consecutive MapReduce jobs. It adopts one-to-many shuffling [20] scheme to avoid frequent check pointing and shuffling of intermediate results. Moreover, different jobs often perform similar work, thus sharing similar work reduces overall amount of data transfer between jobs. MRShare[21] is a sharing framework proposed by T. Nykiel et al. that transforms a batch of queries into a new batch that can be executed more efficiently by Merging jobs into groups and evaluating each group as a single query.

2.4.10 Index Optimization

Many researchers have implemented the traditional and optimized index structures on MapReduce to obtain better performance. In, T. Liu et al. built hybrid spill trees in parallel and implemented a scalable image searching algorithm which can be used efficiently to find near duplicates among over billions of images using MapReduce. However, the tree-based approaches have some problems. They did not scale due to traditional top-down search that overloaded the nodes near the tree root, and failed to provide full decentralization. Whereas Voronoi based index [22] made clusters highly scalable by its loose coupling and shared nothing architecture. Till now, Voronoi based index cannot process multidimensional data. Hence, the index structure which is simple, scalable and we'll be used for distributed processing mode is a best choice for the effective store and processing of the data. Later, Menon et al. presented a novel parallel algorithm for constructing suffix array and BWT of a sequence leveraging the unique features of MapReduce and reduced the end to end runtime from hours to mere minutes. [22] There are also some papers adapting inverted index, which is a simple but practical index structure and appropriate for MapReduce to process big data, such as in[22] etc. We did a research on large-scale spatial data environment and designed a distributed inverted grid index by combining inverted index and spatial grid partition with MapReduce model, which is simple, dynamic, scalable and fits for processing high dimensional spatial data.[23]While most kinds of large data are high dimensional, so in[24], J.Wang et al. designed a new system, epic, in which different types of indexes were built to provide efficient query processing for different applications.

2.4.11 Iterative Optimization

Classic parallel applications are developed using message passing runtimes such as MPI (Message Passing Interface) and PVM (Parallel Virtual Machine), where parallel algorithms are developed using above techniques to utilize the rich set of communication and synchronization constructs offered which are to create diverse communication topologies [29]. In contrast, MapReduce and similar high-level programming models support simple communication topologies and synchronization constructs. MapReduce also is a popular platform in which the data flow takes the form of a directed acyclic graph of operators. However, it requires lots of I/Os and unnecessary computations while solving the problem of iterations with MapReduce.

2.5 Summary

This chapter presented a comprehensive literature review on Improving Query Processing Performance in Database Management Systems and identified the research problem as the inadequate attention to reliability of Improving Query Processing Performance Techniques. We also identified the various Methods to address the above problem. Next chapter will discuss about the technology to be used for our solution.

Chapter 3

Technology Adopted for Improving Query Processing Performance in Database Management Systems

3.1 Introduction

In the previous chapter, various researchers to address the same issues were critically reviewed. Advantages, disadvantages, and features of existing systems and proposed systems were analyzed and listed.

In this chapter, technologies regarding database performance system will be described. Also, technologies and methods and tools used in development, testing and implementation will be discussed.

3.2 Technologies Available

3.2.1 Database Monitoring

Database monitoring app was developed under Microsoft Visual Studio .net framework 4.0 and language used C#, also background running tested large queries and will pop up all details to the front end.

Database query fine tuning porotype was developed under SQL.it is all are precompiled.

3.2.2 Query Analyzing and Optimization

Microsoft SQL server comes with inbuilt Profiler. So we can use this for analyzing SQL queries in some level. But its need more knowledge to use it.

SQL Sentry plan explore is another tool which I used in this research.it is analyze the query and will give the performance statics, such as log write, read, idle time, index analyze.

Database Tuning Engine Advisor analyzes the query and provides a graphical report and it shows index analysis. But it is a little bit difficult to handle and use.

3.3 Technology Stack

Our Technology stack is C#, SQL database, and SQL Profiler. The overall technology stack can be illustrated as Figure 3.1 – Technology Stack.

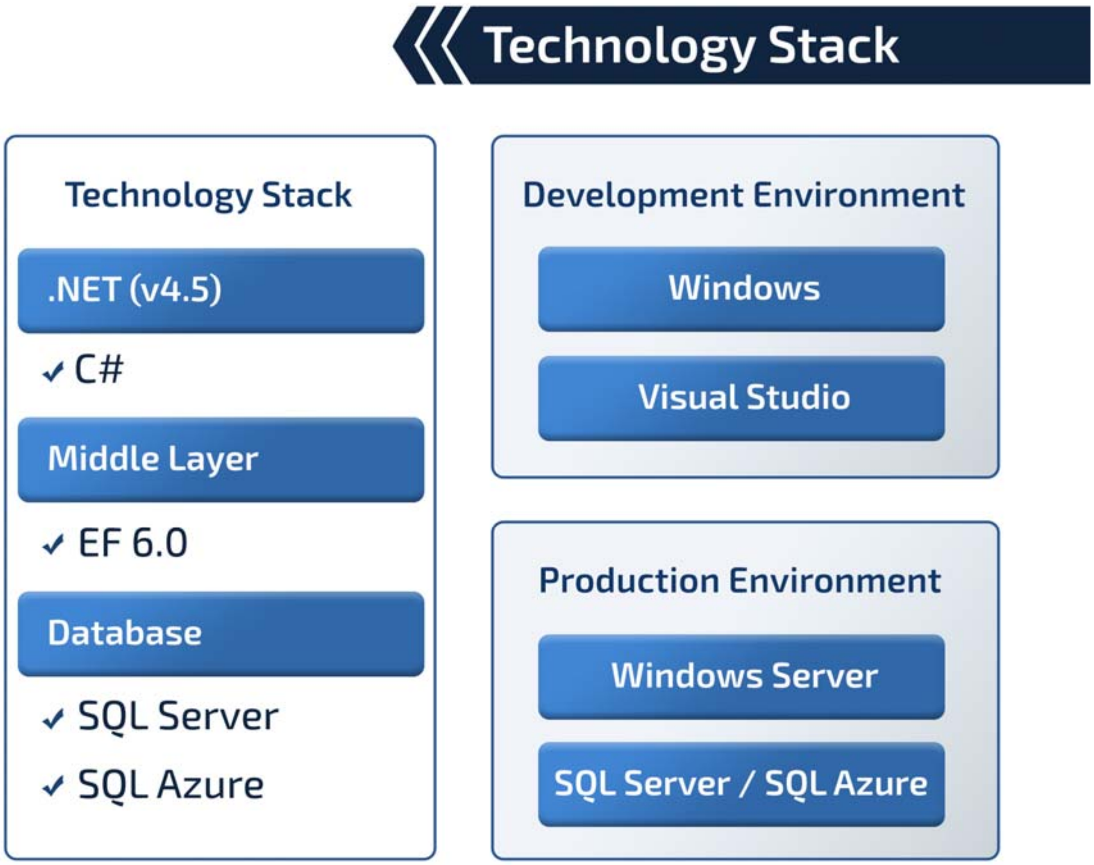


Figure 3.1 – Technology Stack.

3.4 Summary

In this chapter, the technologies used for our Database performance improvement techniques were described. Further, the reasons for selecting relevant technologies were also explained. And some advantages and disadvantages of technologies were briefly discussed.
In next chapter, the approach to implement our Database performance improvement techniques will be described.

Chapter 4

An approach to Improving Query Processing Performance in Database Management Systems

4.1 Introduction

Having defined the problem in Chapter 2 we presented technology required for the proposed solution in chapter 3. This chapter presents our novel approach to use C# and SQL technology to address our research problem. The approach is described under the headings of hypothesis, users, input to the system, output of the system, process to convert the input to the output and overall features of the system.
One of the first tasks in database tuning is to understand the causes of the problem and find the current bottlenecks. This is already challenging in itself, given the very diverse factor affecting the overall database performance. A first category can be grouped under "Hardware" and includes the processor, memory, and disk and network performance. Other factors are more directly related to database systems itself, such as the database design, indexing, partitioning or locking. Finally, the problem can also arise at the application level.

4.2 Hypothesis

We hypothesize that simple and cost-effective Database monitoring app and proposed database performance techniques would address problems in database performance issues, can be developed using Microsoft Visual Studio, SQL, and best practices.

4.3 Users

The number of users can be benefited by the Database performance monitoring system and proposed performance improvement prototype. More importantly, Database administrator, Database developers, System users, Customers and organizational management can be directly benefited by the system.
University students who interested in Database systems and data analyzing can use the system for learning purposes.
Software developing company who interesting product in Database systems and data analyzing can use the system for business purposes.

4.4 Input

The system can accept any version of SQL server database from multiple servers which has location any place and any remote servers. The input could be as,
Any user accessing the system data should go through an authentication process

- SQL Login Username, password (Authentication details)
- Complex SQL queries
- The large volume of a data table

4.5 Output

4.5.1 Database Monitoring application output

- Database installation information
- Database version information
- The graphical graph with displaying all running query count(DB read, write, log read)
- Database allocated memory
- Performance improvement suggestion
- Currently, running process id's in SQL server
- Disk space analysis information
- Locked objects
- Waiting for task analysis
- Missing index details

4.5.2 Database Performance improvement prototype (suggested techniques)

- Reduce execution time
- Rich response time
- High throughput
- Faster processing of the query
- Lesser cost per query
- The high performance of the system
- lesser stress on the database
- Efficient usage of the database engine
- Lesser memory is consumed

4.6 Process

We have divided the entire process into two major parts which are the development of databases monitoring app and proposed the new prototype for optimizing SQL queries.

4.6.2 Developed database monitoring app

A Database monitoring app consists of all the database retrieving, analysis, and suggestion and provides graph and statistics of the database.

Graphs would be used for identifying the performance of current database and database server.

4.6.3 Queries Optimization techniques.

4.6.3.1 Introducing new Queries Optimization techniques.

Here we introduce our own two techniques to optimize complex queries which were found during this research. Also, it was well tested and evaluated by using various types of databases.

Technique 01: Avoid IN Operator in WHERE clause

Solution: Create the temp table and insert needed data and JOIN to the main query

If problematic query consists IN Operator then you must remove that and apply new method as below mentioned way.

- Remove IN operator and create a #temp table instead of that and then insert necessary data to that table
- After that create an index to the temp table.
- Then temp table will join with the base table.

This technique may be questioned for a first time user, but in deeply I have verified this new technique and tested a large number of times, spending a longer period.

This technique would not be found in any research paper or any web reference since last April.

Technique 02: Avoid temp tables as much as you can, but if you need a temp table, create it explicitly using Create Table #temp

Solution: Create #temp table

4.6.3 Queries Optimization techniques (Prototype)

Writing efficient queries in SQL Server is more an exercise in writing elegant relational queries than in knowing specific tricks and syntax tips. Generally, a well-written, relationally correct query written against a well-designed relationally correct database model that uses the correct indexes produces a system that performs fairly well and that is scalable. The following guidelines may help you create efficient queries:

- Know the performance and scalability characteristics of queries.
- Write correctly formed queries.
- Return only the rows and columns needed.
- Avoid expensive operators such as NOT LIKE.
- Avoid explicit or implicit functions in WHERE clauses.
- Use locking and isolation level hints to minimize locking.
- Use stored procedures or parameterized queries.
- Minimize cursor use.
- Avoid long actions in triggers.

- Use temporary tables and table variables appropriately.
- Limit query and index hints use.
- Fully qualify database objects.
- Avoid operators such as IN.
- Know the Performance and Scalability Characteristics of Queries

The best way to achieve performance and scalability is to know the characteristics of your queries. Although it is not realistic to monitor every query, you should measure and understand your most commonly used queries. Do not wait until you have a problem to perform this exercise. Measure the performance of your application throughout the life cycle of your application.

Good performance and scalability also require the cooperation of both developers and database administrators. The process depends on both query development and index development. These areas of development typically are found in two different job roles. Each organization has to find a process that allows developers and database administrators to cooperate and to exchange information with each other. Some organizations require developers to write appropriate indexes for each query and to submit an execution plan to the database architect. The architect is responsible for evaluating the system as a whole, for removing redundancies, for finding efficiencies of scale, and for acting as the liaison between the developer and the database administrator The database administrator can then get information on what indexes might be needed and how queries might be used. The database administrator can then implement optimal indexes.
In addition, the database administrator should regularly monitor the SQL query that consumes the most resources and submits that information to the architect and developers. This allows the development team to stay ahead of performance issues.

4.6.3.1 Write Correctly Formed Queries

Ensure that your queries are correctly formed. Ensure that your joins are correct, that all parts of the keys are included in the ON clause, and that there is a predicate for all queries. Pay extra attention to ensure that no cross products result from missing ON or WHERE clauses for joined tables. Cross products are also known as Cartesian products.

Do not automatically add a DISTINCT clause to SELECT statements. There is no need to include a DISTINCT clause by default. If you find that you need it because duplicate data is returned, the duplicate data may be the result of an incorrect data model or an incorrect join. For example, a join of a table with a composite primary key against a table with a foreign key that is referencing only part of the primary key results in duplicate values. You should investigate queries that return redundant data for these problems.

4.6.3.2 Return Only the Rows and Columns Needed

One of the most common performance and scalability problems are queries that return too many columns or too many rows. One query in particular that returns too many columns are the often-abused SELECT * FROM construct. Columns in the SELECT clause are also considered by the optimizer when it identifies indexes for execution plans. Using a SELECT query not only returns unnecessary data, but it also can force clustered index scans for the query plan, regardless of the WHERE clause restrictions. This happens because the cost of going back to the clustered index to return the remaining data from the row after using a non-clustered index to limit the result set is actually more resource-intensive than scanning the clustered index.

The query shown in Figure 4.6.1 shows the difference in query cost for a SELECT * compared to selecting a column. The first query uses a clustered index scan to resolve the query because it has to retrieve all the data from the clustered index, even though there is an index on the OrderDate column. The second query uses the OrderDate index to perform an index seek operation. Because the query returns only the OrderID column, and because the OrderID column is the clustering key, the query is resolved by using only that index. This is much more efficient; the query cost relative to the batch is 33.61 percent rather than 66.39 percent. These numbers may be different on your computers.

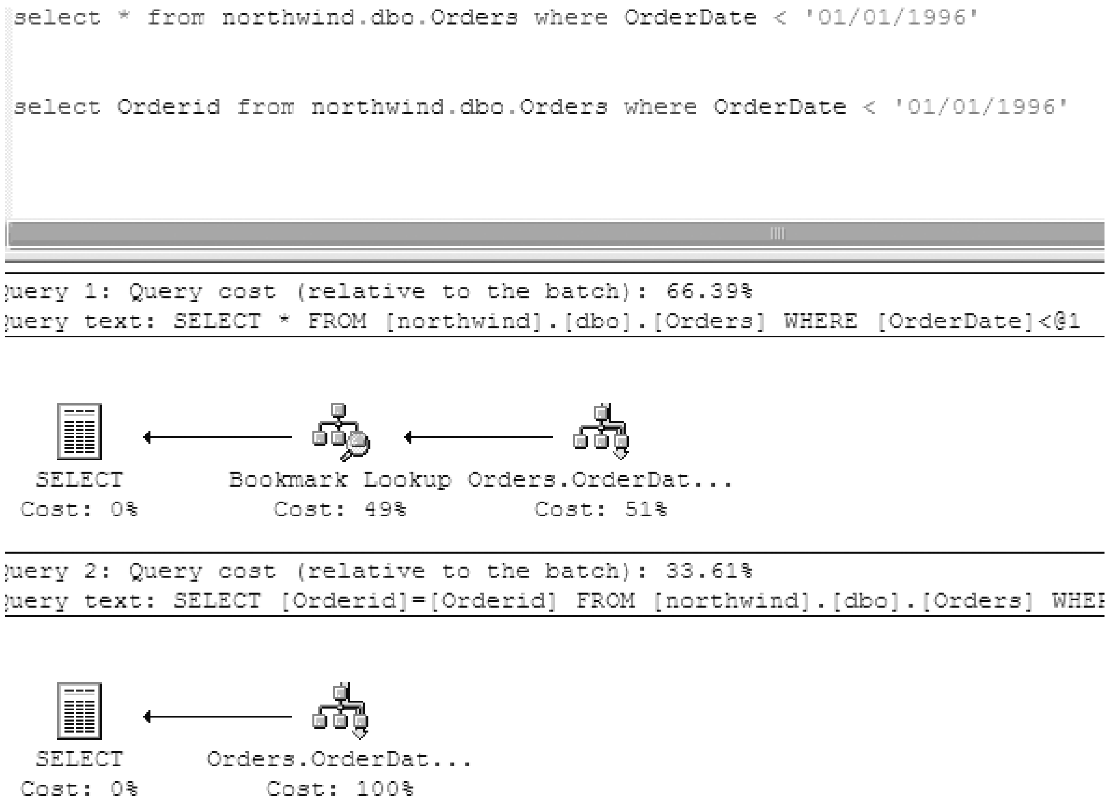


Figure 4.6.1 - Query cost

4.6.3.3 Use Indexed Views for De-normalization

When you have joins across multiple tables that do not change frequently, such as domain or lookup tables, you can define an indexed view for better performance. An indexed view is a view that is physically stored like a table. The indexed view is updated by SQL Server when any of the tables that the indexed view is based on are updated. This has the added benefit of pulling I/O away from the main tables and indexes.

4.6.3.4 Partition Tables Vertically and Horizontally

You can use vertical table partitioning to move infrequently used columns into another table. Moving the infrequently used columns makes the main table narrower and allows more rows to fit on a page.
Horizontal table partitioning is a bit more complicated. But when tables that use horizontal table partitioning are designed correctly, you may obtain huge scalability gains. One of the most common scenarios for horizontal table partitioning is to support history or archive

databases where partitions can be easily delineated by date. A simple method that you can use to view the data is to use partitioned views in conjunction with check constraints.

Data-dependent routing is even more effective for very large systems. With this approach, you use tables to hold partition information. Access is then routed to the appropriate partition directly so that the overhead of the partitioned view is avoided.

If you use a partitioned view, make sure that the execution plan shows that only the relevant partitions are being accessed. Figure 4.6.2 shows an execution plan over a partitioned view on three orders tables that have been horizontally partitioned by the OrderDate column. There is one table per year for 1996, 1997, and 1998. Each table has a PartitionID column that has a check constraint. There is also a partition table that includes a PartitionID and the year for that partition. The query then uses the partition table to get the appropriate PartitionID for each year and to access only the appropriate partition.

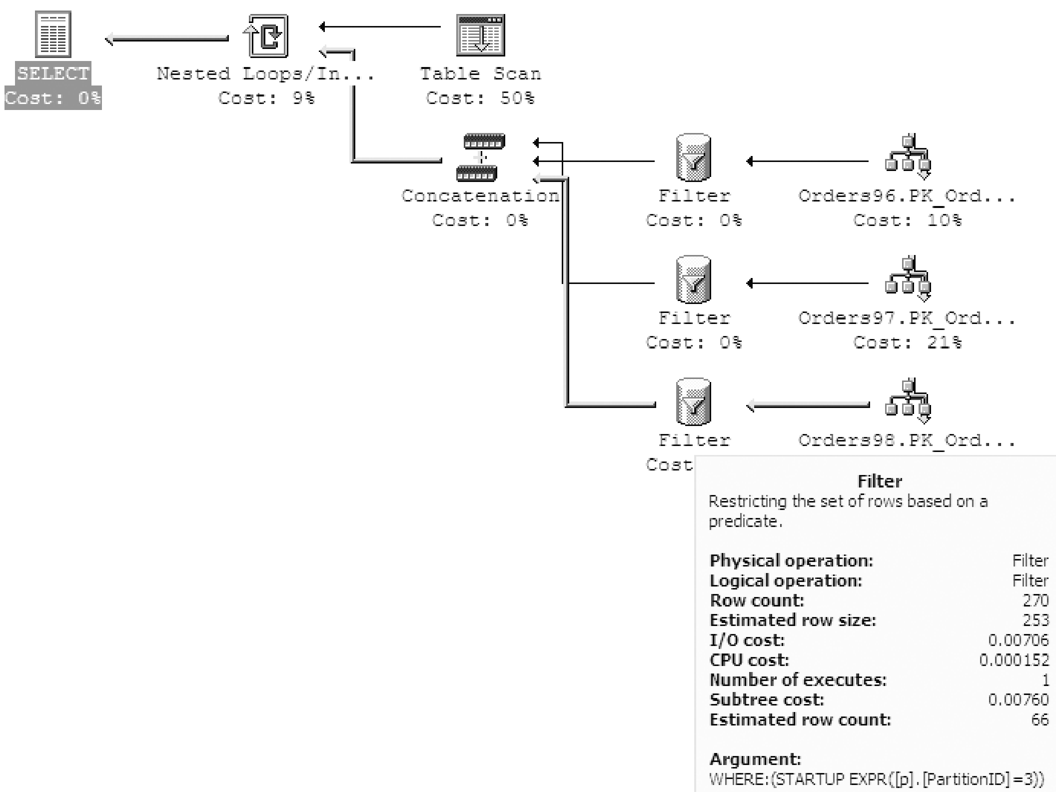


Figure 4.6.2 - Execution plan

4.6.3.5 Avoid Explicit or Implicit Functions in WHERE Clauses

The optimizer cannot always select an index by using columns in a WHERE clause that are inside functions. Columns in a WHERE clause is seen as an expression rather than a column. Therefore, the columns are not used in the execution plan optimization. A common problem is date functions around date time columns. If you have a date time column in a WHERE clause, and you need to convert it or use a data function, try to push the function to the literal expression.

The following query with a function on the date time column causes a table scan in the NorthWind database, even though there is an index on the OrderDate column:

SELECT OrderID FROM NorthWind.dbo.Orders WHERE DATEADD (day, 15, OrderDate) = '07/23/1996'

However, by moving the function to the other side of the WHERE equation, an index can be used on the datetime column. This is shown in the following example:

SELECT OrderID FROM NorthWind.dbo.Orders WHERE OrderDate = DATEADD (day, -15, '07/23/1996')

4.6.3.6 SQL Tuning

SQL tuning is believed to have the largest impact on performance (more than 50%).SQL is a declarative language which only requires the user to specify what data is wanted. There might be hundreds or thousands of different ways to correctly process the query. Hence, it’s very hard for the DBMS query optimizer to decide which access path should be used. The best execution plan chosen by the query optimizer is called the query execution plan (QEP).

4.6.3.6.1 List of methods for SQL query tuning

4.6.3.6.1.1 Gather statistics

It is for Oracle DB. It relies on up to date statistics to generate the best execution plan. Updated statistics help the optimizer to select perfect execution plan for a query.

It can be resource consuming. It must plan accordingly before executing. [3]

4.6.3.6.1.2 Index Management

Indexes are optional structures associated with tables and clusters that allow SQL statements to execute more quickly against these tables. Index created columns help queries to select using index instead of doing the full table scan, which is usually expensive.

DML statements can be slow if there is a lot of indexes on the table. [5]

4.6.3.6.1.3 Table Reorganization

It is used to improve the performance of queries or DML operations performed against these tables. All data blocks will be moved to be together and prevent fragmentation which can cause slowness.
It is usually time-consuming and needs downtime. [6]

4.6.3.6.1.4 Prediction

It involves estimating the space used of a table, estimating the space use of an index and obtaining Object Growth Trends. It is able to predict the problem before it happened.
Sometimes the predictions are not accurate because the data consumed is not increased in sequence. [5]

4.6.3.6.1.5 Data Mart

A data warehouse is designed for data to be collected directly from the various sources. The database will be grouped according to schemas and departments. It is easy to maintain and improve the database performance.
It is applicable only to the schemas which are less than 100 GB. It is not optimum to use data mart for the bigger database. [5]

4.6.3.6.1.6 Materialized View

It provides access to table data by storing the results of a query in separate schema object. It results in fast synchronization between source and target. Data can be refreshed on preferred method.
Complex queries on Materialized View tables perform badly, especially if there are joins with other tables. [5]

4.6.3.6.1.7 Partition Table

It splits the table into smaller parts that can be accessed, stored and maintained independently of one another. It improves performance when selecting data. It can be used easily for data pruning.
It is hard for the DBAs to do maintenance on a partitioned table if it involves lots of partition in a table. Index creation will be slow if a hash partition is used. [4]

4.6.3.6.1.8 Query Rewriting

It consists of the compilation of an ontological query into an equivalent query against the underlying relational database. It improves the way, data are being selected. By adding hints in the SQL, it sometimes enhances the performance of individual queries.
It can be a troublesome job to change hardcoded queries. Queries that are tested thoroughly could cause slowness.

4.6.3.6.1.9 Monitoring Performance

It is used to determine possible problems, locate the root cause and provide recommendations for correcting them. It is able to identify the root cause of the problem.
Only DBA is able to do monitoring.

4.6.3.6.2.0 Optimization

Query optimization is the process of choosing the most efficient way to execute an SQL statement. It helps queries to run faster. Data retrieval can be improved. Parameter tuning and memory tuning enable the database to perform an optimum level.
It must be done with supervision and wrong parameter set can cause the database to go down or perform badly. Memory leak is also possible.

4.7 Features

In connection with the input, output, users, and process, the over features of the system include the following characteristics.

- Online database server analysis
- Online database analysis
- Easy to use
- High level of accuracy
- User-friendly
- Easy to install
- Easy to maintains.

4.8 Summary

In connection with the input, output, users, and process, the over features of the system include the following characteristics.

Chapter 5

Design and Implementation of Database monitoring app and Database query optimizing the prototype

5.1 Introduction

The previous chapter gave the full picture of the entire solution. This chapter describes the design of the solution for the process presented in the approach. We design the solution as a client-server system with a backend database and administration interface. Separate SQL performance analysis and optimizing techniques designed to capture database performance issues and solve them. Here we describe the top-level architecture of the design by elaborating on the role of each component of the architecture.

To retain its users, any application or website must run fast. For mission-critical environments, a couple of milliseconds delay in getting information might create big problems. As database sizes grow day by day, we need to fetch data as fast as possible and write the data back into the database as fast as possible. To make sure all operations are executing smoothly, we have to tune our database server for performance.

The overall solution has been implemented as Microsoft based application running on SQL server. A C# application is implemented as the main analyze source for the system. The algorithms, hardware, software, pseudo codes and relevant code segments of the implementation are presented in this chapter.

5.2 Design Database Monitoring Application

The architecture of Database monitoring app consists layered architecture.it consists of UI, Logic, Controls, common and security layered.

Appendix A illustrates User interface of the system.

Database monitoring app consists of five main modules, namely, Performance analyzes, and Server analyzes, Database connectivity and Activities of the database.

5.3 Implementation of Database Monitoring Application

The main application is developed on C# which is an object oriented languages with many features. Its efficiency is high since it consumes few of system resources. It is cross-platform and has object oriented features. C# is cross-platform since it is running on LINUX, UNIX and WINDOWS. This was used considering the stability, security and further scalability

5.4 Query Optimization Techniques (Prototype)

5.4.1 Finding the Culprits

Tools used to find culprits: Server Profiler / Tuning advisory / DB Monitor App

As with any other software, we need to understand that Database server (SQL) is a complex computer program. If we have a problem with it, we need to discover why it is not running as we expect.

From SQL Server we need to pull and push data as fast and as accurately as possible. If there are issues, a couple of basic reasons, and the first two things to check, are:

The hardware and installation settings, which may need correcting since SQL Server needs are specific

If we have provided the correct T-SQL code for SQL Server to implement

Even though SQL Server is proprietary software, Microsoft has provided a lot of ways to understand it and use it efficiently.

If the hardware is OK and the installation has been done properly, but the SQL Server is still running slowly, then first we need to find out if there are any software related errors. To check what is happening, we need to observe how different threads are performing. This is achieved by calculating wait statistics of different threads. SQL server uses threads for every user request, and the thread is nothing but another program inside our complex program called SQL Server. It is important to note that this thread is not an operating system thread on which SQL server is installed; it is related to the SQLOS thread, which is a pseudo operating system for the SQL Server.

Wait statistics can be calculated using sys.dm_os_wait_stats Dynamic Management View (DMV), which gives additional information about its current state. There are many scripts online to query this view, but my favorite is Paul Randal’s script because it is easy to understand and has all the important parameters to observe wait statistics: Please refer the Figure 5.1 for wait statistics

```
WITH [Waits] AS
(SELECT
[wait_type],
[wait_time_ms] / 1000.0 AS [WaitS],
([wait_time_ms] - [signal_wait_time_ms]) / 1000.0 AS [ResourceS],
[signal_wait_time_ms] / 1000.0 AS [SignalS],
[waiting_tasks_count] AS [WaitCount],
100.0 * [wait_time_ms] / SUM ([wait_time_ms]) OVER() AS [Percentage],
ROW_NUMBER() OVER(ORDER BY [wait_time_ms] DESC) AS [RowNum]
FROM sys.dm_os_wait_stats
WHERE [wait_type] NOT IN (
```

```
N'BROKER_EVENTHANDLER', N'BROKER_RECEIVE_WAITFOR',
N'BROKER_TASK_STOP', N'BROKER_TO_FLUSH',
N'BROKER_TRANSMITTER', N'CHECKPOINT_QUEUE',
N'CHKPT', N'CLR_AUTO_EVENT',
N'CLR_MANUAL_EVENT', N'CLR_SEMAPHORE',
N'DBMIRROR_DBM_EVENT', N'DBMIRROR_EVENTS_QUEUE',
N'DBMIRROR_WORKER_QUEUE', N'DBMIRRORING_CMD',
N'DIRTY_PAGE_POLL', N'DISPATCHER_QUEUE_SEMAPHORE',
N'EXECSYNC', N'FSAGENT',
N'FT_IFTS_SCHEDULER_IDLE_WAIT', N'FT_IFTSHC_MUTEX',
N'HADR_CLUSAPI_CALL', N'HADR_FILESTREAM_IOMGR_IOCOMPLETION',
N'HADR_LOGCAPTURE_WAIT', N'HADR_NOTIFICATION_DEQUEUE',
N'HADR_TIMER_TASK', N'HADR_WORK_QUEUE',
N'KSOURCE_WAKEUP', N'LAZYWRITER_SLEEP',
N'LOGMGR_QUEUE', N'ONDEMAND_TASK_QUEUE',
N'PWAIT_ALL_COMPONENTS_INITIALIZED',
N'QDS_PERSIST_TASK_MAIN_LOOP_SLEEP',
N'QDS_CLEANUP_STALE_QUERIES_TASK_MAIN_LOOP_SLEEP',
N'REQUEST_FOR_DEADLOCK_SEARCH', N'RESOURCE_QUEUE',
N'SERVER_IDLE_CHECK', N'SLEEP_BPOOL_FLUSH',
N'SLEEP_DBSTARTUP', N'SLEEP_DCOMSTARTUP',
N'SLEEP_MASTERDBREADY', N'SLEEP_MASTERMDREADY',
N'SLEEP_MASTERUPGRADED', N'SLEEP_MSDBSTARTUP',
N'SLEEP_SYSTEMTASK', N'SLEEP_TASK',
N'SLEEP_TEMPDBSTARTUP', N'SNI_HTTP_ACCEPT',
N'SP_SERVER_DIAGNOSTICS_SLEEP', N'SQLTRACE_BUFFER_FLUSH',
N'SQLTRACE_INCREMENTAL_FLUSH_SLEEP',
N'SQLTRACE_WAIT_ENTRIES', N'WAIT_FOR_RESULTS',
N'WAITFOR', N'WAITFOR_TASKSHUTDOWN',
N'WAIT_XTP_HOST_WAIT', N'WAIT_XTP_OFFLINE_CKPT_NEW_LOG',
N'WAIT_XTP_CKPT_CLOSE', N'XE_DISPATCHER_JOIN',
N'XE_DISPATCHER_WAIT', N'XE_TIMER_EVENT')
AND [waiting_tasks_count] > 0
)
SELECT
MAX ([W1].[wait_type]) AS [WaitType],
CAST (MAX ([W1].[WaitS]) AS DECIMAL (16,2)) AS [Wait_S],
CAST (MAX ([W1].[ResourceS]) AS DECIMAL (16,2)) AS [Resource_S],
CAST (MAX ([W1].[SignalS]) AS DECIMAL (16,2)) AS [Signal_S],
MAX ([W1].[WaitCount]) AS [WaitCount],
CAST (MAX ([W1].[Percentage]) AS DECIMAL (5,2)) AS [Percentage],
CAST ((MAX ([W1].[WaitS]) / MAX ([W1].[WaitCount])) AS DECIMAL (16,4)) AS
[AvgWait_S],
```

```
CAST ((MAX ([W1].[ResourceS]) / MAX ([W1].[WaitCount])) AS DECIMAL (16,4)) AS
[AvgRes_S],
CAST ((MAX ([W1].[SignalS]) / MAX ([W1].[WaitCount])) AS DECIMAL (16,4)) AS
[AvgSig_S]
FROM [Waits] AS [W1]
INNER JOIN [Waits] AS [W2]
ON [W2].[RowNum] <= [W1].[RowNum]
GROUP BY [W1].[RowNum]
HAVING SUM ([W2].[Percentage]) - MAX ([W1].[Percentage]) < 95;
GO
```

Figure 5.1 wait statistics

When we execute this script, we need to concentrate on the top rows of the result because they are set first and represent the maximum wait type.

We need to understand wait types so we can make the correct decisions. Let's take an example where we have too much PAGEIOLATCH_XX. This means a thread is waiting for data page reads from the disk into the buffer, which is nothing but a memory block. We must be sure we understand what's going on. This does not necessarily mean a poor I/O subsystem or not enough memory and increasing the I/O subsystem and memory will solve the problem, but only temporarily. To find a permanent solution we need to see why so much data is being read from the disk: What types of SQL commands are causing this? Are we reading too much data instead of reading less data by using filters, such as where clauses? Are too many data reads happening because of table scans or index scans? Can we convert them to index seeks by implementing or modifying existing indexes? Are we writing SQL queries that are misunderstood by SQL Optimizer (another program inside our SQL server program)?
We need to think from different angles and use different test cases to come up with solutions. Each of the above wait types needs a different solution. A database administrator needs to research them thoroughly before taking any action. But most of the time, finding problematic T-SQL queries and tuning them will solve 60 to 70 percent of the problems.

5.4.2 Finding Problematic Queries

As mentioned above, the first thing we can do is to search problematic queries. The following T-SQL code will find the 20 worst performing queries. Please refer Figure 5.2

```
SELECT TOP 20
total_worker_time/execution_count AS Avg_CPU_Time
,Execution_count
,total_elapsed_time/execution_count as AVG_Run_Time
,total_elapsed_time
```

```
,(SELECT
SUBSTRING(text,statement_start_offset/2+1,statement_end_offset
) FROM sys.dm_exec_sql_text(sql_handle)
) AS Query_Text
FROM sys.dm_exec_query_stats
ORDER BY Avg_CPU_Time DESC
```

Figure 5.2 find the 20 worst performing queries

We need to be careful with the results; even though a query can have a maximum average runtime, if it runs only once, the total effect on the server is low compared to a query which has a medium average runtime and runs lots of times in a day.

5.4.3 Fine Tuning Queries

The fine-tuning of a T-SQL query is an important concept. The fundamental thing to understand is how well we can write T-SQL queries and implement indexes so that the SQL optimizer can find an optimized plan to do what we wanted it to do. With every new release of SQL Server, we get a more sophisticated optimizer that will cover our mistakes in writing not optimized SQL queries, and will also fix any bugs related to the previous optimizer. But, no matter how intelligent the optimizer may be, if we can't tell it what we want (by writing proper T-SQL queries), the SQL optimizer won't be able to do its job.
SQL Server uses advanced search and sorting algorithms. If we are good at search and sorting algorithms, then most of the time we can guess why SQL Server is taking the particular action.

The best book for learning more and understanding such algorithms is the art of the computer by Donald Knuth.

When we examine queries that need to be fine-tuned, we need to use the execution plan of those queries so that we can find out how SQL server is interpreting them.
I can't cover all the aspects of the execution plan here, but on a basic level, I can explain the things we need to consider.

First, we need to find out which operators take most of the query cost.
If the operator is taking a lot of costs, we need to learn the reason why. Most of the time, scans will take up more cost than seeks. We need to examine why a particular scan (table scan or index scan) is happening instead of an index seek. We can solve this problem by implementing proper indexes on table columns, but as with any complex program, there is no fixed solution. For example, if the table is small then scans are faster than seeks.

There are approximately 78 operators, which represent the various actions and decisions of the SQL Server execution plan. We need to study them in-depth by consulting the Microsoft documentation so that we can understand them better and take proper action.

5.4.4 Execution Plan Re-Use

Even if we implement proper indexes on tables and write good T-SQL code, if the execution plan is not reused, we will have performance issues. After fine-tuning the queries, we need to make sure that the execution plan may be re-used when necessary. Most of the CPU time will be spent on calculating execution plan that can be eliminated if we re-use the plan.

We can use the query below to find out how many times execution plan is re-used, where `usecounts` represents how many times the plan is re-used. Please refer Figure 5.3

```
SELECT [ecp].[refcounts]
, [ecp].[usecounts]
, [ecp].[objtype]
, DB_NAME([est].[dbid]) AS [db_name]
, [est].[objectid]
, [est].[text] as [query_ext]
, [eqp].[query_plan]
FROM sys.dm_exec_cached_plans ecp
CROSS APPLY sys.dm_exec_sql_text ( ecp.plan_handle ) est
CROSS APPLY sys.dm_exec_query_plan ( ecp.plan_handle ) eqp
```

Figure 5.3 How many times execution plan is re-used

The best way to re-use the execution plan is by implementing parameterized stored procedures. When we are not in a position to implement stored procedures, we can use `sp_executesql`, which can be used instead to execute T-SQL statements when the only change to the SQL statements are parameter values. SQL Server most likely will reuse the execution plan that is generated in the first execution.

Again, as with any complex computer program, there is no fixed solution. Sometimes it is better to compile the plan again.

Let’s examine following two example queries:

- `select name from table where name = 'Sri';`
- `select name from table where name = 'pal';`

Let us assume we have a non-clustered index on the `name` column and half of the table has value `Sri` and few rows have `the pal` in the `name` column. For the first query, SQL Server will use the table scan because half of the table has the same values. But for the second query, it is better to use the index scan because only a few rows have `pal` value.

Even though queries are similar, the same execution plan may not be the good solution. Most of the time it will be a different case, so we need to carefully analyze everything before we decide. If we don't want to re-use the execution plan, we can always use the "recompile" option in stored procedures.

Keep in mind that even after using stored procedures or `sp_executesql`, there are times when the execution plan won't be re-used. They are:

- When indexes used by the query change or are dropped
- When the statistics, structure or schema of a table used by the query changes
- When we use the "recompile" option
- When there are a large number of insertions, updates or deletes
- When we mix DDL and DML within a single query

5.4.5 Removing Unnecessary Indexes

After fine-tuning the queries, we need to check how the indexes are used. Index maintenance requires lots of CPU and I/O. Every time we insert data into a database, SQL Server also needs to update the indexes, so it is better to remove them if they are not used.

SQL server provides us `dm_db_index_usage_stats` DMV to find index statistics. When we run the T-SQL code below, we get usage statistics for different indexes. If we find indexes that are not used at all or used rarely, we can drop them to gain performance.
Please refer figure 5.4

```
SELECT
OBJECT_NAME(IUS.[OBJECT_ID]) AS [OBJECT NAME],
DB_NAME(IUS.database_id) AS [DATABASE NAME],
I.[NAME] AS [INDEX NAME],
USER_SEEKS,
USER_SCANS,
USER_LOOKUPS,
USER_UPDATES
FROM SYS.DM_DB_INDEX_USAGE_STATS AS IUS
```

```
INNER JOIN SYS.INDEXES AS I
ON I.[OBJECT_ID] = IUS.[OBJECT_ID]
AND I.INDEX_ID = IUS.INDEX_ID
```

Figure 5.4 Find unnecessary indexes

5.4.6 There are several considerations when writing a query using the IN operator that can have an effect on performance

IN clauses are generally internally rewritten by most databases to use the OR logical connective. So col IN ('a','b','c') is rewritten to: (COL = 'a') OR (COL = 'b') or (COL = 'c'). The execution plan for both queries will *likely* be equivalent assuming that you have an index on col.

When using either IN or OR with a variable number of arguments, you are causing the database to have to re-parse the query and rebuild an execution plan each time the arguments change.

Building the execution plan for a query can be an expensive step. Most databases cache the execution plans for the queries they run using the EXACT query text as a key. If you execute a similar query but with different argument values in the predicate - you will most likely cause the database to spend a significant amount of time parsing and building execution plans.

This is why Join Temp table or bind variables are strongly recommended as a way to ensure optimal query performance.

Many databases have a limit on the complexity of queries they can execute - one of those limits is the number of logical connectives that can be included in the predicate. In your case, a few dozen values are unlikely to reach the built-in limit of the database, but if you expect to pass hundreds or thousands of value to an IN clause - it can definitely happen. In which case the database will simply cancel the query request.

Queries that include IN and OR in the predicate cannot always be optimally rewritten in a parallel environment. There are various cases where parallel server optimization does not get applied - MSDN has a decent introduction to optimizing queries for parallelism. Generally, though, queries that use the UNION ALL operator are trivially parallelizable in most databases - and are preferred to logical connectives (like OR and IN) when possible.

5.4.7 Avoid Cursors

SQL Server cursors are notoriously bad for performance. In any good development environment, people will talk about cursors as if they were demons to be avoided at all costs. The reason for this is plain and simple; they are the best way to slow down an application. This is because SQL Server, like any good relational database management system (RDBMS), is optimized for set-based operations.

5.5 Implementation of Query Optimization Techniques

Below is my list of the top 17 things I believe developers should do as a matter, of course, to tune performance when coding. These are the low hanging fruit of SQL Server performance – they are easy to do and often have a substantial impact. Doing these won't guarantee lightning fast performance, but it won't be slow either.

1. Create a primary key on each table you create and unless you are really knowledgeable enough to figure out a better plan, make it the clustered index (note that if you set the primary key in Enterprise Manager it will cluster it by default).
2. Create an index on any column that is a foreign key. If you know it will be unique, set the flag to force the index to be unique. I recommend using below diagram table (Figure 5.5) when structuring your indexes.

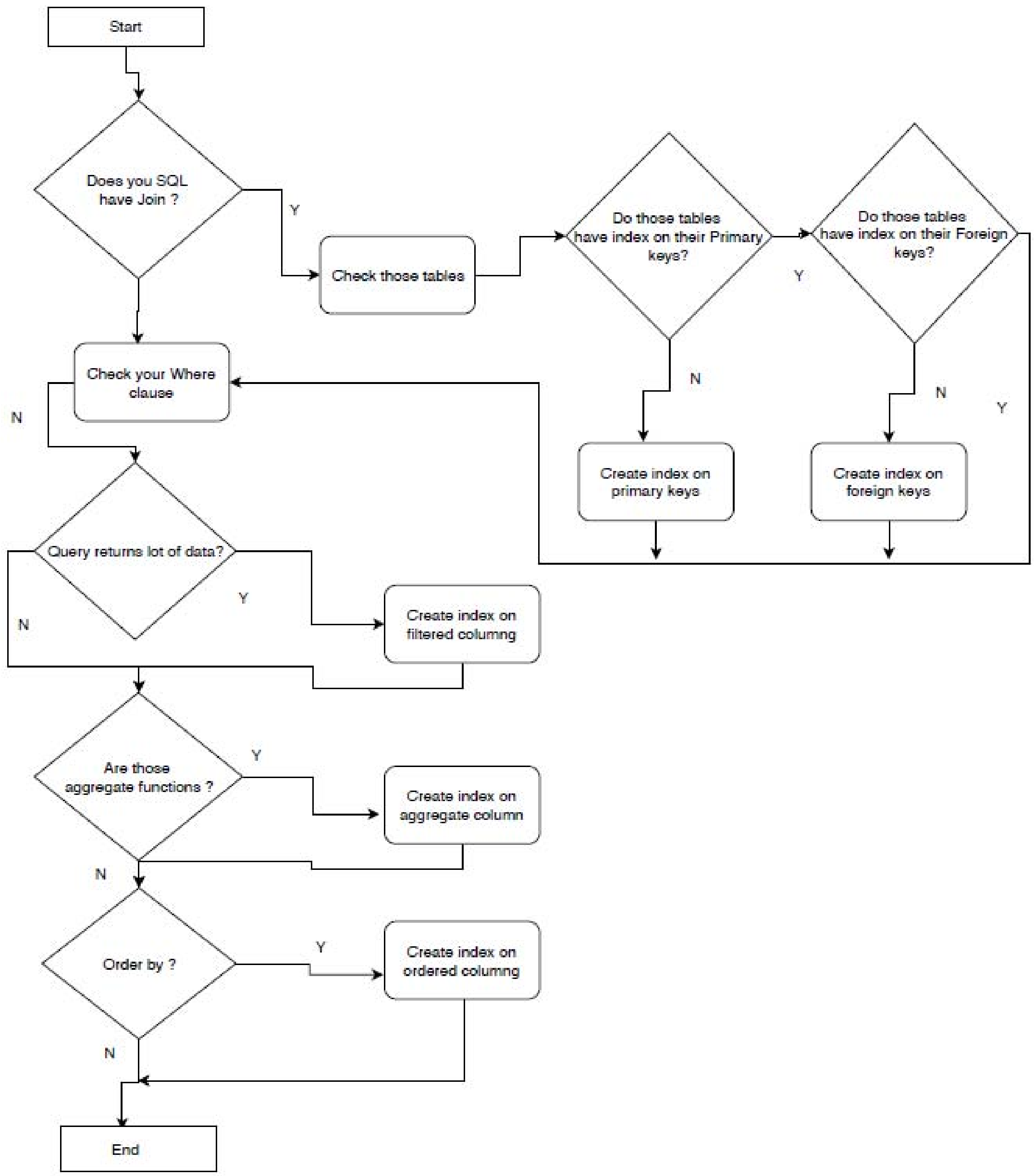


Figure 5.5-Index creation process

3. Don’t index anything else (yet).

4. UNION matchup

   SQL's UNION operator lets you combine records from different sources using the following form. Please refer the figure 5.6

SELECT1 list|*
UNION
SELECT2 list|*

Figure 5.6-Union operators

The important thing to remember with a UNION is that the column order in both SELECT statements must match. The column names don't have to match, but each list must contain the same number of columns and their data types must be compatible. If the data types don't match, the engine sometimes chooses the most compatible for you. The results might work, but then again, they might not.

By default, UNION sorts records by the values in the first column because UNION uses an implicit DISTINCT predicate to omit duplicate records. To include all records, including duplicates, use UNION ALL, which eliminates the implicit sort. If you know there are no duplicate records, but there are a lot of records, you can use UNION ALL to improve performance because the engine will skip the comparison that's necessary to sort (to find duplicates).

4. Unless you need a different behavior, always owner qualify your objects when you reference them in TSQL. Use dbo.sysdatabases instead of just sysdatabases.
5. Use set nocount on at the top of each stored procedure (and set nocount off) at the bottom.
6. Think hard about locking. If you're not writing banking software, would it matter that you take a chance on a dirty read? You can use the NOLOCK hint, but it's often easier to use SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED at the top of the procedure, then reset to READ COMMITTED at the bottom.
7. I know you've heard it a million times, but only return the columns and the rows you need.
8. Use transactions when appropriate, but allow zero user interaction while the transaction is in progress. I try to do all my transactions inside a stored procedure.

9. Avoid temp tables as much as you can, but if you need a temp table, create it explicitly using Create Table #temp.

10. Avoid NOT IN, instead use a left outer join – even though it's often easier to visualize the NOT IN.

11. If you insist on using dynamic SQL (executing a concatenated string), use named parameters and sp_executesql (rather than EXEC) so you have a chance of reusing the

query plan. While it's simplistic to say that stored procedures are always the right answer, it's also close enough that you won't go wrong using them.

12. Get in the habit of profiling your code before and after each change. While you should keep in mind the depth of the change, if you see more than a 10-15% increase in CPU, Reads, or Writes it probably needs to be reviewed.

13. Look for every possible way to reduce the number of round trips to the server. Returning multiple result sets is one way to do this.

14. Avoid index and join hints.

15. When you're done coding, set Profiler to monitor statements from your machine only, then run through the application from start to finish once. Take a look at the number of reads and writes, and the number of calls to the server. See anything that looks unusual? It's not uncommon to see calls to procedures that are no longer used, or to see duplicate calls. Impress your DBA by asking him to review those results with you.

16. GROUP BY considerations

SQL's GROUP BY clause defines subsets of data. The most important thing to remember when including a GROUP BY clause is to include only those columns that define the subset or summarize data for the subset. In other words, a GROUP BY can't include extraneous data. For instance, to learn the number of orders placed on a specific date, you'd use a statement similar to the following.Please refer the figure 5.7

```
SELECT OrderDate, Count(OrderID)
FROM Orders
GROUP BY OrderDate
```

Figure 5.7-Group by clause

This query would return one record for each date. Each record would display the date and the number of orders for that date. You can't include any other columns.

GROUP BY is versatile. You don't need to specify a column in the SELECT clause to group by it. For instance, you could omit OrderDate from the above query and return just the count for each date (although the results wouldn't make much sense). As long as the GROUP BY column is in the source, SQL doesn't require it in the SELECT clause. On the other hand, if you refer to a column in the SELECT clause, you must also include it in the GROUP BY clause or in an aggregate function. For instance, the following statement doesn't work because the freight column isn't part of an aggregate or the GROUP BY clause. Please refer the figure 5.8

```
SELECT OrderDate, Count(OrderID) AS TotalForDate, Freight
FROM Orders
GROUP BY OrderDate
```

Figure 5.8-Group by clause with count

In truth, it doesn't really make sense to try to include a column in this way. If you want the Freight data within the context of a GROUP BY query, you probably want a summary of the freight values in the group, as follows:

```
SELECT OrderDate, Count(OrderID) Max(Freight)
FROM Orders
GROUP BY OrderDate
```

Figure 5.9-Group by clause with more column

Jet can't group a Memo or OLE Object column. In addition, you can't include a GROUP BY clause in an UPDATE statement, which makes sense. SQL would have no way of knowing which record to update.

17. Retrieving only what you need

It's tempting to use the asterisk character (*) when retrieving data via a SELECT clause, but don't, unless you really need to retrieve all columns. The more data you retrieve, the slower your application will perform. For optimum performance, retrieve only the columns you need.

5.6 Overall System

The overall solution has been implemented as windows application and SQL server scripts that can be accessed by any Windows SQL servers. This is primarily client-server architecture. The application is primarily C# based solution with additional use of SQL server data as the main input source.

5.7 Summary

This chapter mainly described the overall architecture and the design of each component with relevant technologies and their interconnections. Reason to use particular component and its functionality also described. Further use case diagrams, sequence diagrams, and database diagrams also listed in this section.
The following chapter is mainly discussed the evaluation details of the Database monitoring application and optimization techniques. It will present some important code segments and related implementation details.

Chapter 6

Evaluation

6.1 Introduction

In this chapter, we will evaluate using the large volume of the database and show the proposed new techniques and developed existing techniques, for the purpose of increasing database performance.

6.2 Setup

Checked with real-time data and more than 250,000 data records have been retrieved.

Analyzed new queries by using SQL Sentry Plan Explorer tool and compare with old queries.

Analyze new queries by using query execution plan. (QEP)

Check with the different type of database.

6.3 Evaluation Methodology for Database Monitor Application

For proper evaluation of the system functionality, the system should be deployed in an actual production environment. Since selected server and a database chosen for the demonstration purposes, selected set of scripts were running under the selected database and check the result by using database monitoring application. Also, check the values by using SQL query analyzer.

Please refer Appendix B for evaluation of database monitoring application.

6.4 Evaluation Methodology for Proposed New Optimization Techniques.

Scenario 01

Step 01
Create two databases in SQL server. Restore database backup to created two databases and name as "Database_before_optimized" and "Database_after_optimized"

Please refer figure 6.1 of Appendix C for database configuration.

Step 02

Create a complex query. Please refer figure 6.1.2 for SQL query.

```
SELECT DISTINCT 'Doctor fees' AS TrnTypeCode, DFRH.ReceiptNo, DFRH.BHTNo,
DFRH.ReferenceNo, DFRH.ReceiptAmount, 0.00 AS PaidAmount,
DFRH.MachineCode,DFRH.MachineBillNo,'D'AS AdvanceReceiptType, DFRH.CreateUser,
DFRH.CreateDate,DFRH.ModifiedUser,DFRH.IsVoid,DFRH.SessionID,
DFRP.PaymentType, PT.[Description] AS [PaymentTypeName],
DFRP.PaymentNo,DFRP.CardType,DFRP.BankCode,DFRP.ChequeDate,
DFRP.CommonReferanceDetails, DFRP.SettledAmount,
CEL.dLogDate, CEL.dLogOutDate, (TL.Description + ' ' + PA.FirstName + ' ' +
PA.LastName) AS PatientName
,'' as DoctorCode , DFRP.SettledAmount as DocAmount, '' AS ProfessionalName
FROM [HMS].[BILL_TRN_DoctorFeeReceiptHeader] AS DFRH
JOIN [HMS].[BILL_TRN_DoctorFeeReceiptPayment] AS DFRP ON DFRH.ReceiptNo =
DFRP.ReceiptNo
JOIN [HMS].[Sys_Audit_TRN_CashierEventLog] AS CEL ON DFRH.SessionID =
CEL.nLogRecId
JOIN [HMS].[BILL_Comm_MST_PaymentType] AS PT ON
LTRIM(RTRIM(DFRP.PaymentType)) = LTRIM(RTRIM(PT.PaymentCode))
JOIN [HMS].[BILL_Comm_MST_PatientAdmissionHeader] AS PA ON PA.BHTNo =
DFRH.BHTNo
 JOIN [HMS].[BILL_TRN_DoctorFeeHeader] AS DFH ON DFRH.BHTNo = DFH.BHTNo
AND DFH.DocReceiptNo = DFRH.ReceiptNo
LEFT OUTER JOIN (SELECT * FROM [HMS].[BILL_Comm_MST_ReferenceData]
WHERE Modulecode ='BILL_COMM_MST_TITLE') AS TL
ON LTRIM(RTRIM(PA.Title)) = LTRIM(RTRIM(TL.ReferenceCode))
```

Figure 6.1.2 – Complex SQL Query

Step 03

Run this in "Database_before_optimized"

Check execution time.

Execution time = 3 seconds. It is too much.

Please refer figure 6.2 of Appendix C for check the query execution time

Step 04

Now run the QEP plan.

Please refer the Figure 6.3 of Appendix C for QEP Plan

Identify the query cost.

Step 05

Create proper index to the database, “Database_after_optimized ". Please refer figure 6.3.1 for SQL query.

```
USE [Database_after_optimized]
GO
CREATE NONCLUSTERED INDEX [IX_cashiereventlog]
ON [HMS].[Sys_Audit_TRN_CashierEventLog] ([nLogRecId])
INCLUDE ([dLogDate],[dLogOutDate])
GO
```

Figure 6.3.1 – Proper Index.

Step 06

Now run again the query in “Database_after_optimized”

Check the execution time and you will see that it has been reduced by 1 second.

Please refer the figure 6.4 of Appendix C for check the query execution time

The difference between before and after indexes are shown in figure 6.4.1

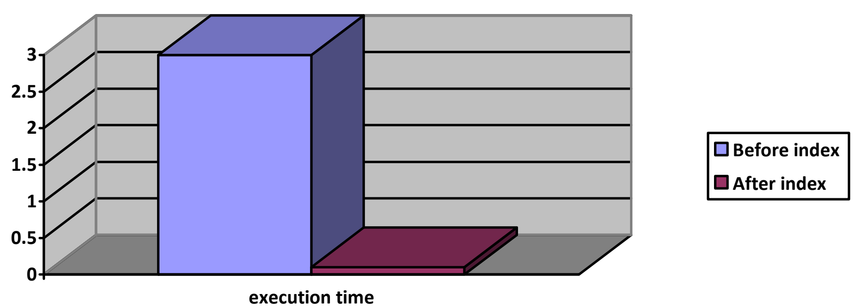


Figure 6.4.1 - Difference between before and after indexes

Scenario 02.

Step 01

Find bad queries and optimize it.

Using SQL profiler, find out the high execution time queries.

Please refer the Figure 6.5 – SQL Profiler of Appendix C

Take high execution query and optimize it. The query in below specified will take more execution time.

```sql
select a.BHTNo,a.FirstName,a.LastName,* from hms.INV_TRN_BillEntryDetails as s
inner join hms.INV_TRN_BillEntryHeader as d on s.EntryNo=d.EntryNo
inner join hms.BILL_Comm_MST_PatientAdmissionHeader as a on a.BHTNo=d.BHTNo
where ItemCode in (select itemcode from hms.BILL_Comm_MST_Item)
andcostcentercode in
(select costcentercode from hms.BILL_Comm_MST_CostCenterHeader)
and a.roomno in (select roomno from hms.BILL_Comm_MST_Room)
```

Figure 6.5.1 – Take high execution query by SQL Profiler

New Solution found by me:
Avoid use IN Keyword, by using the temp table and join with the query.

Optimized query

```sql
CREATE TABLE #TempTable(ID varchar(50)) INSERT INTO #TempTable (ID)
select distinct itemcode from hms.BILL_Comm_MST_Item
select a.BHTNo,a.FirstName,a.LastName,* from hms.INV_TRN_BillEntryDetails as s
inner join #TempTable as t on t.ID=s.ItemCode
inner join hms.INV_TRN_BillEntryHeader as d on s.EntryNo=d.EntryNo
inner join hms.BILL_Comm_MST_PatientAdmissionHeader as a on a.BHTNo=d.BHTNo
where --ItemCode in (select itemcode from hms.BILL_Comm_MST_Item)--and
costcentercode in (select costcentercode from hms.BILL_Comm_MST_CostCenterHeader)
and a.roomno in (select roomno from hms.BILL_Comm_MST_Room)
drop table #TempTable
```

Please refer the figure 6.6 of Appendix C or SQL Profiler result

The difference between before and after is shown in figure 6.6.1

	Elapsed time(duration)mille seconds
Query With IN operator	25809
After optimizing the query	19673

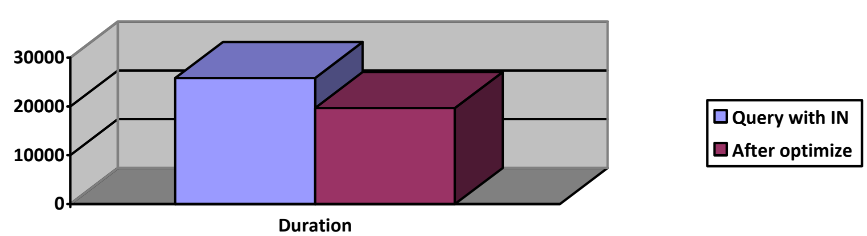


Figure 6.6.1 - Difference between before and after query optimized

Scenario 03.

If problematic query consists IN Operator then you must remove that and apply new method as below mentioned way.

Below Example will be showing the accuracy of this new technique.

Optimization technique - Remove IN operator and create a #temp table instead of that and then insert necessary data to that table.

Step 01

Traditional Query. Please refer the 6.7.1 for Traditional query

```
SELECT S.*
FROM hms.INV_TRN_BillEntryDetails AS s
INNER JOIN hms.INV_TRN_BillEntryHeader AS d
 ON s.EntryNo = d.EntryNo
INNER JOIN hms.BILL_Comm_MST_PatientAdmissionHeader AS a
 ON a.BHTNo = d.BHTNo
WHERE ItemCode IN (SELECT DISTINCT
 itemcode
FROM hms.BILL_Comm_MST_Item)
AND costcentercode IN (SELECT
 costcentercode
FROM hms.BILL_Comm_MST_CostCenterHeader)
AND a.roomno IN (SELECT
 roomno
FROM hms.BILL_Comm_MST_Room)
```

Figure 6.7.1 – Traditional Query

Please refer the figure 6.7 of Appendix C

The proposed new technique to optimize the query.

- Carefully analyze and find whether the below query consists of the join, nested loop, IN, where, group by, order by...
- If IN operator found, then aim to that place.
- Remove IN operator, create #temp table and insert data into the temp table
- Then join the temp table to the main query. Be careful to create an index to that temp table.

Proposed New Query

```
set statistics time on
CREATE TABLE #TempTable (
  ID varchar(50)
)
INSERT INTO #TempTable (ID)
  SELECT DISTINCT
    itemcode
  FROM hms.BILL_Comm_MST_Item

create nonclustered index IX_Itemcode on #TempTable(ID)
SELECTS.*
FROM hms.INV_TRN_BillEntryDetails AS s
INNER JOIN hms.INV_TRN_BillEntryHeader AS d
  ON s.EntryNo = d.EntryNo
INNER JOIN hms.BILL_Comm_MST_PatientAdmissionHeader AS a
  ON a.BHTNo = d.BHTNo
INNER JOIN #TempTable AS t
  ON t.ID = s.ItemCode
WHERE costcentercode IN (SELECT
  costcentercode
FROM hms.BILL_Comm_MST_CostCenterHeader)
AND a.roomno IN (SELECT
  roomno
FROM hms.BILL_Comm_MST_Room)
DROP TABLE #TempTable
set statistics time off
```

Please refer the figure 6.8 - SQL Server Execution time for our new proposed query of Appendix C

See the difference. The new technique will be faster than 10 times compared to the old technique. Please refer the Table 6.8.1 for Differences between with IN and Remove IN.

	CPU TIME	ELAPSED TIME	Operators	Waits
01.With IN	2906	56886	15	2
02.After optimized	4141	5446	4	0

Table 6.8.1 – Differences between with IN and Remove IN

Step 02

Analyze by using Sentry Plan explore

With in - Traditional Query

Please refer the Figure 6.9 - Analyze by using Sentry Plan explore with IN of Appendix C

After optimized- Remove in operator and create the #temp table, then create the internal index

Please refer the Figure 6.10 -Analyze by using Sentry Plan explore without IN of Appendix C

See the difference. The new technique will be faster than more compared to the old technique

Scenario 03.

If Problematic query consists temp tables then avoid them as much as you can, but if you need a temp table, create it explicitly using Create Table #temp

Create it explicitly using Create Table #temp. Please refer the Figure 6.11.1 – Query with temp table

```
SET STATISTICS time ON
WITH BASE
AS (SELECT ProductID,
  YEAR(TransactionDate) AS TransCurrYear,
  COUNT(1) AS NoTrans
FROM Production.TransactionHistory
GROUP BY ProductID,
        YEAR(TransactionDate))
SELECT
  CurrYear.ProductID,
  CurrYear.NoTrans AS CurrTransCnt,
  PrevYear.NoTrans AS PrevTransCnt,
  Prev2Year.NoTrans AS Prev2YearCnt
FROM BASE AS CurrYear
CROSS APPLY (SELECT *
FROM BASE PrevYear
WHERE CurrYear.ProductID = PrevYear.ProductID
AND CurrYear.TransCurrYear = PrevYear.TransCurrYear - 1) AS PrevYear
OUTER APPLY (SELECT *
FROM BASE Prev2Year
WHERE CurrYear.ProductID = Prev2Year.ProductID
AND CurrYear.TransCurrYear = Prev2Year.TransCurrYear - 2) AS Prev2Year
SET STATISTICS time OFF
```

Figure 6.11.1 – Query with temp table

Please refer the figure 6.11 – Query cost with temp table of Appendix C

Observations from Query 1
There were over 27 scans with logical reads of 1998
Although we used a CTE Operation, we could see that similar aggregations are being repeated in the plan.

Let's now use a Temporary Table. Please refer the figure 6.11.2 – Query with #temp table

```
set statistics time on
CREATE TABLE #T1
 (ProductID int
 ,TransCurrYear int
 ,NoTrans int
 );
  CREATE CLUSTERED INDEX CI_#T1 ON #T1 (TransCurrYear)
  INSERT INTO #T1
 SELECT ProductID, YEAR(TransactionDate) AS TransCurrYear, COUNT(1) AS NoTrans
 FROM Production.TransactionHistory
 GROUP BY ProductID, YEAR(TransactionDate)
 ORDER BY YEAR(TransactionDate)
  ;With BASE AS
 (
 SELECT * FROM #T1
 )
 SELECT CurrYear.ProductID, CurrYear.NoTrans AS CurrTransCnt, PrevYear.NoTrans AS
PrevTransCnt, Prev2Year.NoTrans AS Prev2YearCnt
 FROM BASE AS CurrYear
 CROSS APPLY (SELECT *
 FROM BASE PrevYear
 WHERE CurrYear.ProductID = PrevYear.ProductID
 AND CurrYear.TransCurrYear = PrevYear.TransCurrYear - 1 ) AS PrevYear
 OUTER APPLY (SELECT *
 FROM BASE Prev2Year
 WHERE CurrYear.ProductID = Prev2Year.ProductID
 AND CurrYear.TransCurrYear = Prev2Year.TransCurrYear - 2 ) AS Prev2Year
 drop table #T1
 set statistics time off
```

Figure 6.11.2 – Query with #temp table

Please refer the figure 6.12 - Query cost with #temp table of Appendix C

Observations from Query 2:
Logical Reads are down when compared to Query1.
No Repetition of computation of Aggregated Values.
SQL Server uses statistics as can be seen from the properties above which is good when data is more.

Great. Let's now do it with a table variable.

Please refer the figure 6.13.1 - Query with @temp table

```
set statistics time on
DECLARE @T1 AS TABLE
 (ProductID int
 ,TransCurrYear int
 ,NoTrans int
 , INDEX [IX_TransactionYear] CLUSTERED (ProductID,TransCurrYear)
 );

  INSERT INTO @T1
 SELECT ProductID, YEAR(TransactionDate) AS TransCurrYear, COUNT(1) AS NoTrans
 FROM Production.TransactionHistory
 GROUP BY ProductID, YEAR(TransactionDate)
 ORDER BY YEAR(TransactionDate)
  ;With BASE AS
 (
 SELECT * FROM @T1
 )
 SELECT CurrYear.ProductID, CurrYear.NoTrans AS CurrTransCnt, PrevYear.NoTrans AS
PrevTransCnt, Prev2Year.NoTrans AS Prev2YearCnt
 FROM BASE AS CurrYear
 CROSS APPLY (SELECT *
 FROM BASE PrevYear
 WHERE CurrYear.ProductID = PrevYear.ProductID
 AND CurrYear.TransCurrYear = PrevYear.TransCurrYear - 1 ) AS PrevYear
 OUTER APPLY (SELECT *
 FROM BASE Prev2Year
 WHERE CurrYear.ProductID = Prev2Year.ProductID
 AND CurrYear.TransCurrYear = Prev2Year.TransCurrYear - 2 ) AS Prev2Year
 set statistics time off
```

Figure 6.13.1 - Query with @temp table

Please refer the figure 6.13 - Query cost with @temp table of Appendix C

Scan Count and Logical Reads are slightly up.
There are no statistics associated with Clustered Key Creation in Table Variable which can be seen from the Estimated Number of Rows Value. This could be bad when data is more.

One reason why the Optimizer could not sniff the "Estimated Number of Rows" is because the entire batch query populates the variable table followed by querying on it. And hence, it is not able to figure out the cardinality. If we add the RECOMPILE Option to the query, SQL Server is able to detect the cardinality like so,

Conclusion: Temporary Table is better.

Let’s now do it with Sentry plan

Create it explicitly using Create Table #temp
Please refer the figure 6.14 - Sentry plan with #temp table of Appendix C

Temporary Table.
Please refer the figure 6.15 - Sentry plan with @temp table of Appendix C

See the difference in Total time.

Please refer the figure 6.15.1 – #Table and @Table Difference

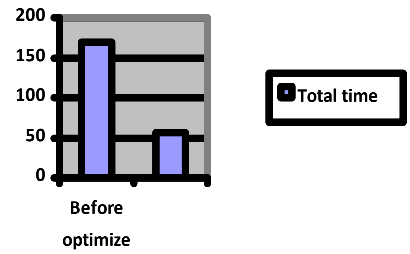


Figure 6.15.1 – #Table and @Table Difference

Scenario 04.

How to find the Missing index.

Please refer the figure 6.16 – How to find missing index of Appendix C

Scenario 05.

Best practice for IN and Where.

Please refer the Figure 6.17.1 - Best practice for IN and Where.

```
  SELECT top 2000  [CostPrice]
      ,[Quantity]
      ,[DiscountAmount]
      ,[CreditAmount]
      ,[DebitAmount]
      ,[WardNo]
      ,[RoomNo]
      ,[ProcessDate]
      ,[IsDayEnd]
      ,[IsAdditionalItem]
      ,[UniqueID]
      ,[IsVoid]
      ,[IsPackageEntry]
      ,[IsPackageBillProcessed]
      ,[RateType]
      ,[RecordMode]
 FROM [Database_before_optimized].[HMS].[INV_TRN_BillEntryDetails]
 where  ItemCode ='ITM0001552'
```

Figure 6.17.1 - Best practice for IN and Where.

Please refer the Figure 6.17 – Analyzed best practice IN and Where Clause. Of Appendix C

Bad practice for IN and Where

Please refer the Figure 6.18.1 – Bad practice for IN and Where.

```
SELECT top 2000   [EntryNo]
      ,[CostCenterCode]
      ,[SortSequence]
      ,[ItemCode]
      ,[UnitPrice]
      ,[CostPrice]
      ,[Quantity]
      ,[DiscountAmount]
      ,[CreditAmount]
      ,[DebitAmount]
      ,[WardNo]
      ,[RoomNo]
      ,[ProcessDate]
      ,[IsDayEnd]
      ,[IsAdditionalItem]
      ,[UniqueID]
```

```
      ,[IsVoid]
      ,[IsPackageEntry]
      ,[IsPackageBillProcessed]
      ,[RateType]
      ,[RecordMode]
      ,[AdditionalQuantity]
      ,[IsOldEntry]
      ,[PackageId]
  FROM [Database_before_optimized].[HMS].[INV_TRN_BillEntryDetails]
  where  ItemCode in ('ITM0001554')
```

Figure 6.18.1 - Bad practice for IN and Where.

Please refer the figure 6.18 - Analyzed Bad practice IN and Where Clause of Appendix C

Scenario 06.

Bad practice IN and Where clause.

Please refer the Figure 6.19.1 – Bad practice for IN and Where.

```
 SELECT top 2000   [EntryNo]
      ,[CostCenterCode]
      ,[SortSequence]
      ,[ItemCode]
      ,[UnitPrice]
      ,[CostPrice]
      ,[Quantity]
      ,[DiscountAmount]
      ,[CreditAmount]
      ,[DebitAmount]
      ,[WardNo]
      ,[RoomNo]
      ,[ProcessDate]
      ,[IsDayEnd]
      ,[IsAdditionalItem]
      ,[UniqueID]
      ,[IsVoid]
      ,[IsPackageEntry]
      ,[IsPackageBillProcessed]
      ,[RateType]
      ,[RecordMode]
FROM [Database_before_optimized].[HMS].[INV_TRN_BillEntryDetails]
where  ItemCode in (select itemcode from
[Database_before_optimized].[HMS].[BILL_Comm_MST_Item] where
itemcode='ITM0001554')
```

Figure 6.19.1 – Bad practice for IN and Where

Please refer the Figure 6.19 – Bad practice for IN and Where of Appendix C

Scenario 07.

Please refer the Figure 6.20.1 – Correlated SQL subqueries

Avoid Correlated SQL Subqueries

```
SELECT c.FirstName,
       c.LastName,
       (SELECT top 1 ConsultantCode FROM
[HMS].[BILL_Comm_MST_PatientAdmissionDetail] WHERE  [BHTNo] = c.[BHTNo]) AS
Consultant
FROM [HMS].[BILL_Comm_MST_PatientAdmissionHeader] c
```

Figure 6.20.1 – Correlated SQL subqueries

Please refer the figure 6.20 -QEP plan and Cost of Correlated SQL subqueries of Appendix C

Please refer the figure 6.21- QEP plan and Cost of Correlated SQL subqueries in Sentry planner of Appendix C

Solution Correlated SQL Subqueries.

Please refer the Figure 6.22.1 – Correlated SQL subqueries

```
 SELECT c.FirstName,
       c.LastName,
       co.ConsultantCode
FROM [HMS].[BILL_Comm_MST_PatientAdmissionHeader] c
          LEFT JOIN [HMS].[BILL_Comm_MST_PatientAdmissionDetail] co
                   ON c.[BHTNo] = co.[BHTNo]
```

Figure 6.22.1 – Solution for Correlated SQL subqueries

Please refer the figure 6.22- Our Query QEP plan and Cost of Correlated SQL subqueries of Appendix C

Please refer the figure 6.23 - Our Query QEP plan and Cost of Correlated SQL subqueries in Sentry planner of Appendix C

Scenario 08.

Avoid Cursor

Please refer the figure 6.24.1 – Query with Cursor

```
DECLARE @BHTNo varchar(30)
DECLARE @FirstName varchar(30), @LastName varchar(30)
-- declare cursor called
DECLARE ActivePatient Cursor FOR
 SELECT BHTNo, FirstName, LastName
 FROM [HMS].[BILL_Comm_MST_PatientAdmissionHeader]
 WHERE IsDischarge = 1
-- Open the cursor
OPEN ActivePatient
-- Fetch the first row of the cursor and assign its values into variables
FETCH NEXT FROM ActivePatient INTO @BHTNo, @FirstName, @LastName
-- perform action whilst a row was found
WHILE @@FETCH_STATUS = 0
BEGIN
 -- get next row of cursor
   Print @BHTNo
	print @FirstName
	print @LastName

 FETCH NEXT FROM ActivePatient INTO @BHTNo, @FirstName, @LastName
END
-- Close the cursor to release locks
CLOSE ActivePatient
-- Free memory used by cursor
DEALLOCATE ActivePatient
```

Figure 6.24.1 – Query with Cursor

Please refer the figure 6.24 - QEP in Cusror of Appendix C

Alternatives for Cursors.

Create a temporary table, note the IDENTITY column that will be used to loop through
The rows of this table

Please refer the figure 6.25.1 – Alternative solution for cursor

```
CREATE TABLE #ActivePatient (
 RowID int IDENTITY(1, 1),
 BHTNo varchar(30),
 FirstName varchar(130),
 LastName varchar(130)
)
```

```sql
DECLARE @NumberRecords int, @RowCount int
DECLARE @BHTNo varchar(50), @FirstName varchar(130), @LastName varchar(130)

-- Insert the resultset we want to loop through
-- into the temporary table
INSERT INTO #ActivePatient (BHTNo, FirstName, LastName)
SELECT BHTNo, FirstName, LastName
FROM [HMS].[BILL_Comm_MST_PatientAdmissionHeader]
WHERE IsDischarge = 1
-- Get the number of records in the temporary table
SET @NumberRecords = @@ROWCOUNT
SET @RowCount = 1
-- loop through all records in the temporary table
-- using the WHILE loop construct
WHILE @RowCount <= @NumberRecords
BEGIN
SELECT @BHTNo = BHTNo, @FirstName = FirstName, @LastName = LastName
FROM #ActivePatient
WHERE RowID = @RowCount
Print @BHTNo
        print @FirstName
        print @LastName
SET @RowCount = @RowCount + 1
END
-- drop the temporary table
DROP TABLE #ActivePatient
```

6.25.1 – Alternative solution for cursor

We can see the above code gives the same functionality as the first code example but without using a cursor. This gives us the benefits that the Customer table is not locked as we are looping through our result set so other queries on the Customer table that are submitted by other users will execute much faster. We will also have a faster-operating SQL script by avoiding cursors which are slow in themselves.

Please refer the figure 6.25 - Alternative solutinon QEP plan and query cost
Of Appendix C

Cursor Alternative 2: Using User Defined Functions

Cursors are sometimes used to perform a calculation on values that come from each row in its row set. This scenario can also be achieved by replacing a Cursor with a User Defined Function. An example of a User Defined Function performing a calculation is given below:
Please refer the figure 6.26.1 - Using User Defined Functions

```sql
CREATE FUNCTION  dbo.GetDiscountLevel(
 @CustomerID int
)
RETURNS int
```

```
AS
BEGIN
DECLARE @DiscountPercent int
DECLARE @NumberOrders int, @SalesTotal float
SELECT @NumberOrders = COUNT(OrderID),
 @SalesTotal = SUM(TotalCost)
FROM Sales WHERE CustomerID = @CustomerID
 IF @SalesTotal > 5000.00 AND @NumberOrders > 5
  SET @DiscountPercent = 5
 ELSE
 BEGIN
  IF @SalesTotal > 3000.00 AND @NumberOrders > 3
   SET @DiscountPercent = 3
  ELSE
   SET @DiscountPercent = 0
 END

 Return @DiscountPercent
END
```

Figure 6.26.1 - Using User Defined Functions

Scenario 09.

SET NO COUNT ON

Please refer the figure 6.26 – Set no count on execution time of Appendix C

WITHOUT NO COUNT

Please refer the figure 6.27 -– Without no count execution time of Appendix C

See the difference. The new technique will be faster than 10 times compared to the old technique. Please refer the Table 6.5 – Difference between set no count and without no count

	Execution time
With no count on	27
Without no count	40

Table 6.5 – Difference between set no count and without no count

Please refer the figure 6.27.1 – Difference between set no count and without no count

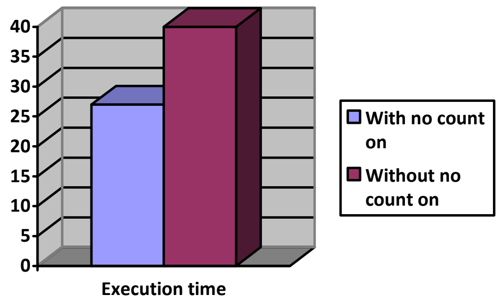


Figure 6.27.1 - Difference between set no count and without no count

6.5 Participants

Basically will involving database administrator, software developers, and end users.

6.6 Data Collection

Large volume database backup from the hospital.

Created database with more records. (Bulk insert)

6.7 Discussion

The party who are mainly benefited by this Database monitoring application and proposed techniques system is the database administrator's, developers, users, and clients. With currently available systems, they are only capable to see the graphical interface of system status. Also currently there is no proper single system to get both database activity and fine tune query techniques. This system facilitates to view both database and database server issues. Limitation on this Support only for Microsoft products.

6.8 Summary

According to the evaluation done in this chapter, the system has maintained its evaluation above the critical line in all evaluation features for both database monitoring application and the proposed query optimization techniques. In next chapter, the conclusion and the further possible enhancements for database performance improvement will be discussed.

Chapter 7

Conclusion and Further Work

7.1 Introduction

In this paper, an attempt is made to present a review of database performance tuning techniques is made. This paper focuses on tuning techniques which are directly related to database design. The main purpose of this study is to understand major factors
That can lead to database performance improvement. As query response time is the number one metrics when it comes to database performance, SQL tuning is one of the widely used tuning technique. SQL tuning aims to decrease response time and increase
System throughput.

7.2 Overall Conclusion

The evaluation evident that the database monitoring application and proposed optimization techniques to address problems in database performance issues, is in above 70% acceptances level accordingly our hypothesis is proven to be true.

7.3 Objective-Wise Conclusion

Objective (i) has been achieved by conducting a comprehensive literature survey comprising more than 25 research papers related to database performance issues. Here we have discovered many key problems and defined the research problems of this thesis.

The achievement of objective (ii) has also been supported by the literature review chapter. In addition, chapter 3 presented a description of each technology selected for developing the proposed solution which fulfills the achievement of objective (iii).

Achievements in objectives (iv), (v) and (vi) are evident from the details in chapters for approach, design, and implementation.

The objective related to the evaluation of the hypothesis is presented in chapter 6. The overall success of the solution has been 70%.

7.4 Further Work

Database monitoring application should be able to work with open source products.

7.5 Summary

In this chapter, the overview of Database performance improvement project was discussed along with the evaluation results and identified limitations. Further possible enhancements were also discussed together with possible practical applications.

References

[1] G. Ramakrishnan. Database Management Systems, Third Edition. McGraw-Hill, 2003

[2] Fox, B. 2011. "Leveraging Big Data for Big Impact", Health Management Technology, http://www.healthmgttech.com/.

[3] A. Hameurlain, "Evolution of Query Optimization Methods: From Centralized Database Systems to Data Grid Systems", Proceedings of the 20th International Conference on Database and Expert Systems Applications.

[4]. Andrew N.K. Chen. Robust optimization for performance tuning of modern database Systems, European Journal of Operational Research. 171, 412--429 (2006)

[5] The State of the Art in Distributed Query Processing DONALD KOSSMANN, University of Passau, ACM Computing Surveys, Vol. 32, No. 4, December 2000.

[6] Jacobs, A. 2009. "Pathologies of Big Data", Communications of the ACM, 52(8):36-44.

[7] JASON. 2008. "Data Analysis Challenges", The Mitre Corporation, McLean, VA, JSR-08-142

[8] Kaisler, S. 2012. "Advanced Analytics", CATALYST Technical Report, i_SW Corporation, Arlington, VA

[9]. Rasha Osman, Irfan Awan, Michael E. et al.: QuePED-Revisiting queuing networks for the Performance evaluation of database designs. Simulation Modeling Practice and Theory, 19, 251--270 (2011)

[10]. Balsamo, S., A. Di Marco, P. Inverardi and M. Simeoni, Model-based performance Prediction in software development: a survey, IEEE Transactions on Software Engineering. 30 ,5, 295--310 (2004)

[11] K. Ono and G.M.Lohman. Measuring the complexity of join enumeration in query optimization. In D.McLeod, R Sacks-Davis, and J.-J. schek, editors,16th International Conference on Very Large Data Bases, August 13-16,1990, Brisbane, Queensland, Australia, Proceedings, pages 314-325. Morgan Kaufmann, 1990.

[12] A. Hameurlain, "Evolution of Query Optimization Methods: From Centralized Database Systems to Data Grid Systems", Proceedings of the 20th International Conference on Database and Expert Systems Applications.

[13] Fox, B. 2011. “Leveraging Big Data for Big Impact”, Health Management Technology, http://www.healthmgttech.com/.

[14] https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html.

[15] Gantz, J. and E. Reinsel. 2011. “Extracting Value from Chaos”, IDC’s Digital Universe Study, sponsored by EMC.

[16] https://cs.uwaterloo.ca/~kmsalem/courses/.../Chalamalla-HadoopDB.pdf

[17] https://en.wikipedia.org/wiki/Greenplum

[18] https://pig.apache.org/

[19] www.cs.rutgers.edu/~zz124/cs671.../srikanth_mapreducemerge.pdf. Map-Reduce-Merge: Simplified Relational Data. Processing on Large. Clusters. Hung-chih Yang, Ali Dasdan. Yahoo! Ruey-Lung Hsiao, D. Sto Parker.

[20] http://www.journalofcloudcomputing.com/content/3/1/12. Improving the performance of Hadoop Hive by sharing scan and computation tasks Tansel Dokeroglu1, Serkan Ozal1, Murat Ali BayMuhammetSerkanCinar3 and Ahmet Cosar1.

[21] Liu et al. "An Investigation of Practical Approximate Nearest Neighbor Algorithms", 2004. Carnegie-Mellon University, pp. 1-8.

[22] www.elsevier.com/locate/jcss, Journal of Computer and System Sciences 77 (2011) 637-651.

[23] Computing Semantic Relatedness using Wikipedia-based Explicit Semantic Analysis, IJCAI-07 1606, Evgeniy Gabrilovich and Shaul Markovitch Department of Computer Science Technion—Israel Institute of Technology, 32000 Haifa, Israel

[24] {gabr,shaulm}@cs.technion.ac.il.

[25] https://en.wikipedia.org/wiki/MapReduce.

[26] Applied Spatial Data Analysis with R Authors: Roger S. Bivand, Edzer Pebesma, Virgilio Gómez-Rubio.

[27] A twelve-analyzer detector system for high-resolution powder diffraction P. L. Lee, D. Shu, M. Ramanathan, C. Preissner, J. Wang, M. A. Beno, R. B. Von Dreele, L. Ribaud, C. Kurtz, S. M. Antao, X. Jiao and B. H. Toby. J. Synchrotron Rad. (2008). 15, 427-432.

Appendixes

Appendix A - User interface and architecture diagram of the system.

Security Module - Authentication

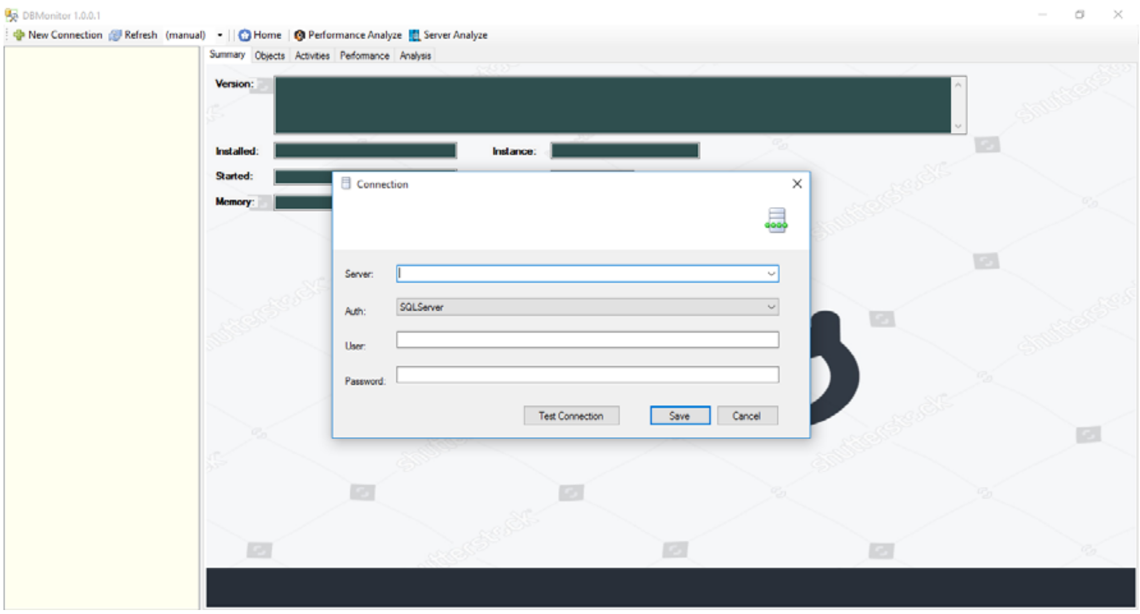


Figure 5.1 – Security Module - Authentication

Control Module – Server and Database Information

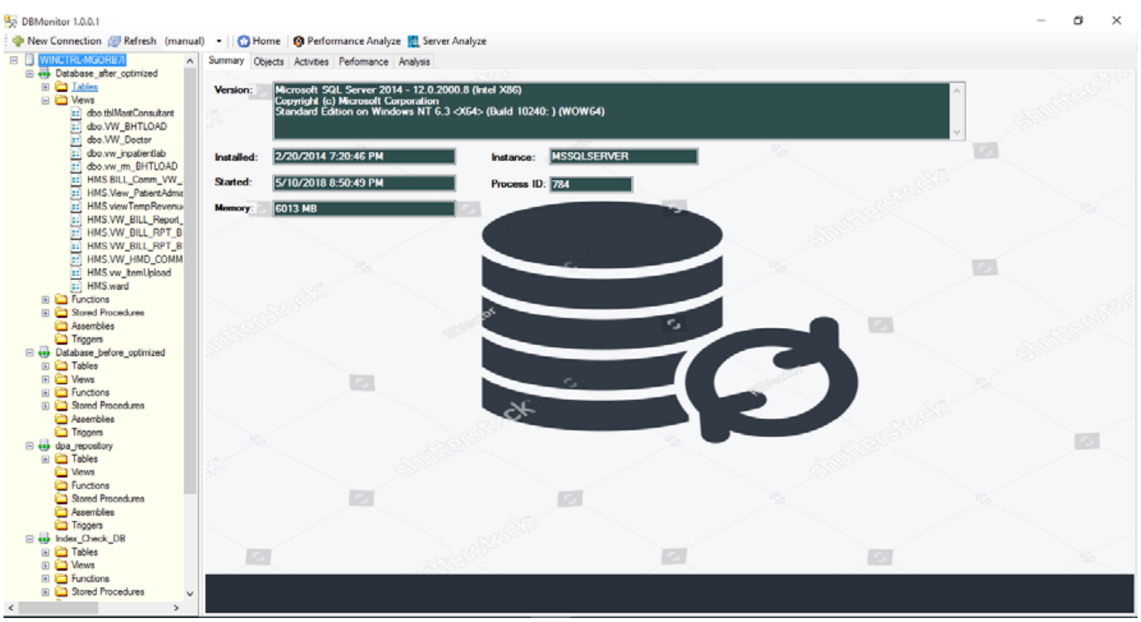


Figure 5.2 – Control Module-Server and Database Information

Server Configuration

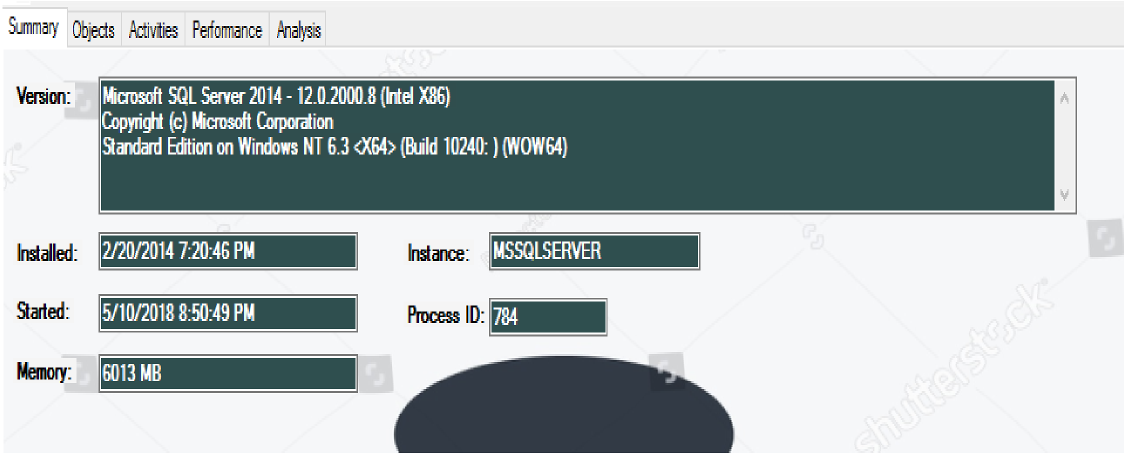


Figure 5.3 – Server Configuration

Database Server Performance Analyzer

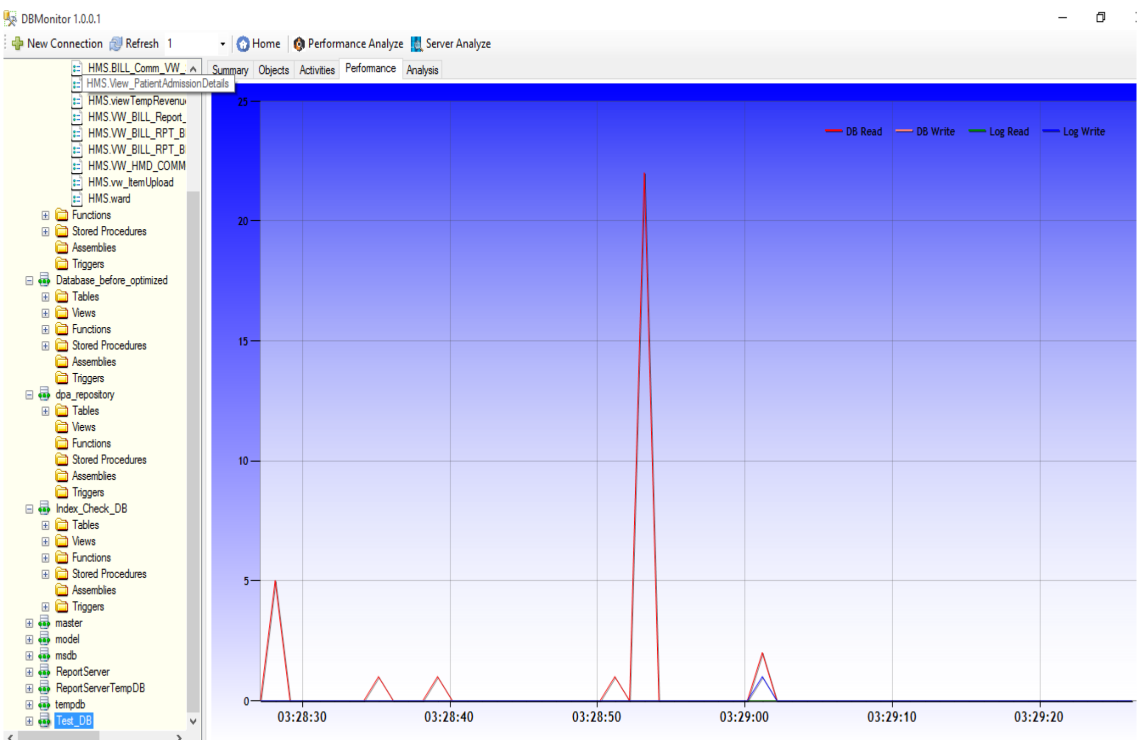


Figure 5.4 – Database Server Performance Analyzer

Database Log Information and Suggestions

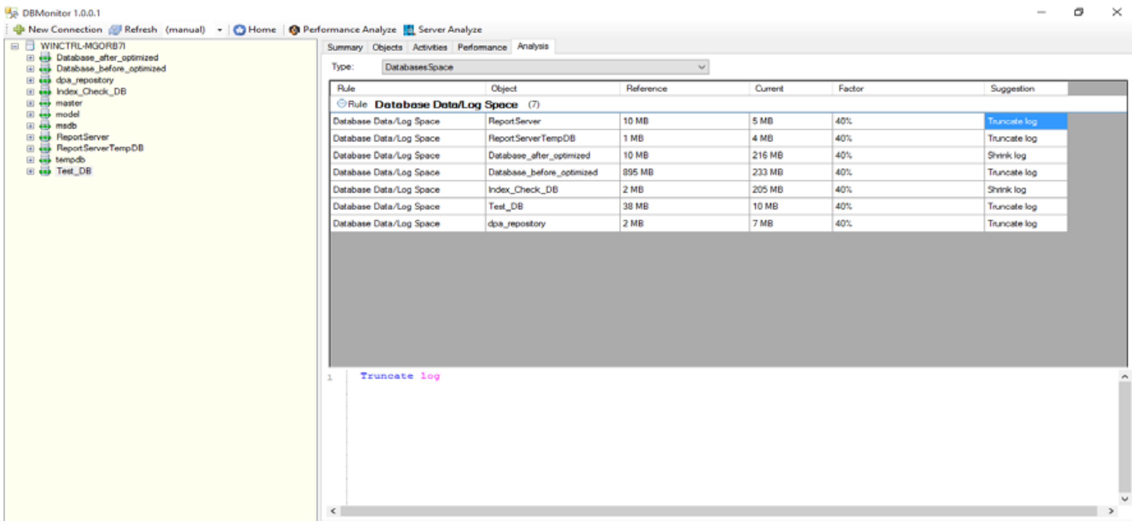


Figure 5.5 – Database Log Information and Suggestions.

Database Performance Improvement Suggestions

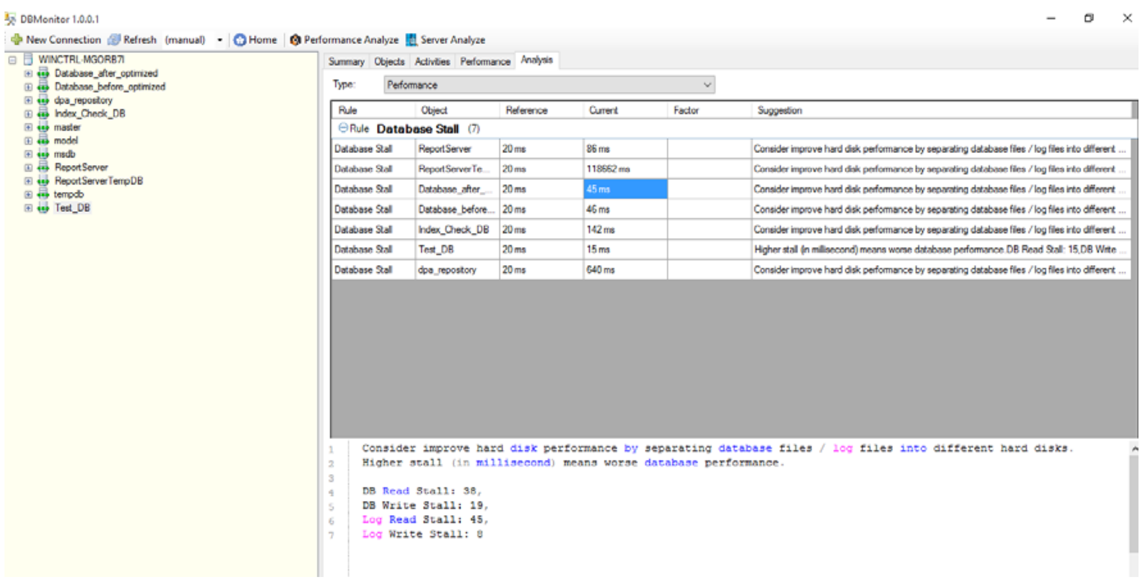


Figure 5.6 – Database Performance Improvement Suggestions.

Database Waiting Tasks

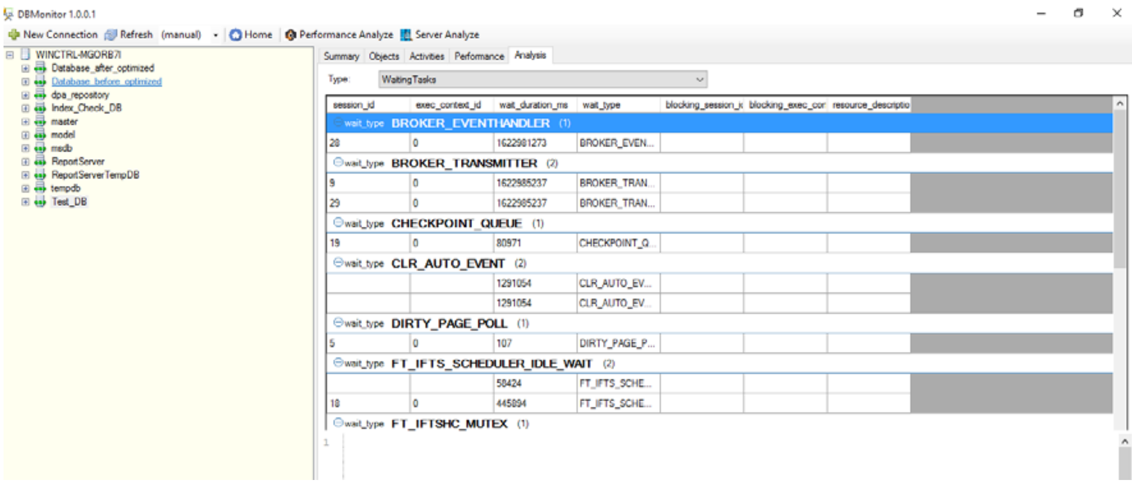


Figure 5.7 – Database Waiting Tasks.

Database Missing Index Details and Suggestions

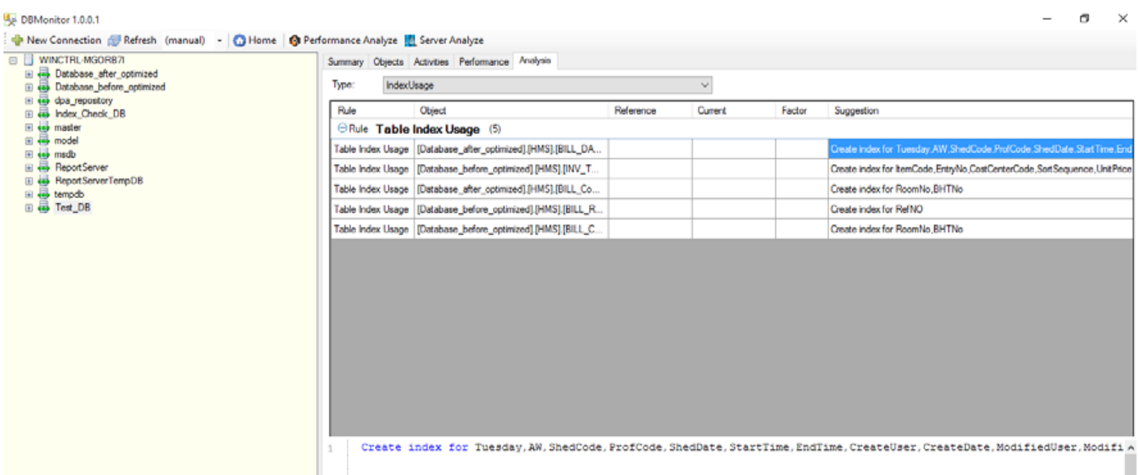


Figure 5.8 – Database Missing Index Details and Suggestions

Database IO operations

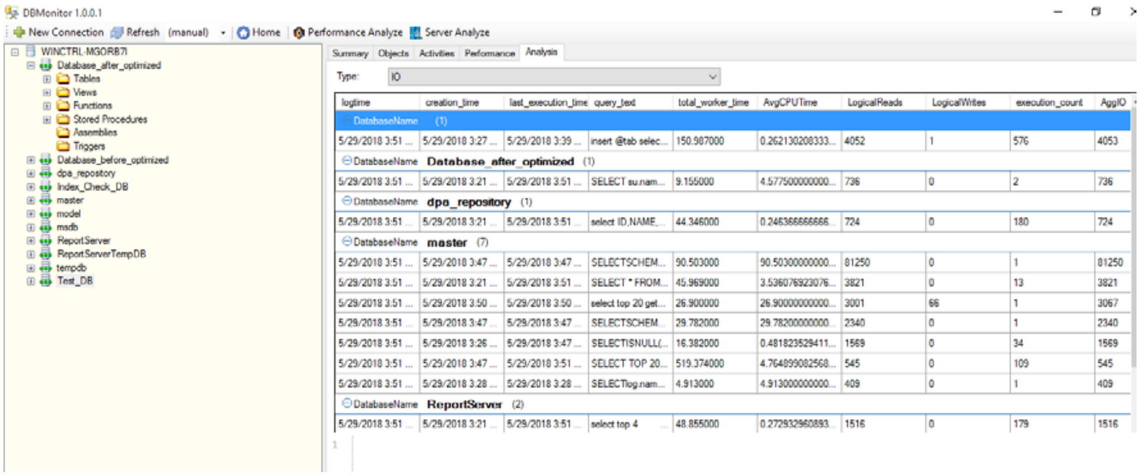


Figure 5.9 – Database IO Operations

Database Objects and Details

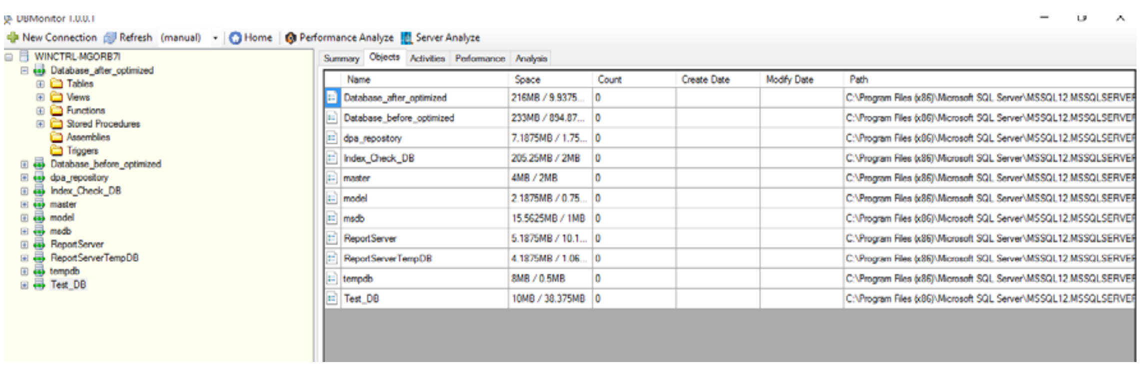


Figure 5.10 – Database Objects and Details

Database Monitoring Application Options

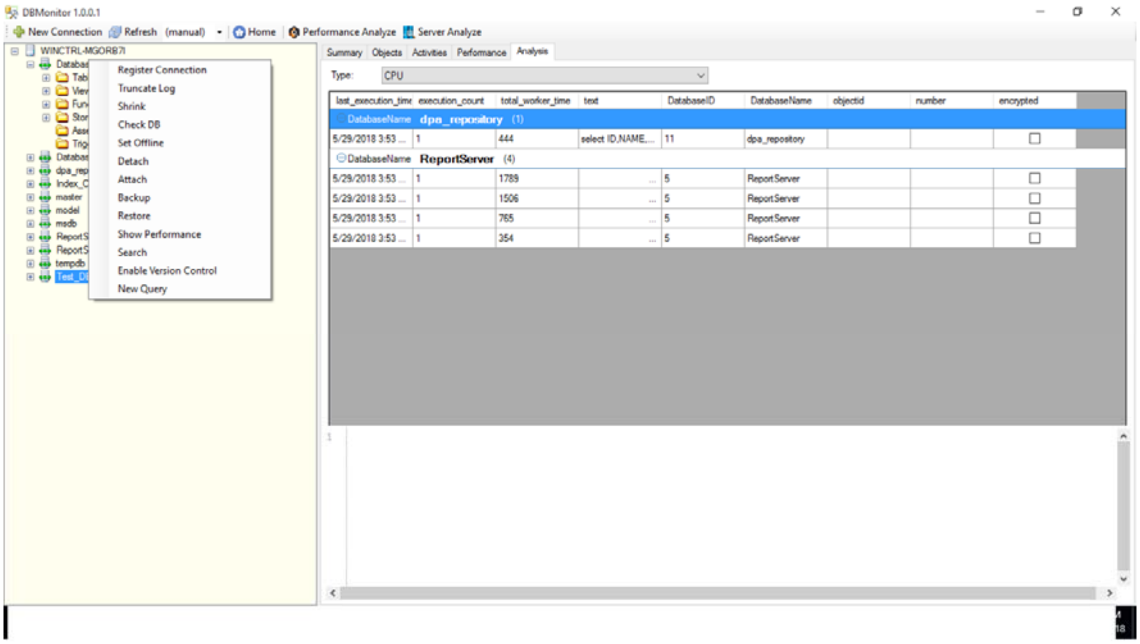


Figure 5.11 – Database Monitoring Application Options

Appendix B – Evaluation of Database Monitoring Application

Functionality	Result in our approach	The result from manually check in the database	Are the both results are same?
Server information	Please refer figure 7.4.1.1	Please refer figure 7.4.1.2	**Yes**
Database Table Count	Please refer figure 7.4.1.3	Please refer figure 7.4.1.4	**Yes**
Database current reading count	Please refer figure 7.4.1.3	Please refer figure 7.4.1.4	**Yes**
Database current writing count	Please refer figure 7.4.1.3	Please refer figure 7.4.1.4	**Yes**
Missing index details	Please refer figure 7.4.1.5	Please refer figure 7.4.1.6	**Yes**
Database current reading count	Please refer figure 7.4.1.3	Please refer figure 7.4.1.4	**Yes**
Database memory utilization details	Please refer figure 7.4.1.7	Manually checked	**Yes**
Database lock	Please refer figure 7.4.1.8	Manually checked	**Yes**
Currently running Processors	Please refer figure 7.4.1.9	Manually checked	**Yes**

Table 7.1 – Evaluation functionality in database monitoring application

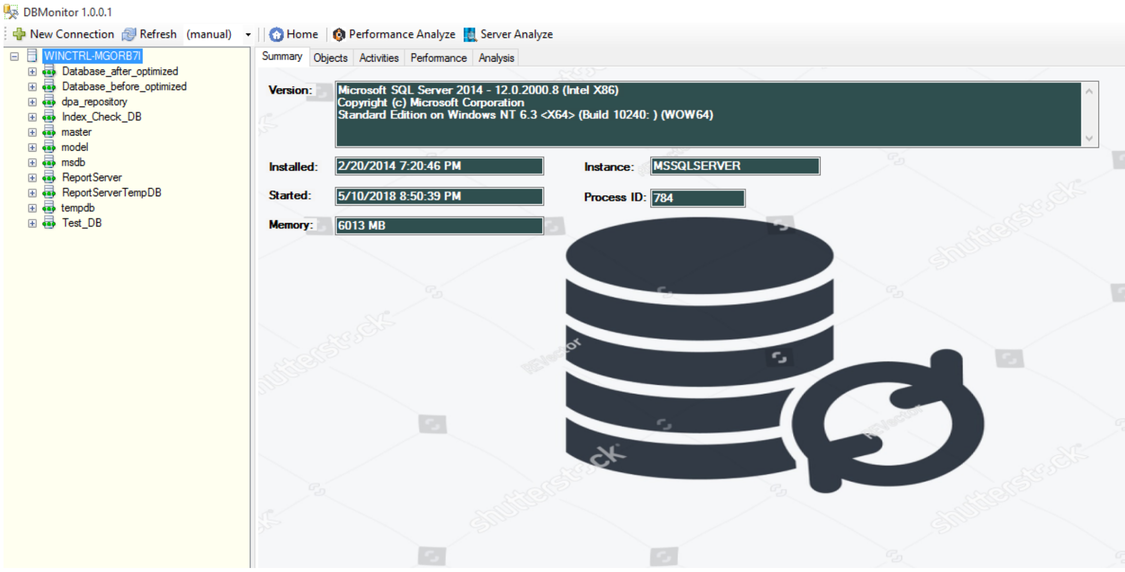


Figure 7.4.1.1 – Database server information from newly developed database monitoring application

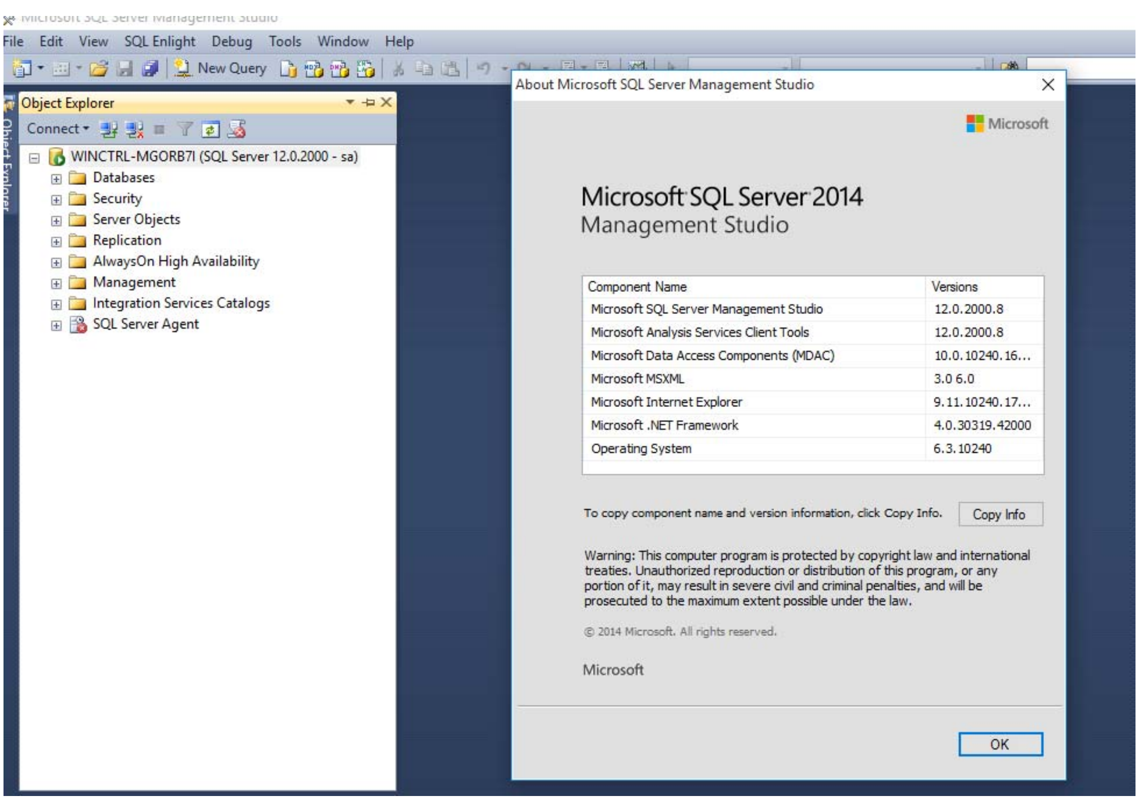


Figure 7.4.1.2 – Database server information

Figure 7.4.1.3 - Database server statistics from newly developed database monitoring application

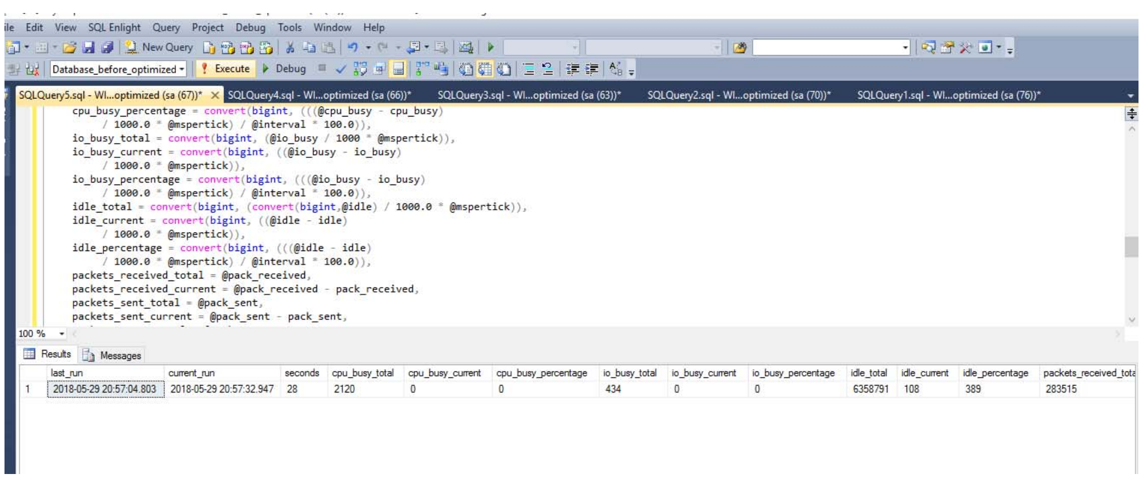


Figure 7.4.1.4 - Database server statistics

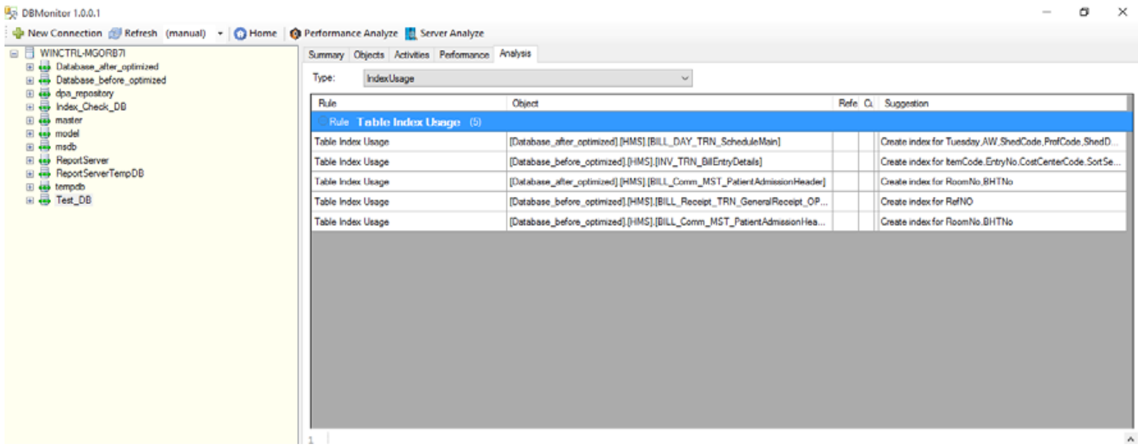


Figure 7.4.1.5 – Missing index suggestions from newly developed database monitoring application

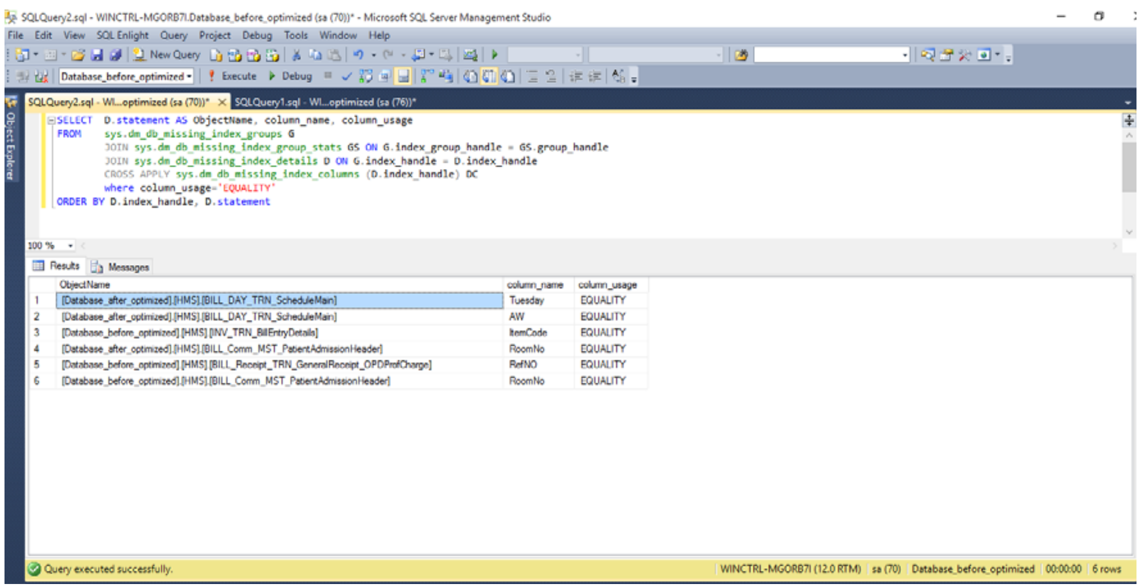


Figure 7.4.1.6 – Missing index suggestions by manually

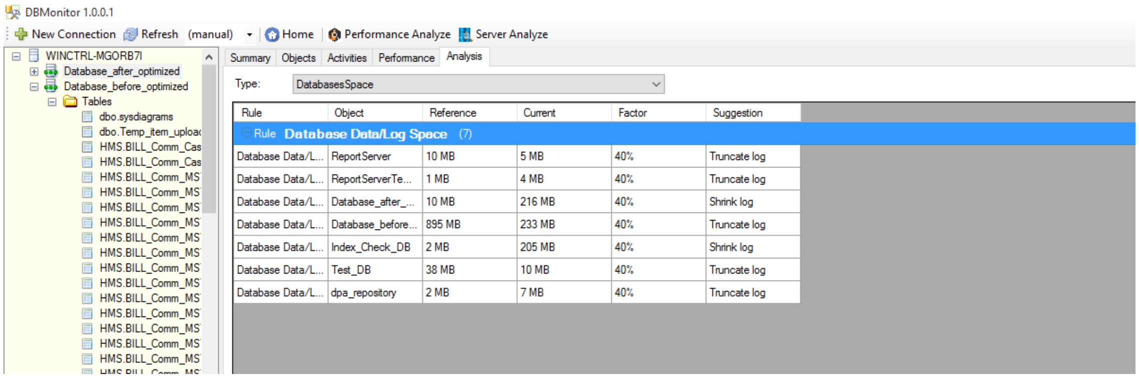


Figure 7.4.1.7 – Database memory utilization details

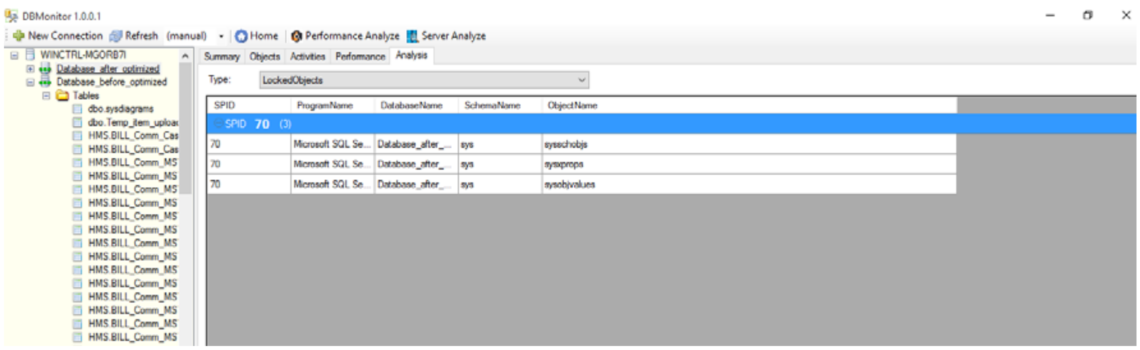


Figure 7.4.1.7 – Database lock

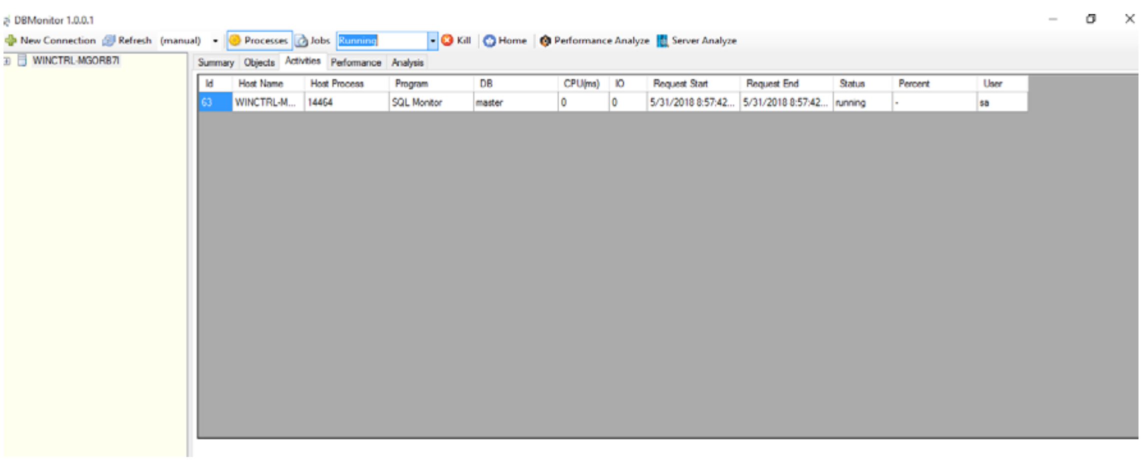


Figure 7.4.1.8 – Currently running Processors

Appendix C – Evaluation of proposed optimization techniques

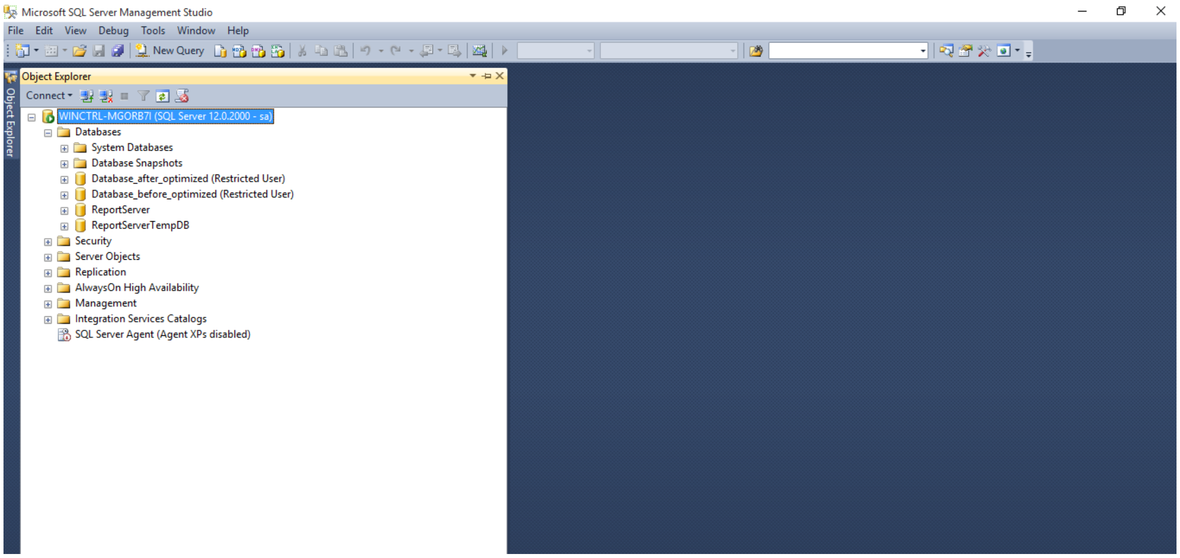


Figure 6.1 – Database Configuration

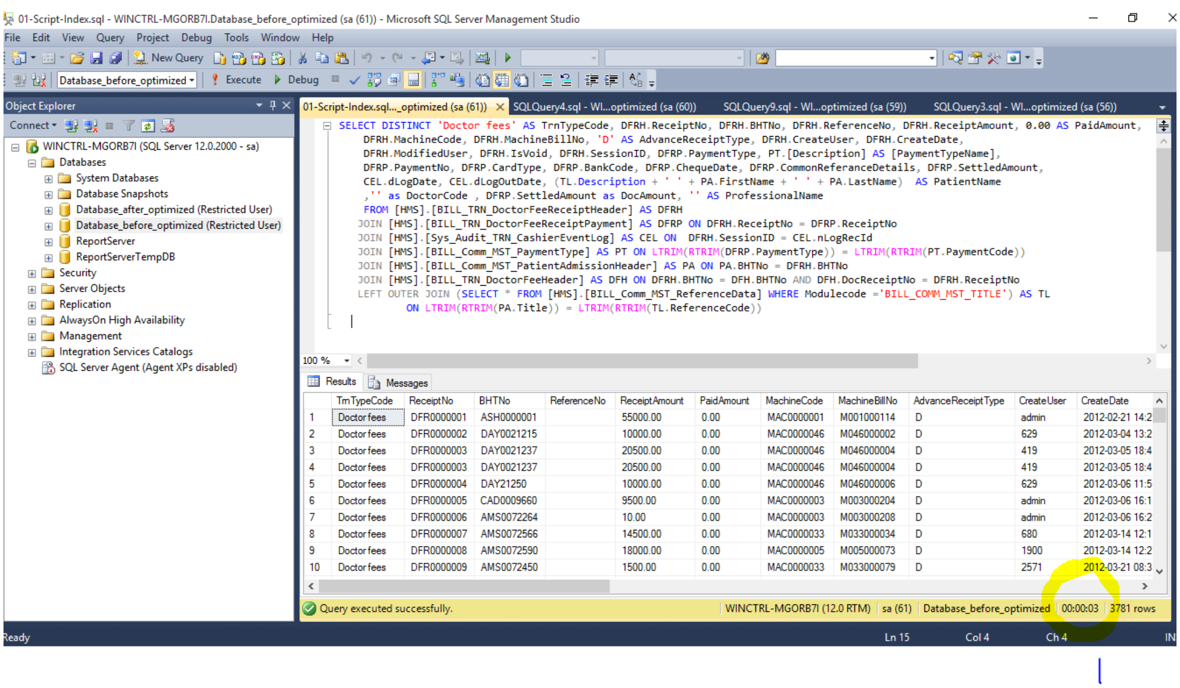


Figure 6.2 – Complex Query Execution Time

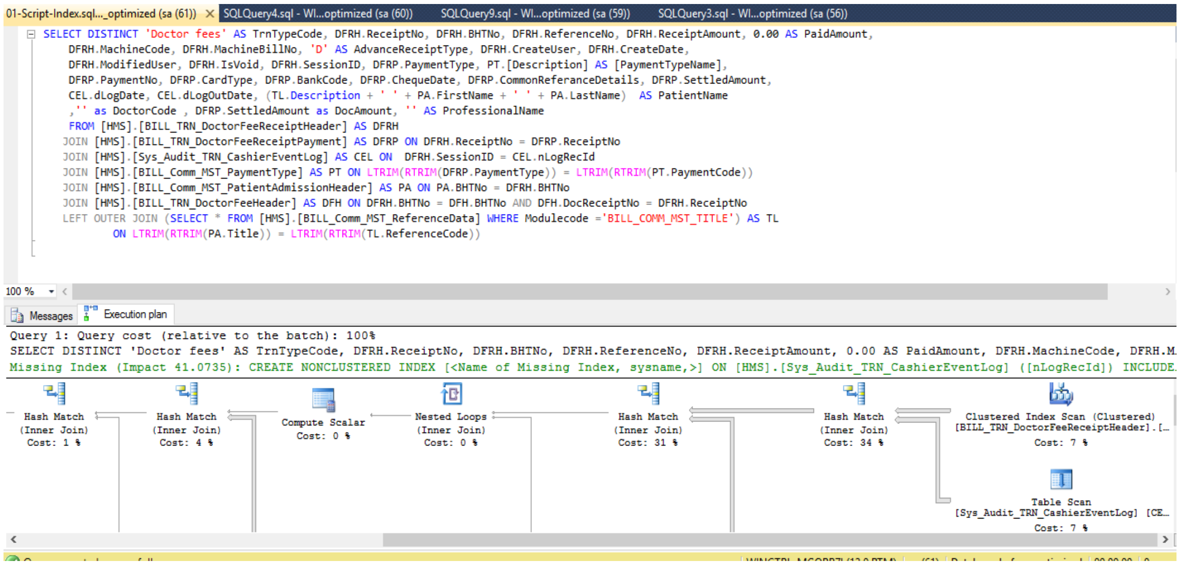


Figure 6.3 - QEP Plan

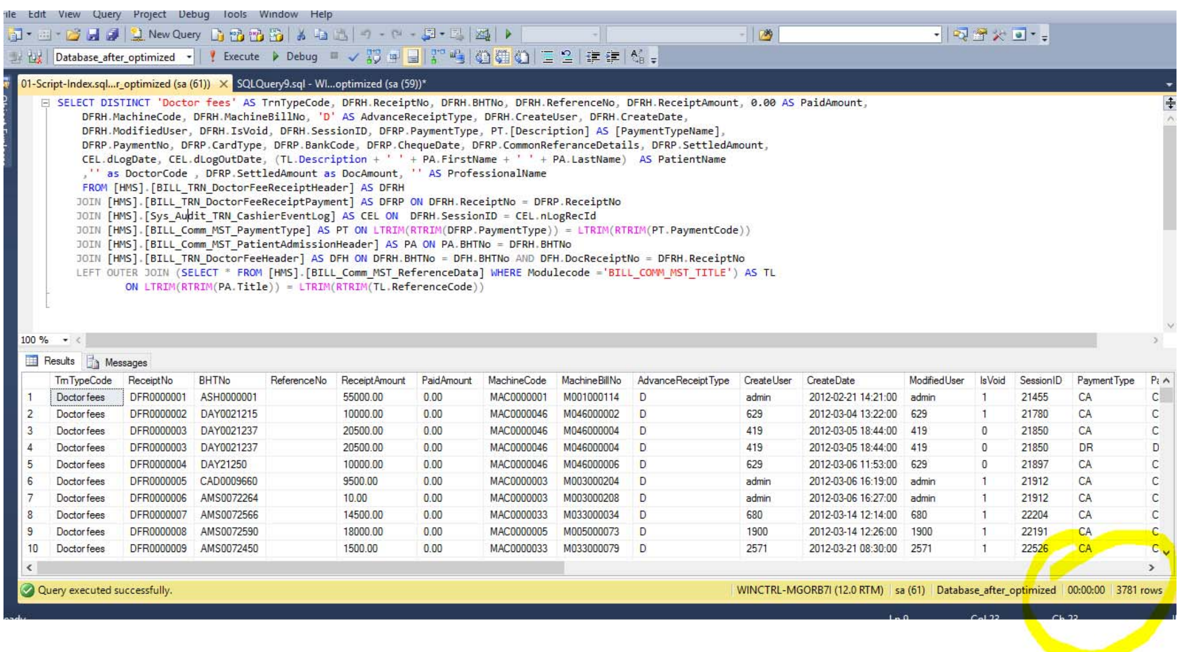


Figure 6.4 – Query Execution Time After Optimized

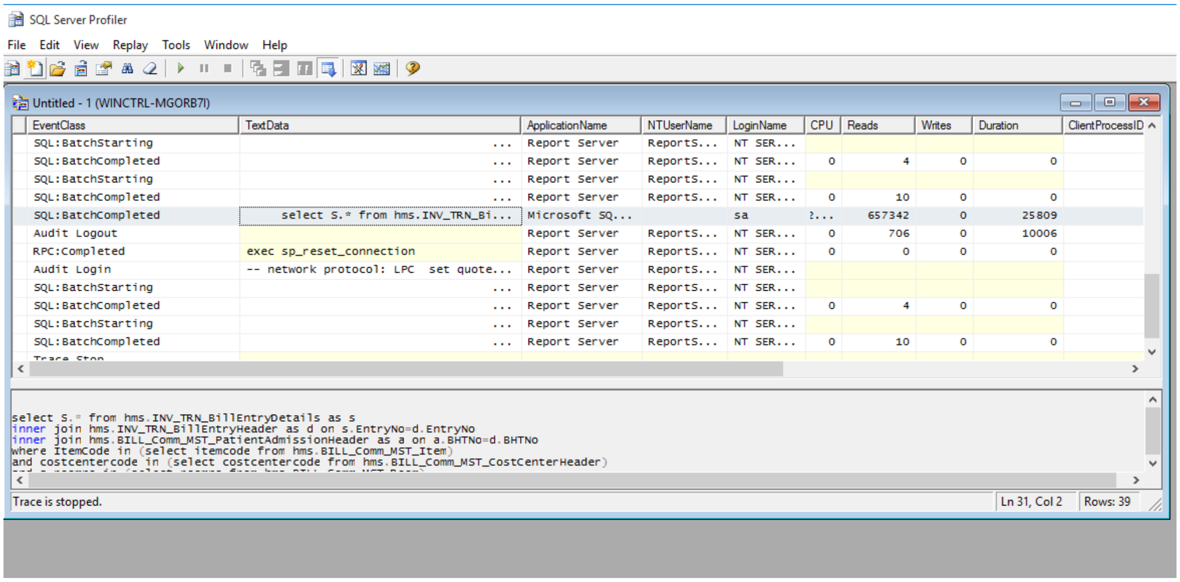


Figure 6.5 – SQL Profiler

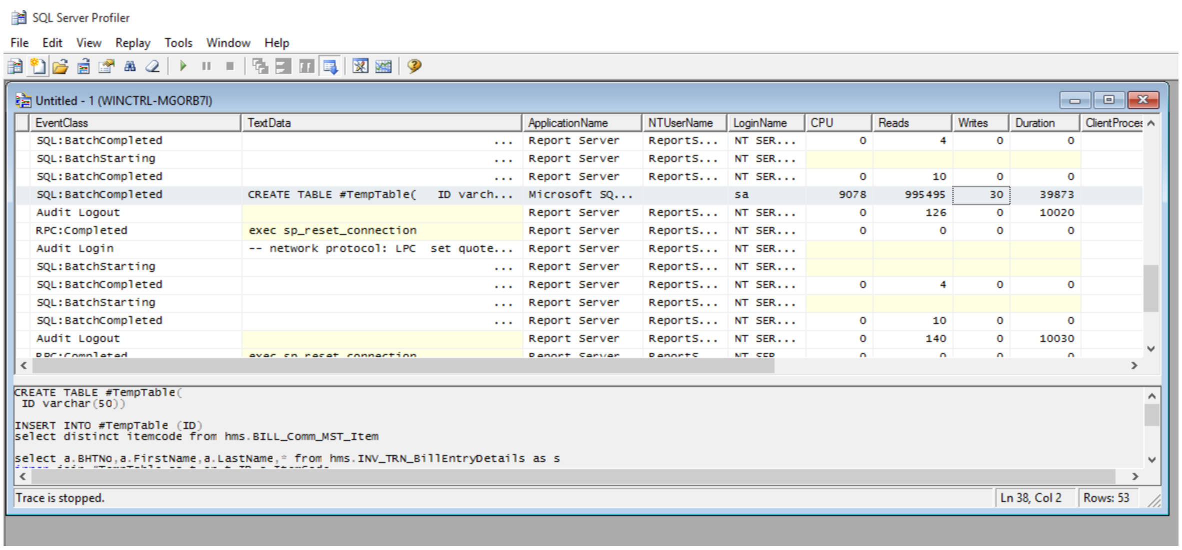


Figure 6.6 – SQL Profiler result

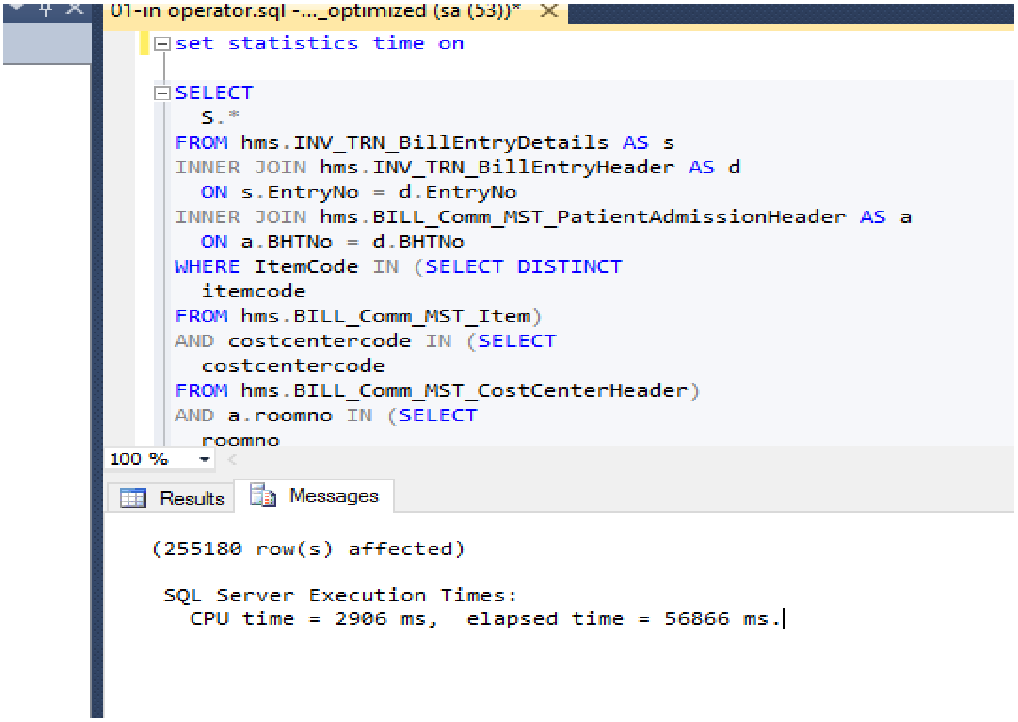


Figure 6.7 – SQL Server Execution time for Traditional query

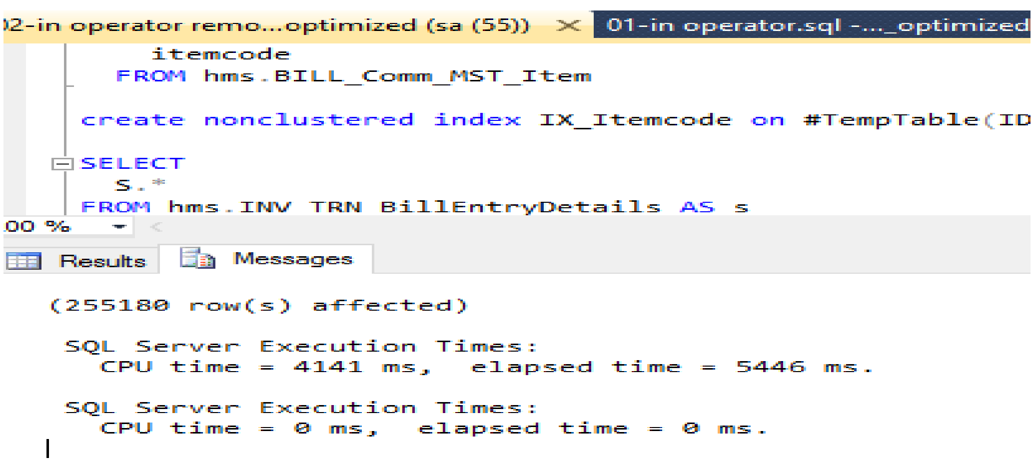


Figure 6.8 – SQL Server Execution time for our new proposed query

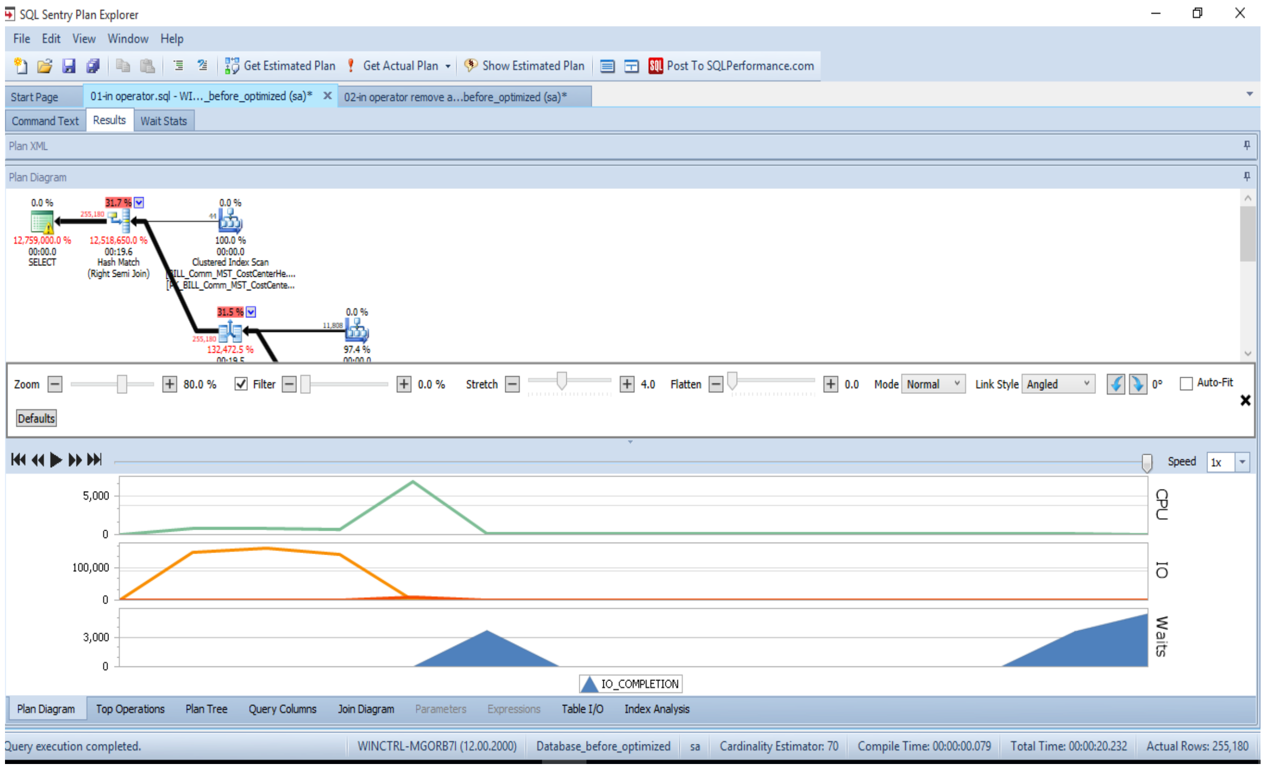


Figure 6.9 - Analyze by using Sentry Plan explore with IN

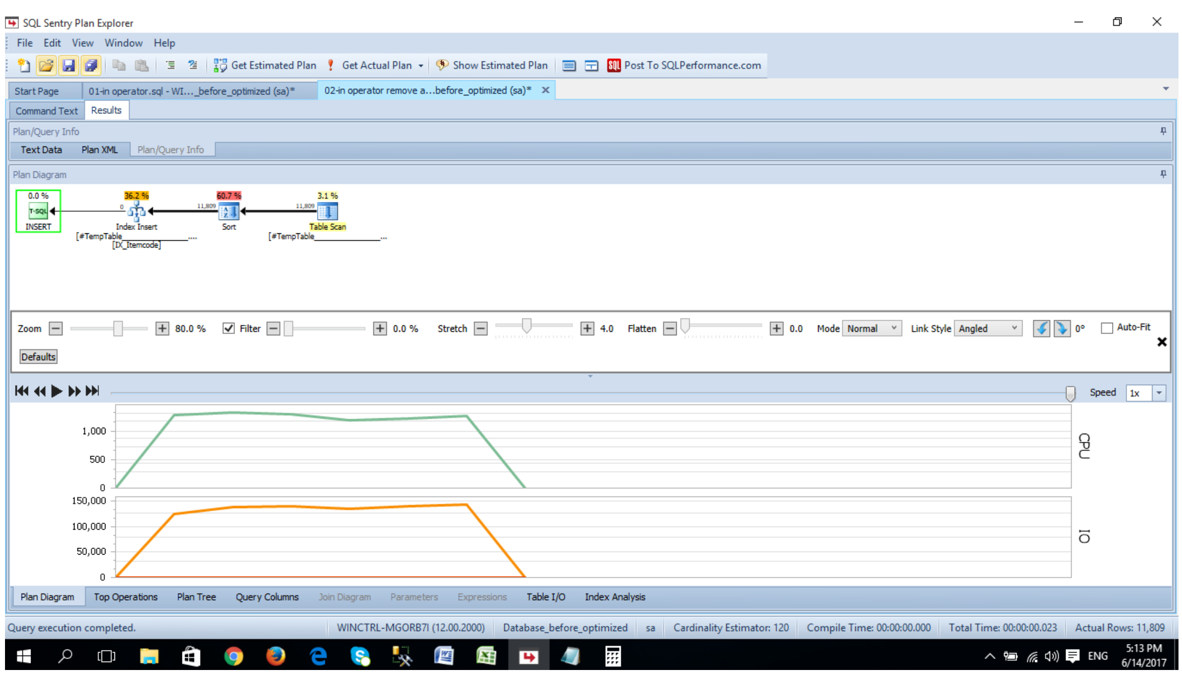


Figure 6.10 - Analyze by using Sentry Plan explore without IN

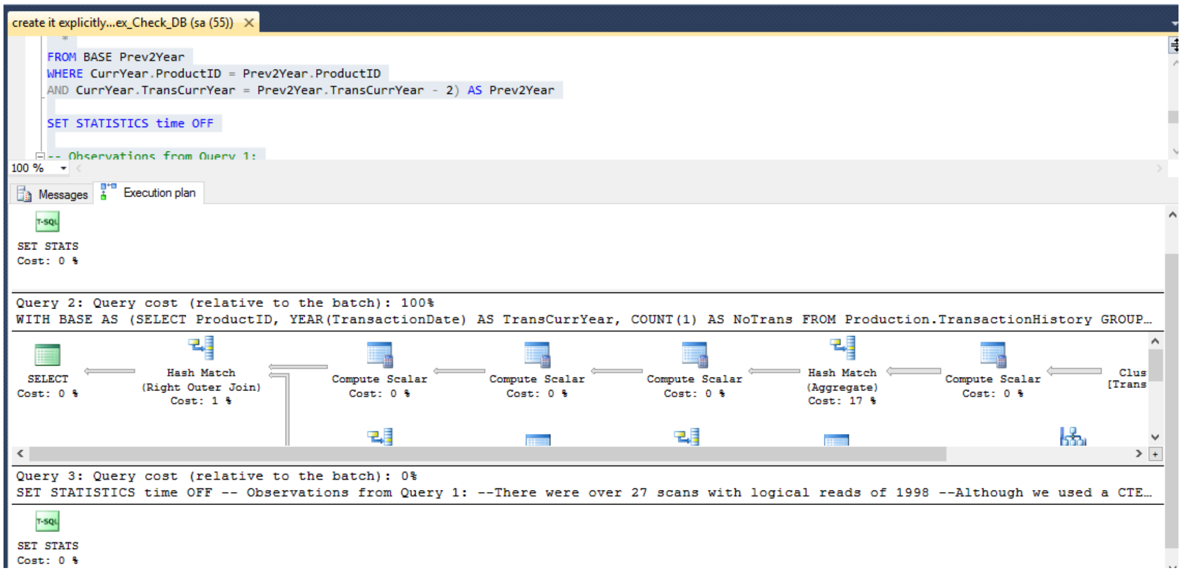


Figure 6.11 – Query cost with temp table

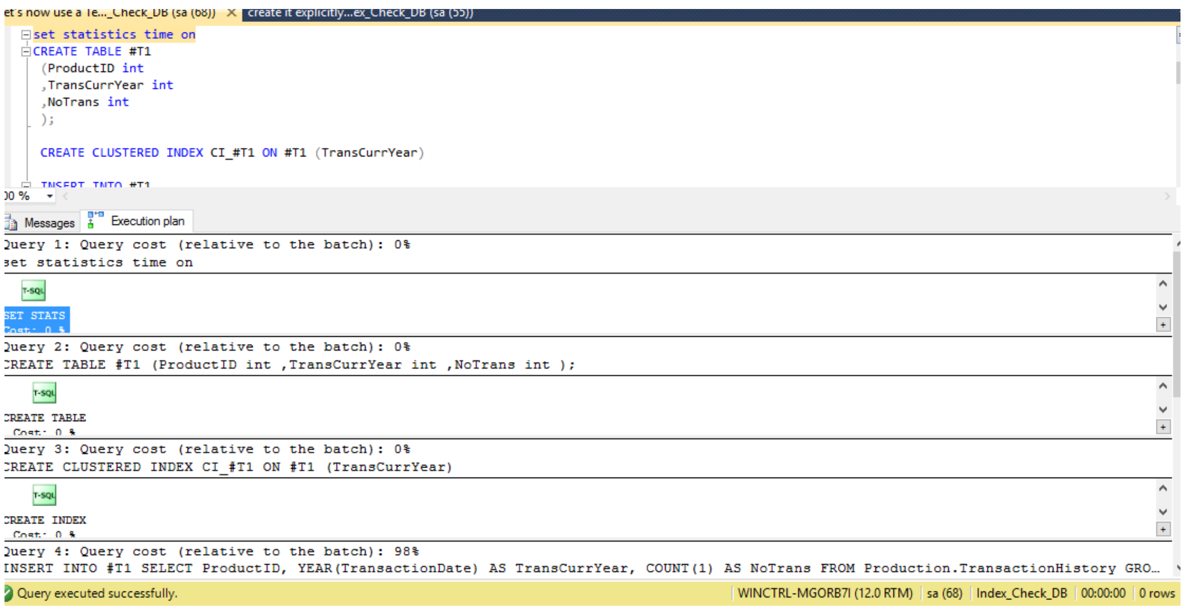


Figure 6.12 - Query cost with #temp table

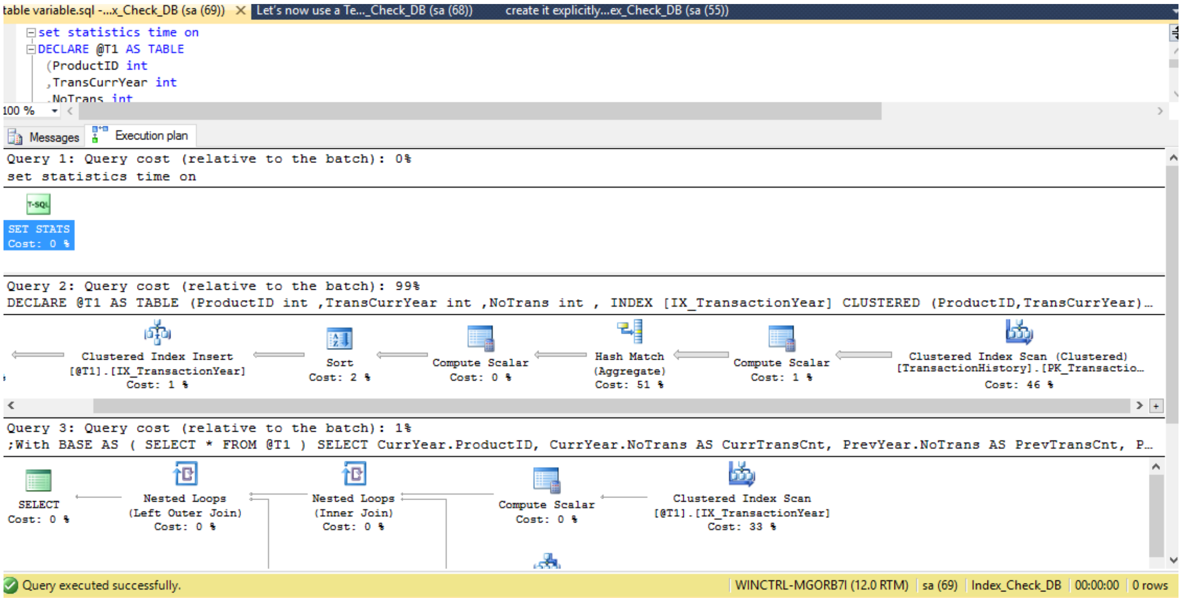
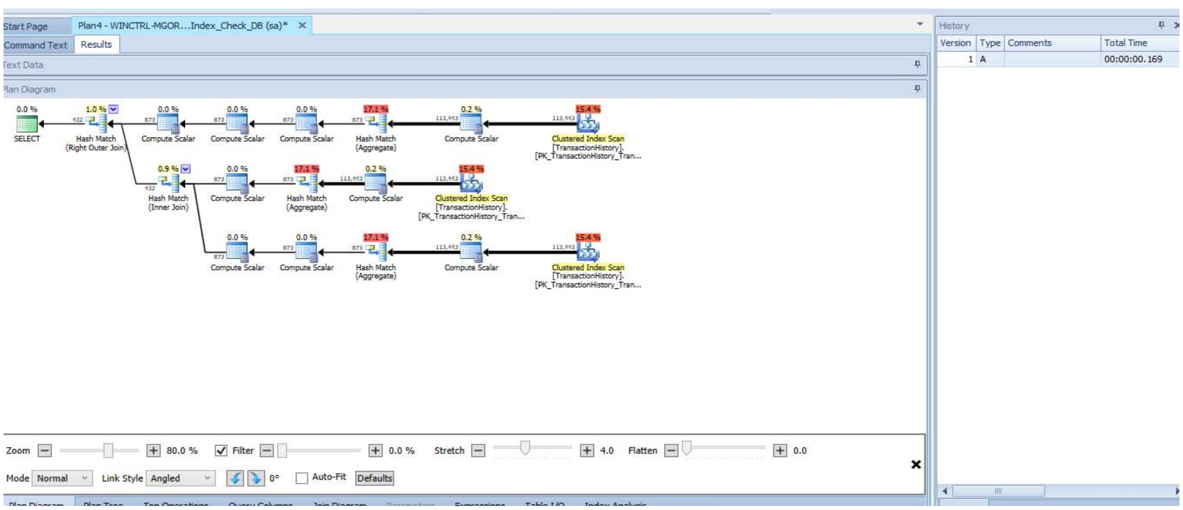
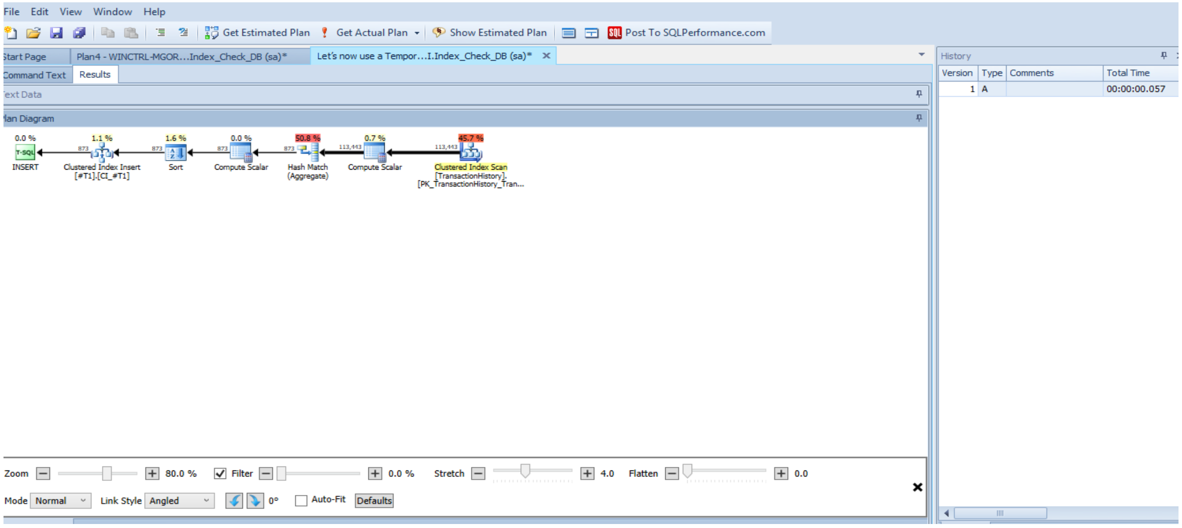


Figure 6.13 - Query cost with @temp table



6.14 - Sentry plan with #temp table



6.15 - Sentry plan with @temp table

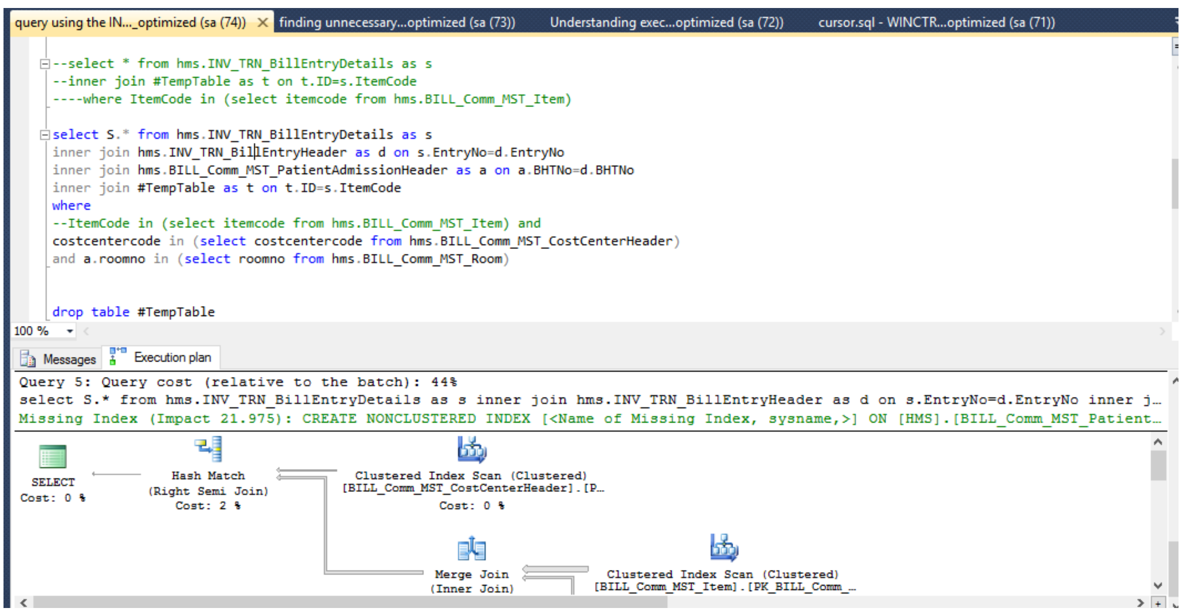


Figure 6.16 – How to find missing index

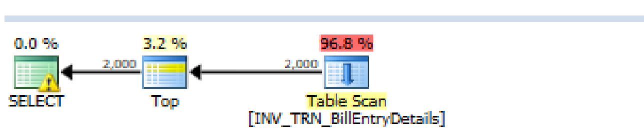


Figure 6.17 – Analyzed best practice IN and Where Clause.

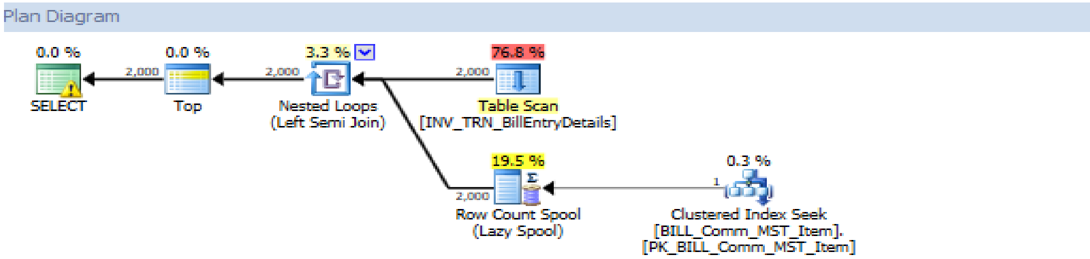


Figure 6.18 – Analyzed bad practice IN and Where Clause

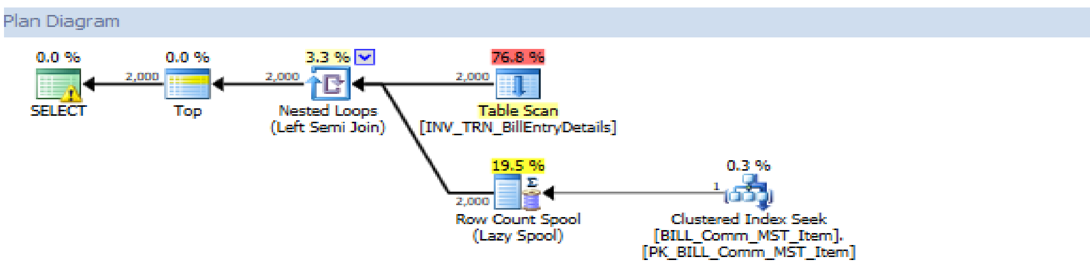


Figure 6.19 – Bad practice for IN and Where

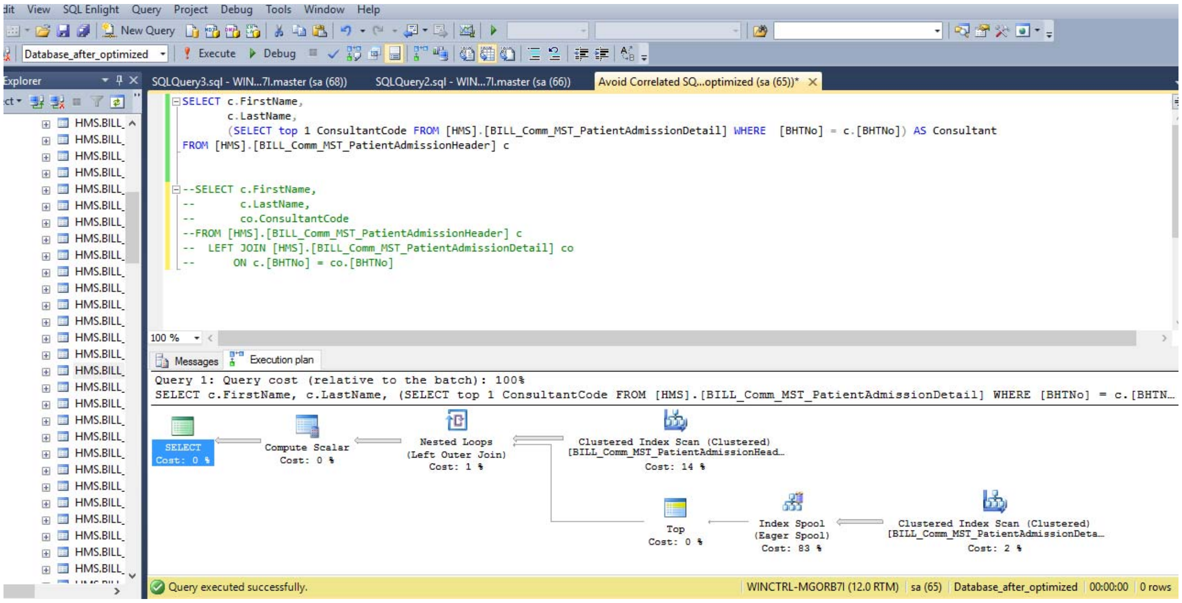


Figure 6.20 – QEP plan and Cost of Correlated SQL subqueries

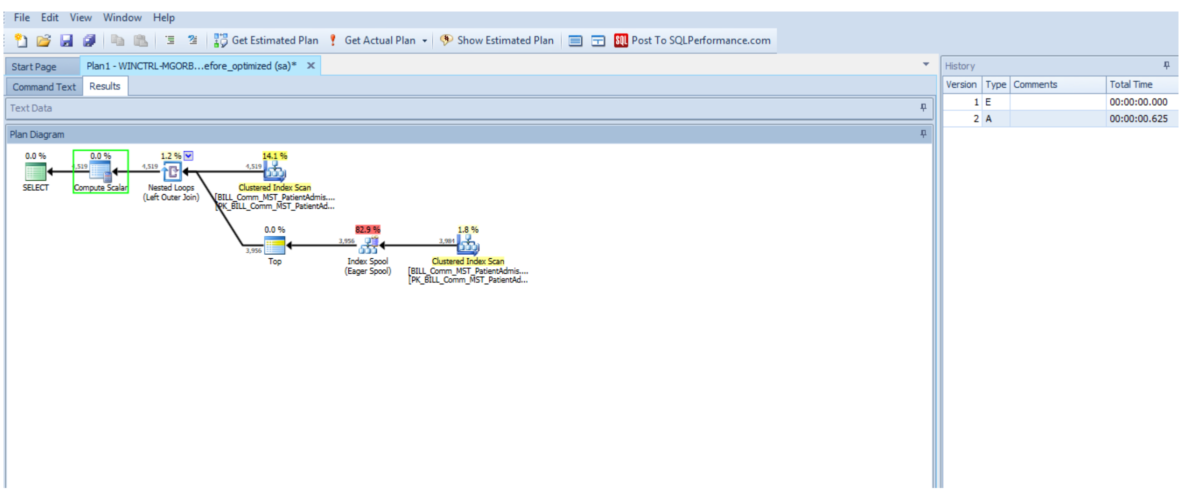


Figure 6.21- QEP plan and Cost of Correlated SQL subqueries in Sentry planner

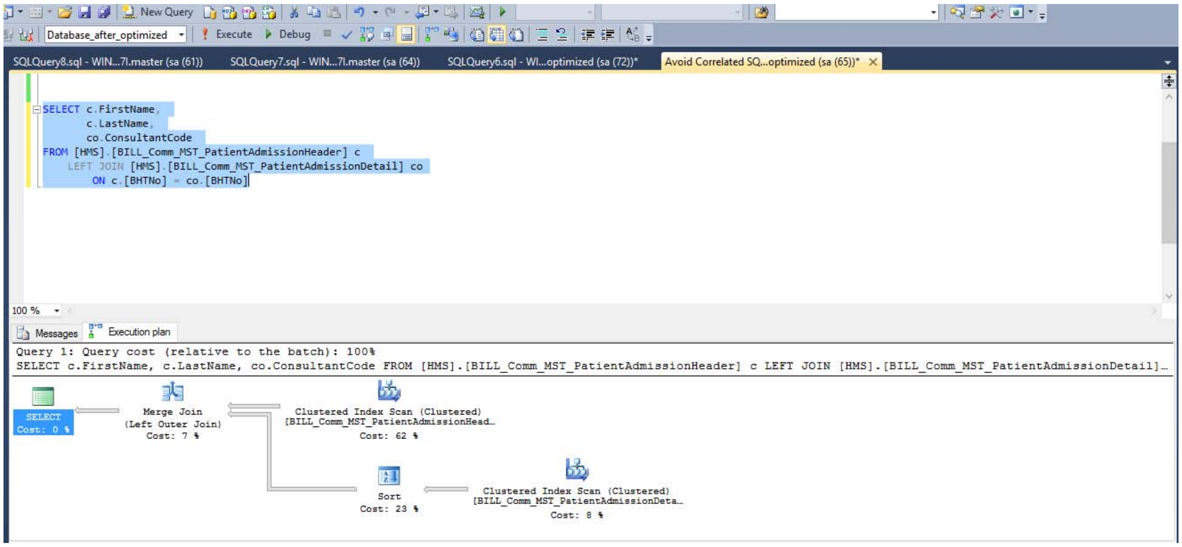


Figure 6.22- Our Query QEP plan and Cost of Correlated SQL subqueries

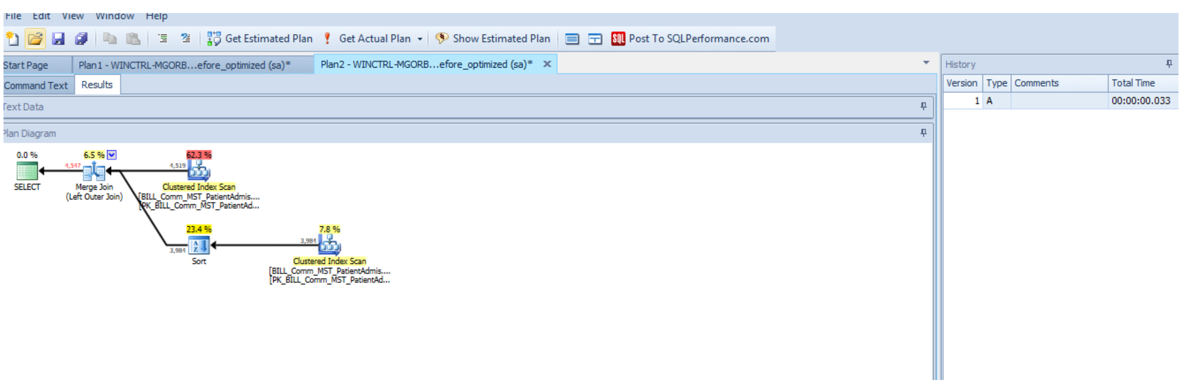


Figure 6.23- Our Query QEP plan and Cost of Correlated SQL subqueries in Sentry planner

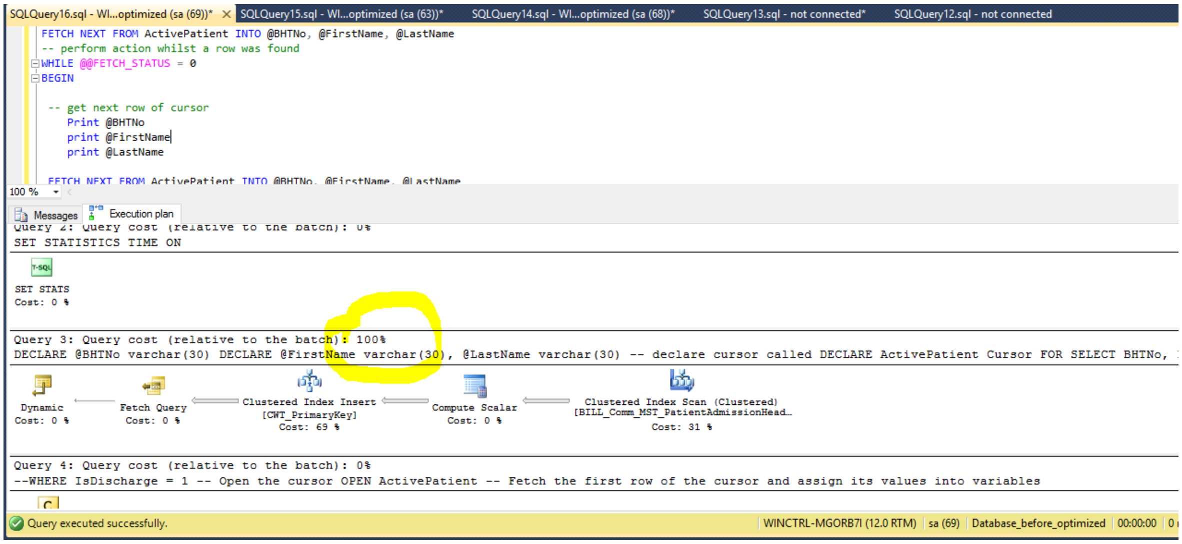


Figure 6.24 – QEP in Cusror

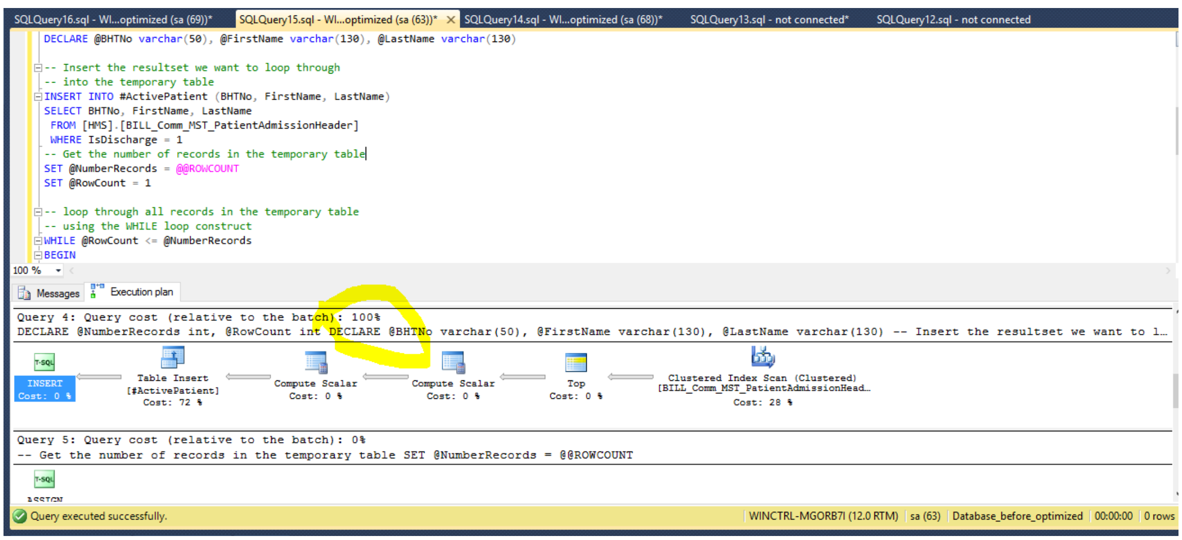


Figure 6.25 – Alternative solutinon QEP plan and query cost

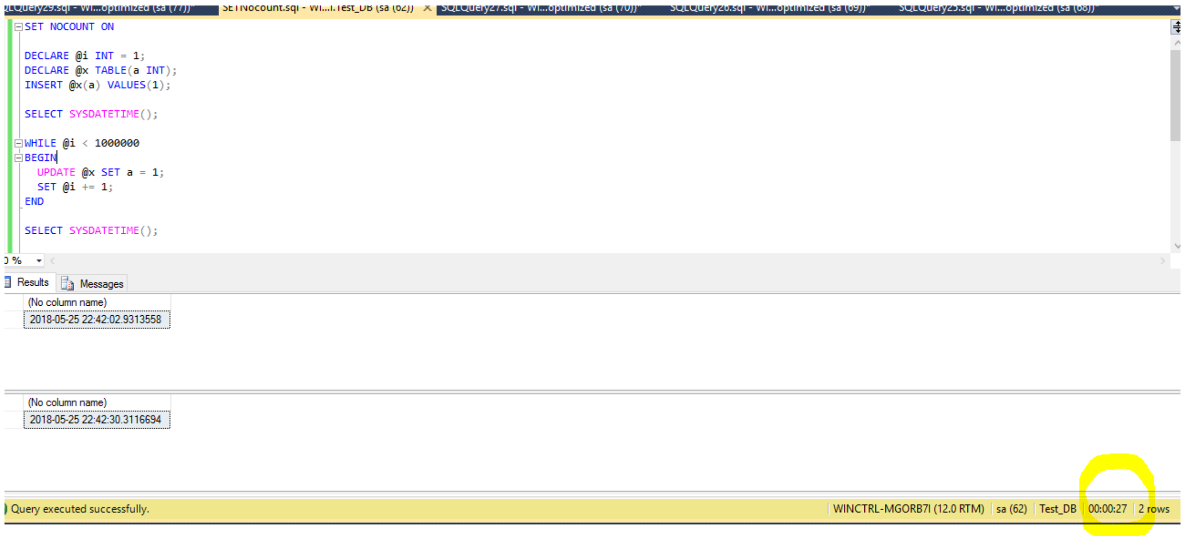


Figure 6.26 – Set no count on execution time

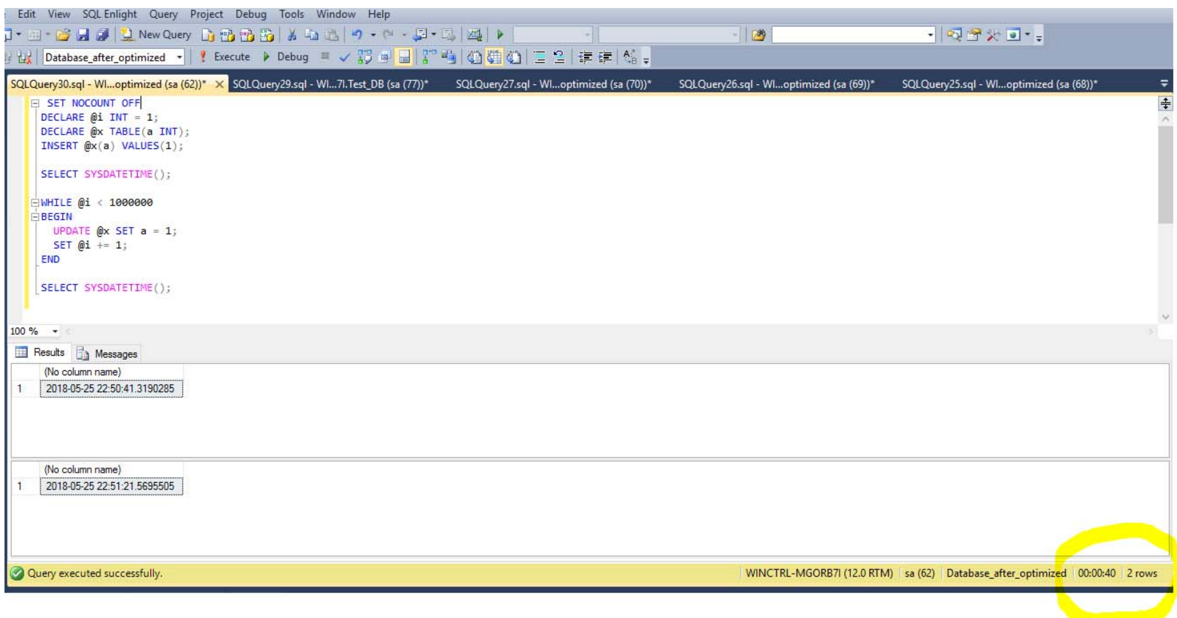


Figure 6.27 - 6.26 – Without no count execution time