**Review of Information Engineering and Applications**
*2014 Vol. 1, No. 2, 66-76.*
*ISSN(e): 2409-6539*
*ISSN(p): 2412-3676*
*DOI: 10.18488/journal.79/2014.1.2/79.2.66.76*


# ANALYTICAL REVIEW OF SQL SERVER OPTIMIZATION

**Gaurav Kumar[1]† --- Arun Sangwan[2]**

[1,2]*Department of Information Technology, Panipat Institute of Engineering & Technology, Haryana, India*

## ABSTRACT

*It is a complex task to optimize query as well as to validate the correctness and effectiveness of query optimizer. A query optimizer should estimate and compare the costs of executing a query using different execution strategies and should choose the strategy with the lower cost estimate. To fairly and realistically compare different strategies accurate cost estimation is required. This is a challenging task to measure quality of query optimization as modern query optimizers provide more advanced optimization strategies and adaptive techniques. This paper describes different ways to improve the performance of SQL Server queries, index optimization with occasional references to particular SQL code and how to achieve the best performance for the given tables and queries by giving some tips for query optimization in Microsoft SQL Server. The paper provides a detailed overview of query optimization, Optimization techniques, testing of optimization techniques that are used to validate the query optimizer of Microsoft's SQL Server and issues in query optimization testing.*

**Keywords:** DBMS, Estimation, Normalization, Optimization, SQL server, Testing.

## 1. INTRODUCTION

Optimizing database server means tuning the performance of individual query as some improper queries can negatively affect the performance of database server even the database is running on most powerful hardware. Even one bad query also called runaway query can cause serious performance problem for the database. A query expressed in high level query language such as SQL must first be scanned, parsed and validated. The *scanner* identifies the language tokens such as SQL keywords, attribute names and relation names in the query. *Parser* checks the query syntax to determine whether it is formulated according to the syntax rules of the query language. The query must be *validated*, by checking that all attribute and relation names are valid and semantically meaningful names in the schema of the particular database being queried [1].

Today's query optimizers provide highly complex functionality that is designed to serve a large variety of workloads, data sizes and usage patterns. They are the result of many years of



† Corresponding author

research and development, which has come at the cost of increased engineering complexity, specifically in validating correctness and measuring quality.

Query optimizers handle a practically infinite input space of declarative data queries (e.g. SQL), logical/physical schema and data. A simple listing of all possible input combinations is impractical and it is hard to predict or express expected behavior by grouping similar elements of the input space into equivalent classes. The query optimization process in itself is of high algorithmic complexity, and relies on inexact cost estimation models. Query optimizers needs to satisfy workloads and usage scenarios with a variety of different requirements and expectations, e.g. to optimize for throughout or for response time. Over time, the number of existing customers that need to be supported increases, a fact that introduces constraints in advancing query optimization technology without disturbing existing customer expectations. While new optimizations may improve query performance by orders of magnitude for some workloads, the same optimizations may cause performance degradation (or unnecessary overhead) to other workloads. For those reasons, a large part of the validation process of the query optimizer is meant to provide an understanding of the different tradeoffs and design choices in respect to their impact across different customer scenarios. At the same time, validation process needs to provide an assessment of degradation risk for code changes that may have a large impact across a large number of workload and query types.

## 2. SIGNIFICANCE OF OPTIMIZATION

When for a given table one or more indexes are available with some restrictions specified on them, optimization of this table access can involve various complex techniques that may cause a big impact on performance. However, the task of choosing a correct retrieval strategy can be difficult even in simple cases. Consider the simplest query:

select * from Students where marks >= ml;

with parameter m1 taking values 50 and 80, delivering all or no records in two different runs. The available indexes are used in several different ways during retrieval. If a certain retrieval order is requested, some indexes can provide this order. These types of index are called *order-needed indexes*. If an index contains all attributes needed for table restriction evaluation and for retrieval result delivery, the index scan alone can select and deliver all result records; these indexes are called *self-sufficient Indexes*. With several self-sufficient indexes present, the only optimization task to be resolved is to pick the one whose scan is the cheapest. Several fetch-needed indexes are used most optimally by their collective (joint) scan aiming at delivery of the shortest list of record IDS (RIDS) satisfying a cumulative fetch-needed index restriction. The RID list is built by intersecting/unionizing individual index RID lists according to the restriction AND/OR operations, and then is used for final fetches of data records. It is gradually becoming common knowledge in the industry that the static optimizer is helpless to consistently choose a correct scan strategy if host language variables are present or when good data distribution/ interaction estimation is not possible or too costly. Dynamic reevaluation of execution plan helps only partially since some estimation are impossible, or imprecise, or too costly when done at the

start retrieval time, and since data interaction uncertainty can often be irresolvable unless by the actual retrieval run. Query optimization usually seeks a good or not-bad execution strategy for a query instead of a truly optimal strategy [2].

## 3. FACTORS DEFINING QUERY OPTIMIZATION

The goal of query optimization is to produce efficient execution strategies for declarative queries. This involves the selection of an optimal execution plan out of a space of alternatives, while operating within a set of resource constraints. Depending on the optimization goals, the best-performing strategy could be optimized for response time, throughput, I/O, memory, or a combination of such goals. The different attributes of the query optimization process and the constraints within which it has to function make the tuning of the optimization choices a challenging problem [3]

### A. Optimization Time

The problem of finding the optimal join order in query optimization is NP-hard. Thus, in many cases the query optimizer has to cut its path aggressively through the search space and settle for a plan that is hopefully near to the theoretical optimum. The finding of the good place between optimization time/resources and plan performance along with the tuning of the different heuristics is a challenging engineering problem. New optimizations typically introduce new alternatives and extend the search space, often making necessary the tuning of such tradeoff decisions.

### B. Large Input Space and Multiple Paths

The significant power of query languages results in a practically infinite space of inputs to the query optimizer. For each query the query optimizer considers a large number of execution plans, which are code paths that need to be exercised and validated. The unbounded input space of possible queries along with the large number of alternative execution paths, generate a combinatorial explosion that makes exhaustive testing impossible. The selection of a representative set of test cases in order to achieve appropriate coverage of the input space can be a rather difficult task.

### C. Cardinality Estimation

A factor that complicates the validation of execution plan optimality is the confidence of the query optimizer on cardinality estimation. Query optimizers mainly rely on statistical information to make cardinality estimates, which is inherently inexact and it has known limitations as data and query patterns become more complex. Moreover, there are query constructs and data patterns that are not covered by the mathematical model used to estimate cardinalities. In such cases, query optimizers make rough estimations or alternative to simple heuristics. In the early days of SQL Server the majority of workloads consisted of prepared, single query-block statements, at this time query generator interfaces are very common, producing complex ad-hoc queries with characteristics that make cardinality estimation very challenging. Improvements in the estimation model, such as increasing the amount of detail captured by statistics and enhancing

the cardinality estimation algorithms, increase the quality of the plan selection process. However, such enhancements usually come with additional CPU cost and increased memory consumption.

### D. Cost Estimation

Cost models used by query optimizers, similarly to cardinality estimation models are also inexact and incomplete. Not all hardware characteristics, runtime conditions, and physical data layouts are modeled by the query optimizer. Although such design choices can obviously lead to reliability problems, there are often reasonable compromises chosen in order to avoid highly complex designs or to satisfy optimization time and memory constraints. The cost of executing a query includes the following components.

*Access cost to secondary storage:* This is the cost of searching for, reading and writing data blocks that reside on secondary storage, mainly on disk. The cost of searching for records in a file depends on the type of access structures on that file, such as ordering, hashing and primary or secondary indexes. In addition, factors such as whether the file blocks are allocated contiguously on the same disk cylinder or scattered on the disk affect the access cost.

*Storage cost:* This is the cost of storing any intermediate files that are generated by an execution strategy for the query.

*Computation cost:* This is the cost of performing in-memory operations on the data buffers during query execution. Such operations include searching for and sorting records, merging records for a join, and performing computations on field values.

*Memory usage cost:* This is the cost pertaining to the number of memory buffers needed during query execution.

*Communication cost:* This is the cost of shipping the query and its results from the database site to the site of terminal where the query originated. For large databases, the main emphasis is on minimizing the access cost to secondary storage. Simple cost functions ignore other factors and compare different query execution strategies in terms of the number of block transfers between disk and main memory. For smaller databases, where most of the data in the files involved in the query can be completely stored in memory, the importance is on minimizing computation cost [4].

### E. Hit and Trial for Optimization

Occasionally, the query optimizer can produce nearly optimal plans, even in presence of large estimation errors and estimation guesses. They can be the result of lucky combinations of two or more inaccuracies canceling each other. Additionally, applications may be built in a way that they depend on specific limitations of the optimizer's model. It can also happen by chance, when the developer continuously tries different ways to develop their application until the desired performance is achieved because a specific combination of events was hit. Therefore, applications and any tests based on such applications that rely on overfitting may experience unpredictable changes when the conditions on which they depend changes [5].

### F. Self-Tuning Techniques

The use of self-tuning techniques is to simplify the tasks of system administration and to diminish the effect of estimation errors, themselves generate tuning and validation challenges. For example, SQL Server's policy for automatically updating statistics can be too effective for certain customer scenarios, resulting in unnecessary CPU and I/O consumption and for others it can be too lazy, resulting in inaccurate cost estimations. Advanced techniques used to decrease the cost model inaccuracies and limitations, e.g. the use of execution feedback to correct cardinality estimates, or the implementation of corrective actions during execution time, introduce similar tradeoffs and tuning problems [1, 6]. Case study given under section-V reveals as Data volume (per query) increases Query Set (Diversity and Complexity) also increases as shown in figure 1. For the lower portion of both parameters, we have OLTP, for the middle portion, line of business and custom apps, for the top most region DSS comes into picture.

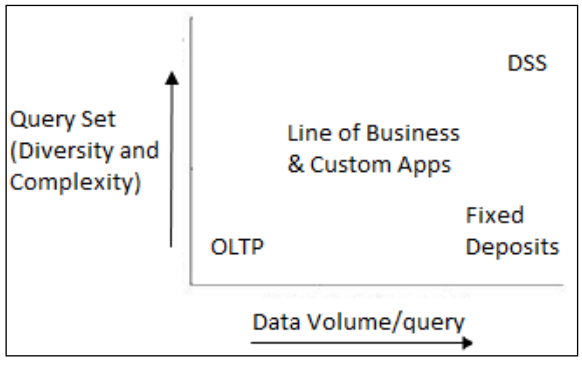


**Figure-1.** Database application space for different workloads

Workloads on the left-bottom area of the space are typical Online Transaction Processing (OLTP) workloads, which include simple, often parameterized queries. Such workloads require short optimization times and they benefit from plan reuse. Workloads on the right side of the space may include Decision Support System (DSS) or data warehousing applications, which usually consist of complex queries over large data sets. DSS workloads have higher tolerance for longer optimization times and thus more advanced optimization techniques can be used for those. The middle area of the application space contains a larger variety of applications. Those applications can contain a mixture of simple and more complex queries, which can be either short or long running. Changes in the optimization process affect queries from different parts of the application space in different ways, either because of shifts in existing tradeoffs and policies, or because of issues related to over fitting.

## 4. QUERY OPTIMIZATION TESTING TECHNIQUES

Validation of a software system can be divided into two categories [7]:

- Those that aim to simulate usage scenarios and verify that the end result of a system operation satisfies the customer's requirements.

- Those that aim to implement specific subcomponents and code paths to ensure that they function according to system design. Test cases typically aim to validate the correctness of query results, measure query and optimization performance, or verify that specific optimization functionality works as expected.

Examples of testing techniques from these two categories, used to validate SQL Server's query optimizer are given below:

### A. Correctness Testing

The query optimization process should produce execution plans which are correct, i.e. plans that will produce correct results when executed. Correctness can be validated up to some extent logically, by verifying that the various query tree transformations result in semantically correct alternatives. Additionally, it can be validated by executing various alternative execution plans using plan enumeration techniques and then comparing their results with each. Another common practice is to run playbacks. *Playbacks* are SQL traces collected from customers also used to verify correctness against a reference implementation.

### B. Using Test/Query Generators through Stochastic Testing

Steps performed in test engineering to identify the space of different inputs to the system under test and then to define test scenarios that use the system using instances selected from these equivalence classes. As mentioned earlier, the input space for a query optimizer is multi dimensional and very large. Different server configurations and query execution settings introduce additional dimensions to the input space. An effective testing technique for tackling large input spaces is to use test/query generators that can generate huge sets of test cases. The generation process can be random. Such techniques have been very effective in testing SQL Server.

### C. Performance Baselines

The task of validating changes in the query optimizer's logic can be difficult. A typical approach is to evaluate changes by measuring query performance against a known baseline. Industry-standard benchmarks, covers only a small part of SQL Server's functionality. Therefore, our testing process includes a wider set of benchmarks that cover a larger variety of scenarios and product features. Normally, those benchmarks consist of test cases based on real customer scenarios. They are used for performance comparisons with a previous product release or with an alternative implementation [1].

### D. Optimization Quality Scorecards

Although optimizations of time and query performance are good measures of plan choice effectiveness, they are not sufficient for an in-depth understanding of the impact of changes to the optimization process. New exploration rules may expand the search space with valuable new alternatives but at the cost of increased memory consumption, which may cause performance blockage on a loaded server. In order to gain as much insight as possible into the impact of changes, our process includes a variety of metrics in addition to query and optimization

performance. Examples of such metrics are the amount of optimization memory, cardinality estimation errors, execution plan size, search space size etc.

## 5. CASE STUDY

Large scale stochastic testing i.e. using Test/Query Generator has been effective in extending the coverage provided by regular tests. Query generators can be driven towards exploring the space of queries and query plans much further than what can be achieved by other types of testing. The combination of stochastic testing with self-checking mechanisms in the code has been very effective in detecting irregularities in internal data structures that would result to incorrect query results. In past releases of SQL Server, the performance tuning of the database engine was done towards the final phases of product development and hence regressions in optimization time were detected late. The establishment and regular monitoring of the query optimization scorecard during the development cycle has allowed us to be proactive in identifying regressions as compared to the past. Early detection allows more time to tune the optimization heuristics towards an appropriate balance between plan efficiency and optimization time. The combination of stochastic testing techniques and benchmarks based on realistic customer workloads has been very helpful for the development of some features of SQL Server [4, 8-10].

### A. The Importance of a Reliable Benchmark

Given the statistical nature of optimization quality, it is essential that the benchmark used for making quality measurements is reliable and balanced. During the SQL Server 2000 release, a testing practice followed was to add a new test case every time when any of the customers and partners would experience a performance regression. Adding regression tests in order to prevent future reoccurrences of code defects is a standard practice in test engineering. The regular application of the above process introduced a large number of regressions tests in our benchmark. During the development cycle, there were times when our benchmark was heavily affected by the performance of those tests. In some cases, valid improvements in the cost model would cause performance regressions. The performance of those regression tests was often unpredictable, and it could drop enough to overshadow the performance gains in other tests. At that point, it became clear that the practice of continuously extending the benchmark with various regression tests was problematic. Today, benchmarks are developed that are more complete and balanced in terms of application type but also in terms of their conformance to the optimizer's model. If there is a specific application with which there is an issue in the past and tracking of its performance has to be done, a subset of that application workload is added into the benchmark that helps to understand the impact of a code change on multiple queries from that application.

### B. Improvement on the Measurement

Increase in compilation time during the development cycle had a significant impact across a large part of the overall benchmark, while the effect of more advanced optimizations only appeared in smaller areas. The hardware configuration used for executing the benchmark can affect the making of tuning decisions in similar ways. For this reason, the different scenarios and

hardware configurations need to be defined and maintained in a way that represents the product's goals as carefully as possible.

### C. Defect due to Regression

Legitimate code changes can result in slower execution for some queries. It is very important that the engineering team agrees on a well defined process on how to treat such issues, both internally as well as externally when communicating with customers. Fixing regressions in ways that do not conform to the optimizer's model and assumptions, results in code health issues and architectural problems. For this reason it is very important to have a clear definition of the optimizer's model. At the same time, every decision needs to take into account the expected impact on customer experience.

### D. Design for Testability

Back to the past four to five years of product development several times addition of testability features into the query optimizer is done in order to expose internal run-time information and add control-flow mechanisms for white-box testing. Designing new features with testability in mind is a task much easier. This helps in clarifying the interfaces and contracts between different subcomponents and the resulting test cases ensure that they remain valid during future development.

## 6. RESULT ANALYSIS

Query optimization has a very big impact on the performance of a DBMS and it continuously evolves with new optimization strategies. Also, with every addition of a new rule, the search space expands and the number of possible plan choices increases accordingly. Although query optimization testing is described with focus on correctness and optimality, another interesting dimension of the query optimization quality is the concept of performance predictability. More work is needed on defining, measuring and validating predictability for different classes of applications. The validation process and testing techniques will continue to evolve along with the evolution of the optimization technology and product goals. The techniques described in this paper allow basic validation and also provide insight regarding the impact of code changes in the optimization process.

## 7. SUMMARY AND CONCLUSIONS

Here, we are summarizing as well as concluding some steps for SQL Server Optimization while designing tables [9-13].

### A. Normalize the Table in 3rd Normal Form

A table is in third normal form (3NF) if it is in 2nd Normal form (2NF) and if it does not contain transitive dependencies. The normalization is used to reduce the total amount of redundant data in the database. The less data, less work SQL Server has to perform, speeding its performance.

**B. Denormalize Tables from 4th/5th Normal Forms to the 3rd Normal Form**

Normalization to the 4th and 5th normal forms can result in some performance degradation, especially when one needs to perform many joins against several tables. It may be necessary to de-normalize the tables to prevent performance degradation.

**C. Partition Horizontally Very Large Tables into Current and Archives Versions**

The less space used, the smaller the table, the less work SQL Server has to perform to evaluate such queries e.g., if one needs to query only data for the current year in the daily work, and need all the data only once per month for the monthly report we create two tables: one with the current year's data and one with the old data.

**D. Create the Table's Columns as Narrow As Possible**

This can reduce the table's size and improve performance of queries as well as some maintenance tasks such as backup, restore etc.

**E. Try to Reduce Number of Columns in a Table**

The fewer the number of columns in a table, the less space the table will use, since more rows will fit on a single data page, and less I/O overhead will be required to access the table's data.

**F. Use Constraints Instead of Triggers, Rules, and Defaults Whenever Possible**

Constraints are much more efficient than triggers and can boost performance. Constraints are more consistent and reliable in comparison to triggers, rules and defaults, because you can make errors when you write your own code to perform the same actions as the constraints.

**G. Use Tinyint Data Type if Integer Data is from 0 through 255.**

The columns with tinyint data type use only one byte to store their values, in comparison with two bytes, four bytes and eight bytes used to store the columns with smallint, int and bigint data types accordingly. For example, if table has to be designed for a small company with 5-7 departments, Departments table can be created with the DepartmentIDtinyint column to store the unique number of each department.

**H. Use Smallint Data Type if Integer Data from -32,768 through 32,767.**

The columns with smallint data type use only two bytes to store their values, in comparison with four bytes and eight bytes used to store the columns with int and bigint data types respectively e.g. if a table has to be designed for a company with hundreds of employees, employee table can be created with the EmployeeIDsmallint column to store the unique number of each employee.

**I. Use Int Data Type if Integer Data is from -2,147,483,648 to 2,147,483,647.**

The columns with int data type use only four bytes to store their values, in comparison with eight bytes used to store the columns with bigint data types. For example, to design tables for a library with more than 32,767 books, create a books table with a BookIDint column to store the unique number of each book.

**J. Use Smallmoney Data Type Instead of Money Data Type to Store Monetary Data Values from 214,748.3648 through 214,748.3647.**

The columns with smallmoney data type use only four bytes to store their values, in comparison with eight bytes used to store the columns with money data types. For example, if monthly employee payments need to be stored, it might be possible to use a column with the smallmoney data type instead of money data type.

**K. Use Smalldatetime Data Type Instead of Datetime Data Type, to Store The Date and Time from January 1, 1900 through June 6, 2079.**

The columns with smalldatetime data type use only four bytes to store their values, in comparison with eight bytes used to store the columns with datetime data types. For example, if employee's hire date needs to be stored, column with the smalldatetime data type can be used instead of datetime data type.

**L. Use Varchar/Nvarchar Columns Instead of Text/Ntext Columns.**

Because SQL Server stores text/ntext columns on the Text/Image pages separately from the other data, stored on the Data pages, it can take more time to get the text/ntext values.

**M. Use Char/Varchar Columns Instead of Nchar/Nvarcharif No Need to Store Unicode Data.**

The char/varchar value uses only one byte to store one character; the nchar/nvarchar value uses two bytes to store one character, so the char/varchar columns use two times less space to store data in comparison with nchar/nvarchar columns.

**N. Use Cascading Referential Integrity Constraints Instead of Triggers whenever Possible While Working with SQL Server 2000.**

For example, if one needs to make cascading deletes or updates, specify the on delete or on update clause in the references clause of the create table or alter table statements. The cascading referential integrity constraints are much more efficient than triggers and can boost performance.

**O. Avoid Too Many Indexes**

If too many indexes are created on a table, sql server engine will take longer to insert a record in a table since index processing must be done for each record i.e. inserted, updated or deleted. Thus while indexes speeds up data retrieval, data insertion slows down greatly. A balance must be maintained such that only columns that are frequently used for data retrieval i.e. querying the table are indexed [12].

**P. Avoid Clusters for the Queries That Reference One of the Tables in the Cluster**

Clusters are used to store data from different tables in the same physical data blocks. They are appropriate to use if the records from those tables are frequently queried together. Clusters may have a negative performance impact on the data manipulation transactions and on queries that only references one of the tables in the cluster [12].

## REFERENCES


[1] Leo Giakoumakis, "Cesar Galindo-Legaria – testing SQL server's query optimizer: Challenges, techniques and experiences," *IEEE Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, pp. 1-8, 2008.

[2] Gennady Antoshenkov, "Dynamic query optimization in Rdb/VMS," in *Proc. of IEEE 9th International Conference on Data Engineering*, 1993, pp. 538-547.

[3] SQL Server Optimization Rips for Designing Tables by Alexander Chigrik http://www.databasejournal.com/features/article.php/3593466/MS-SQL-Series.htm.

[4] Navathe, Elmarsi, Somayajulu, and Gupta, *Fundamentals of database systems*, 6th ed.: Pearson Education, 2010.

[5] L. Kollar, "Statistics used by the query optimizer in microsoft SQL server 2000." Available: http://technet.microsoft.com/en-us/library/aa902688(v=sql.80).aspx.

[6] Ivan Bayross, *SQL, PL/SQL the programming language of Oracle*: BPB Publications, 2010.

[7] Z. Qiang, L. Per-Ake, Z. DeQiang, and L. Per-Ake, "Establishing a fuzzy cost model for query optimization in a multidatabase system," *IEEE*, pp. 1060-3425194, 1994.

[8] Introducing SQL Trace, Available: http://technet.microsoft.com/en-us/library/ms191006.aspx.

[9] K. Lubor, "Microsoft SQL 2000 technical articles, statistics used by the query optimizer in microsoft SQL server 2000." Available: http://msdn2.microsoft.com/en-s/library/aa902688(SQL.80).aspx.

[10] MSDN Blogs, "Tips, tricks, and advice from the SQL server query optimization team." Available: http://blogs.msdn.com/queryoptteam/default.aspx.

[11] Helios Solutions, "SQL query optimization and performance tuning." Available: http://blog.heliossolutions.in/sql-query-optimization-and-performance-tuning/.

[12] SQL Server Optimization Tips http://santhoshgudise.weebly.com/uploads/8/5/4/7/8547208/sql_server_optimization_tips-1.doc.

[13] "Optimization tips for designing tables." Available: http://www.mssqlcity.com/Tips/design_optimization.htm.