Testing SQL Server's Query Optimizer: Challenges, Techniques and Experiences

Leo Giakoumakis, Cesar Galindo-Legaria
Microsoft SQL Server
{leogia,cesarg}@microsoft.com

Abstract

Query optimization is an inherently complex problem, and validating the correctness and effectiveness of a query optimizer can be a task of comparable complexity. The overall process of measuring query optimization quality becomes increasingly challenging as modern query optimizers provide more advanced optimization strategies and adaptive techniques. In this paper we present a practitioner's account of query optimization testing. We discuss some of the unique issues in testing a query optimizer, and we provide a high-level overview of the testing techniques used to validate the query optimizer of Microsoft's SQL Server. We offer our experiences and discuss a few ongoing challenges, which we hope can inspire additional research in the area of query optimization and DBMS testing.

1 Introduction

Today's query optimizers provide highly sophisticated functionality that is designed to serve a large variety of workloads, data sizes and usage patterns. They are the result of many years of research and development, which has come at the cost of increased engineering complexity, specifically in validating correctness and measuring quality. There are several unique characteristics that make query optimizers exceptionally complex systems to validate, more so than most other software systems.

Query optimizers handle a practically infinite input space of declarative data queries (e.g. SQL, XQuery), logical/physical schema and data. A simple enumeration of all possible input combinations is unfeasible and it is hard to predict or extrapolate expected behavior by grouping similar elements of the input space into equivalence classes. The query optimization process itself is of high algorithmic complexity, and relies on inexact cost estimation models. Moreover, query optimizers ought 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 regressions (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,

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

the validation process needs to provide an assessment of regression risk for code changes that may have a large impact across a large number of workload and query types.

2 Key Challenges

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 and tradeoffs a challenging problem.

Large input space and multiple paths: The expressive 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.

Optimization time: The problem of finding the optimal join order in query optimization is NP-hard [8, 4]. 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 infeasibility of exhaustive search introduces a tradeoff between optimization time and plan performance. The finding of the ”sweet spot” 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.

Cardinality estimation: A factor that complicates the validation of execution plan optimality is the reliance 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 [9]. 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 crude estimations or resort to simple heuristics [12]. While 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. Inevitably, testing the plan selection functionality of the query optimizer depends on the accuracy of the cardinality estimation. 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 typically come with additional CPU cost and increased memory consumption.

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.

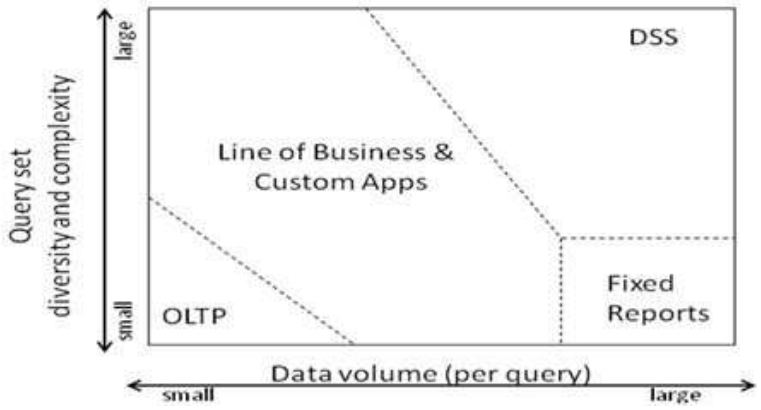

Figure 1: An illustration of the database application space

"Two wrongs can make a right" and Overfitting: 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 rely on specific limitations of the optimizer's model. Such *overfitting* of the application's behavior around the limitations of the optimizer's model can happen intentionally, when a developer has knowledge of specific system idiosyncrasies and develops their application in a way that depends on those idiosyncrasies. It can also happen unintentionally, 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). Of course there are no guarantees that system idiosyncrasies and lucky combinations of events would remain constant between product releases or over changes during the application lifecycle. Therefore, applications (and any tests based on such applications) that rely on overfitting may experience unpredictable regressions when the conditions on which they depend change.

Adaptive optimization and self-tuning techniques: The use of self-tuning techniques to simplify the tasks of system administration and to mitigate the effect of estimation errors, themselves generate tuning and validation challenges. For example, SQL Server's policy for automatically updating statistics [10], can be too eager 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 mitigate the cost model inaccuracies and limitations, for example the use of execution feedback to correct cardinality estimates [14], or the implementation of corrective actions during execution time, introduce similar tradeoffs and tuning problems.

Optimization quality is a problem of statistical nature: SQL Server's customer base includes a variety of workload types with varying performance requirements. Figure 1 illustrates the space of 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. They typically contain ad-hoc queries, generated by query-generator tools/interfaces. The middle area of the application space contains a larger variety of applications that cannot be characterized as simply as the ones above. 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

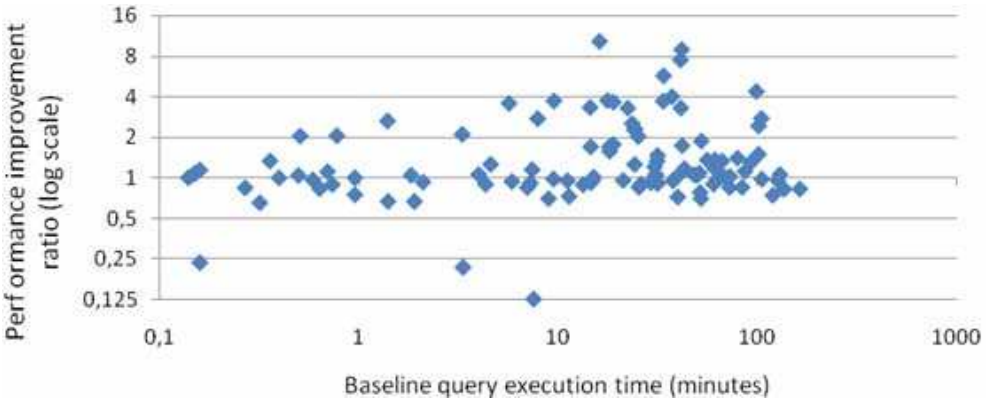

Figure 2: Performance impact of new optimizations

space in different ways, either because of shifts in existing tradeoffs and policies, or because of issues related to overfitting. Inevitably, that makes the measurement of optimization quality a problem of statistical nature. As an example, Figure 2 illustrates the results of experiments with new query optimizer features on a realistic query workload. In most cases the new features provide significant performance gains (especially for long-running queries), but they cause regressions for some parts of the workload (all points below 1 in Figure 2). Some short-running queries were affected by increases in compilation time, while a few others regressed because of a suboptimal plan choice or lack of tuning for the specific hardware used in the experiment. While in this example the benefits of the new functionality outweigh the performance regressions, there have been other cases where it was more difficult to make a judgment about the advantages vs. the disadvantages of introducing a particular new feature.

3 Query Optimization Testing Techniques

The practices of validating a software system can be typically divided in two categories: a) those that aim to simulate usage scenarios and verify that the end result of a system operation satisfies the customer's requirements and b) those that aim to exercise 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. We provide some examples of testing techniques from these two categories, used to validate SQL Server's query optimizer.

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 [15] and then comparing their results with each other and/or with a reference implementation (that is typically a previous product release or a different database product). Another common practice is to run *playbacks*. Playbacks are SQL traces [1] collected from customers also used to verify correctness against a reference implementation.

Large-scale stochastic testing: Typical steps in test engineering are to identify the space of different inputs to the system under test, to recognize the equivalence classes within the input space, and then to define test scenarios that exercise the system using instances selected from these equivalence classes. As mentioned earlier, the input space for a query optimizer is multidimensional 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 massive sets of test cases. The generation process can be random or can be guided towards covering specific areas of the input space or certain areas of the product. Such techniques have been very effective in testing SQL Server [7, 13, 15].

Performance baselines: The task of validating changes in the query optimizer's plan choice logic in presence of the various engineering tradeoffs can become rather difficult. A typical approach is to evaluate changes by measuring query performance against a known baseline. Industry-standard benchmarks, like TPCH [3] cover only a small part of SQL Server's functionality and contain very well-behaved data distributions. 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.

Optimization quality scorecards: Although optimization 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. Improvements in the optimizer's model will not always result in improvements in plan choice (for the queries included in a benchmark), but this should not necessarily mean that they have no value overall. On the other hand, new exploration rules may expand the search space with valuable new alternatives but at the cost of increased memory consumption, which may cause performance bottlenecks 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, and others. These metrics can be collected across the whole set of queries included in our various benchmarks, across an individual benchmark and across segments of the application taxonomy, providing a number of different *optimization quality scorecards*.

4 Experiences and Lessons Learned

The testing techniques mentioned in this article target different classes of defects. We briefly discuss a few representative classes here, and how they correspond to the various testing techniques. We then continue with a summary of some of the lessons learned during our efforts.

Large scale stochastic testing has been effective in extending the coverage provided by regular tests. Specifically, it has contributed in eliminating *MEMO cycles* and incorrect results. MEMO cycles can occur when defects in the implementation of a set of transformation rules allow cycles to be generated in the recursive group structures. We refer the reader to [15] for an explanation of SQL Server's MEMO structure. SQL Server's code contains self-verification mechanisms to detect cycles and other inconsistencies in the MEMO structure. Therefore, the discovery of such a defect is an exercise of generating the appropriate test case. 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. A case in point is the USE PLAN query

hint [2], which allows forcing the optimizer to use a particular query plan that is provided by the user. While the initial prototyping and testing using real customer queries didn't indicate major issues, testing with complex queries generated by query generators showed that our technique required a lot more memory than what was anticipated. That discovery led to a number of generic improvements to the original algorithm.

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, our testing practice was to add a new test case every time each of our 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. After following this practice for some time, the net of regression tests becomes increasingly denser and eventually provides complete coverage of areas that may have been missed in the original test plan. The regular application of the above process introduced a large number of regressions tests in our benchmark. A significant number of the regression tests corresponded to queries with large estimation errors, and included areas of query optimization with known limitations, i.e. areas where the cost model was inaccurate. During the development cycle, there were times when our benchmark was heavily affected by the performance of those tests. In some cases, legitimate 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 evident to us that the practice of continuously extending our benchmark with various regression tests was problematic. While the regression tests represented areas in which customers had reported problems, they led to a benchmark that could produce inconclusive results and skew the coverage of scenarios in ways that were not well-understood. Today, we try to develop benchmarks 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 we had issues in the past and we want to track its performance, we will add a subset of that application workload into the benchmark. That helps us understand the impact of a code change on multiple queries from that application. We also try to characterize each query in the benchmark and understand its degree of conformance to the optimizer's model. That is helping us determine when a regression is caused due to a defect, a shift in optimization tradeoffs or due to side-effects of overfitting.

You improve on what you measure: The blend of application scenarios and their corresponding queries included in the benchmark influences the decisions made for the different engineering tradeoffs and eventually the tuning of the query optimizer. Initially, our testing process included a larger set of OLTP scenarios and a much smaller set of DSS-like application scenarios. OLTP customer databases were more easily accessible at the time and since they are typically smaller in size it was easier to adopt them in our test labs. Consequently, increases 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 rigorously as possible.

Test each component in isolation: The use of end-to-end query performance as the sole metric of plan choice quality has often been ineffective. First, changes made to components downstream from the optimizer in the Query Execution and Storage layers could result in end-to-end performance changes. Although the source of the regression could be pinpointed to the right component by simply checking for changes in the execution plan, there were times during the development cycle when both the execution plan and the implementation of the downstream components would change at the same time. In such cases, it was difficult to determine which component contained the root cause of the regression. Also, assumptions made by the query optimizer (such

as the CPU cost for a certain operator) would change causing regressions across our benchmark. This problem was mitigated by putting in place a parallel development process, which allows development in isolated code branches. Thus, changes could be tested in isolation, and if needed, component assumptions and expectations could be adjusted before the final code integration. The concept of testing in isolation extends to testing the internal subcomponents of the query optimizer as well. In addition to evaluating the optimizer using end-to-end query performance metrics, it is valuable to be able to test each layer of the cost model independently, so that the root cause of defects can be identified quickly within the faulty subcomponent. Additionally, validating the subcomponents located lower in the optimization stack in isolation (e.g. the Statistics subcomponent) guarantees that the subcomponent located higher in the stack (e.g. the Cardinality Estimation subcomponent) operates with valid inputs and assumptions when being validated itself.

Clarify the model: It is essential that the contracts between the different components and any assumptions made in the design are crisply defined in order to validate subcomponents in isolation. For example, the cardinality estimation component operates over histograms under the assumptions of independence and uniformity. Inputs that violate those assumptions will surely result in estimation errors and possibly in suboptimal plans. Creating inputs that provide the ideal conditions expected by the cardinality estimation component allows the development of highly deterministic tests, which return accurate results. While some assumptions and contracts are fundamental and well-understood, query optimization logic can be very fine-grained. Over time, the original rationale for certain parts of that logic can fade unless it is well-documented and ensured by tests.

Agree on when a regression is a defect: As discussed earlier, it is likely that 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 with the optimizer's model and assumptions, results in code health issues and architectural debt. Supporting special cases creates instant legacy on which new applications may rely on. 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. Customers need to be given the appropriate tools to work around plan choice issues, and guidance through tools and documentation so that they can correct and avoid bad practices.

Design for testability. During the past four to five years of product development we went back several times to add testability features into the query optimizer 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 that retrofitting testability later on. This helps in clarifying the interfaces and contracts between different subcomponents and the resulting test cases ensure that they remain valid during future development.

5 Future Challenges and Conclusions

Query optimization has a very big impact on the performance of a DBMS and it continuously evolves with new, more sophisticated optimization strategies. We describe two more challenges, which we expect will play a larger role in the future.

The transformation-based optimizer architecture of Volcano [6] and Cascades [5] provides an elegant framework, which makes the addition of new optimization rules easy. While it is straightforward to test each rule in isolation using simple use cases, it is harder to test the possible combinations and interactions between rules and ensure plan correctness. Also, with every addition of a new exploration rule, the search space expands and the number of possible plan choices increases accordingly. There is a need of advanced metrics and tools that help the analysis of the impact of such changes in the plan space. As query optimizers advance, the opportunities for

optimizations that provide value across most scenarios decrease, hence optimization logic becomes more granular. There has been research that indicates that query optimizers are already making very fine-grained choices [11], perhaps unnecessarily so, given the presence of cardinality estimation errors.

Although we described query optimization testing with focus on correctness and optimality, another interesting dimension of the query optimization quality is the concept of performance predictability. For a certain segment of mission-critical applications we see the need for predictable performance to be as important as the need for optimal performance. More work is needed on defining, measuring and validating predictability for different classes of applications.

Clearly, not all the challenges that we presented in this paper have been fully tackled. 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. As query optimizers become more sophisticated and supplemented with more self -tuning techniques, additional challenges will continue to surface.

References

[1] SQL Server 2005 Books Online, Introducing SQL Trace. http://technet.microsoft.com/en-us/library/ms191006.aspx.

[2] SQL Server 2005 Books Online, Understanding plan forcing. http://msdn2.microsoft.com/en-us/library/ms186343.aspx.

[3] Tpc benchmark h. decision support. http://www.tpc.org.

[4] S. Cluet and G. Moerkotte. On the complexity of generating optimal left-deep processing trees with cross products. In *ICDT '95: Proc. of the 5th Intl. Conf. on Database Theory*, pages 54–67, London, UK, 1995. Springer-Verlag.

[5] G. Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[6] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *ICDE '93: Proc. of the 9th Intl. Conf. on Data Engineering*, pages 209–218, 1993.

[7] S. Herbert H. Bati, L. Giakoumakis and A. Surna. A genetic approach for random testing of database systems. In *VLDB '07: Proc. of the 33rd Intl. Conf. on Very Large Data Bases*, pages 1243–1251. VLDB Endowment, 2007.

[8] T. Ibaraki and T. Kameda. On the optimal nesting order for computing n-relational joins. *ACM Trans. Database Syst.*, 9(3):482–502, 1984.

[9] Y. E. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *SIGMOD '91: Proc. of the 1991 ACM SIGMOD Intl. Conf. on Management of Data*, pages 268–277, New York, NY, USA, 1991.

[10] L. Kollar. SQL Server 2000 Technical Articles, Books Online, statistics used by the query optimizer in microsoft SQL Server 2000. http://msdn2.microsoft.com/en-us/library/aa902688(SQL.80).aspx.

[11] N. Reddy and J. R. Haritsa. Analyzing plan diagrams of database query optimizers. In *VLDB '05: Proc. of the 31st Intl. Conf. on Very Large Data Bases*, pages 1228–1239. VLDB Endowment, 2005.

[12] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD '79: Proc. of the 1979 ACM SIGMOD Intl. Conf. on Management of Data*, pages 23–34, New York, NY, USA, 1979. ACM.

[13] D. R. Slutz. Massive stochastic testing of SQL. In *VLDB '98: Proc. of the 24rd Intl. Conf. on Very Large Data Bases*, pages 618–622, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[14] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO - DB2's LEarning Optimizer. In *VLDB '01: Proc. of the 27th Intl. Conf. on Very Large Data Bases*, pages 19–28, San Francisco, CA, USA, 2001.

[15] F. Waas and C. Galindo-Legaria. Counting, enumerating, and sampling of execution plans in a cost-based query optimizer. In *SIGMOD '00: Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, pages 499–509, New York, NY, USA, 2000. ACM.