How I Learned to Stop Worrying and Love Re-optimization

Matthew Perron
MIT CSAIL
mperron@csail.mit.edu

Zeyuan Shang
MIT CSAIL
zeyuans@mit.edu

Tim Kraska
MIT CSAIL
kraska@csail.mit.edu

Michael Stonebraker
MIT CSAIL
stonebraker@csail.mit.edu

***Abstract*—Cost-based query optimizers remain one of the most important components of database management systems for analytic workloads. Though modern optimizers select plans close to optimal performance in the common case, a small number of queries are an order of magnitude slower than they could be. In this paper we investigate why this is still the case, despite decades of improvements to cost models, plan enumeration, and cardinality estimation. We demonstrate why we believe that a re-optimization mechanism is likely the most cost-effective way to improve end-to-end query performance. We find that even a simple re-optimization scheme can improve the latency of many poorly performing queries. We demonstrate that re-optimization improves the end-to-end latency of the top 20 longest running queries in the Join Order Benchmark by 27%, realizing most of the benefit of perfect cardinality estimation.**

***Index Terms*—Cardinality Estimation, Query Optimization, Dynamic Query Re-optimization**

I. Introduction

The basic structure of query optimizers, dating back to the pioneering paper in 1979 [1], is known to suffer serious performance problems. Notably, selectivity estimates assume the independence of columns in a table or across joins and also assume a uniform distribution of data elements in each column. In the real world, correlation between columns is widespread. For example, salary is usually correlated with age. Also, column values are often strongly skewed, for example 40 stocks out of 4000 in the NYSE account for 50% of the total volume of trades. Moreover, estimates for the size of intermediate tables become increasingly imprecise as one ascends higher in the plan tree.

Surprisingly, even with simplifying assumptions, query optimizers mostly choose plans that are near optimal performance even when cardinality estimates are wrong. But in a minority of cases, the optimizer chooses plans that are many times slower than optimal. This can significantly slow down the end-to-end execution time of query workloads. Put simply, a small number of optimization mistakes leads to workload performance far below what is possible. The purpose of this paper is to shed light on profitable directions to explore.

Leis et al. [2] provide experimental evidence that cardinality estimates, particularly join cardinality estimates, are poor and can result in queries with noticeably sub-optimal plans. The inference is that improving estimates would result in better plans. In Figure 1 we consider the 20 longest running queries in the Join Order Benchmark [2] (JOB). The total query planning time, including optimization, of all 20 queries is in orange and the time to execute the resulting plans is in blue. We define perfect-(n) as a cardinality estimator with perfect estimates for joins of n tables and fewer. We then compare the PostgreSQL optimizer with perfect-(3), perfect-(4), a re-optimization scheme, and all perfect estimates. perfect-(3) achieves no improvement for the longest running queries, while perfect-(4) and re-optimization improve end-to-end latency by 25%. This is a worst-case scenario since we assume ad-hoc queries that have to be optimized before execution. If we instead assume a recurring workload where optimized query plans have been cached from previous executions, the improvement with re-optimization is over 35%, including about 30 seconds to re-optimize. We conclude that improvements to cardinality estimates that do not approach perfect-(4) will not improve the end-to-end latency of JOB. This level of improvement currently seems out of reach. However, We find that much of the benefit of perfect estimates can be realized with re-optimization strategies.

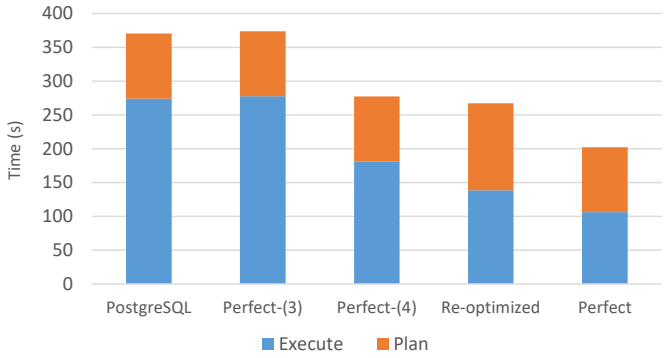

Fig. 1. Total Query planning and execution times for the top 20 longest running queries in JOB with the default PostgreSQL optimizer.

In Section II we examine how much cardinality estimates must be improved to significantly improve the execution time of JOB. We then consider re-optimization strategies as an alternative in Section III. For more detailed information, see our full report on arXiv.

II. On Cardinality Estimation Improvement

We first consider how the quality of cardinality estimates impacts the runtime of JOB.

Relative Runtime	Number of Queries
0.1 - 0.8	7
0.8 - 1.2	32
1.2 - 2.0	28
2.0 - 5.0	32
> 5.0	14

TABLE I
EXECUTION TIME OF JOB QUERIES WITH POSTGRESQL CARDINALITY ESTIMATION RELATIVE TO PERFECT-(17)

We modified PostgreSQL 10.1 allowing us to replace the PostgreSQL cardinality estimates with arbitrary values. We then compare the performance of different cardinality estimation schemes. We add indexes on foreign key columns, as suggested by Leis et al. [2], making access path selection more challenging.

JOB, proposed by Leis et al. [2], contains a set of 113 queries on the Internet Movie Database (IMDB) dataset. Each query is a select-project-join query with 4 to 17 tables and only equi-joins. This real world dataset includes both skew and correlation. Thus, queries in JOB have proved more difficult for optimizers to choose good plans than standard decision support benchmarks like TPC-H [3]. As a result, JOB is a popular choice for evaluation in recent query optimization work.

We execute our experiments on a `n1-highmem-4` virtual machine instance on Google Cloud Platform with a 256GB SSD persistent disk, and 26 GB of memory. We disable intra-query parallelism, and set `shared_buffers`, `temp_buffers`, and `work_mem` to 400MB each. Because PostgreSQL, in addition to managing it's own buffer pool, uses the file system's large buffer cache, all tables and indexes are cached in memory.

To give PostgreSQL the best chance at good cardinality estimates, we set the `default_statistics_target` to it's maximum value and execute `ANALYZE` to collect statistics.

We define "planning time" as the time to parse and optimize the query and "execution time" as the time spent on execution of query plans.

A. Quality of Cardinality Estimates and Execution Time

We explore how the quality of improved cardinality estimates impacts the execution time of the workload. To get a rough estimate, we define perfect-(n) as a version of PostgreSQL where the cardinality estimator is given an oracle for cardinality estimates on joins of n tables and fewer. Perfect-(n) has an oracle for a subset of cardinality estimates of perfect-$(n+1)$. To estimate the cardinality of joins of more than n tables, the default PostgreSQL cardinality estimate is used, with perfect estimates of n tables and fewer used as input. For example, to make an estimate of a join of 5 tables in perfect-(4), the cardinality estimator receives as input perfect base table cardinalities and join cardinalities of up to 4 tables, but otherwise uses its default estimation techniques including independence and uniformity assumptions. The quality of estimates for joins of 5 tables are, on average, better with perfect-(4) than perfect-(3). Perfect-(1) gives only perfect base table cardinality estimates. Since the there are at most 17 relations in JOB perfect-(17) has perfect cardinality estimates.

We find perfect estimates on base tables, pairs of tables, and triples give virtually no benefit to JOB execution time. Surprisingly, any method of improving cardinality estimation that does not achieve estimates better than perfect-(3) can expect to achieve nearly no improvement to execution time.

We look at the potential for execution time improvement in JOB by comparing plans generated with PostgreSQL cardinality estimates to plans generated with perfect cardinalities. While it is possible that better plans are not in the search space of PostgreSQL, perfect cardinality estimates improve execution time of the benchmark by a factor of two. This means much better plans are in the search space of the optimizer but are simply not chosen due to cardinality estimation or cost model errors. Less than 15% of queries in the benchmark have execution time more than five times the optimal execution time, seen in Table I. Errors in just 20 queries make up more than 95% of the execution time difference of perfect estimates and PostgreSQL estimates, not including query planning time. Surprisingly, the PostgreSQL cardinality estimation model, assuming no correlation between columns and assuming uniformity chooses plans with execution time within a factor of two of a plan generated with perfect cardinalities in nearly 60% of queries. This indicates that for most queries the additional cost of building more statistics or sampling to improve estimates will not decrease query execution time substantially, but may slow down planning time.

One challenge in improving cardinality estimates is how many must be accurately predicted. While query optimizers use heuristics like avoiding plans with Cartesian products, the number of estimates made by the cardinality estimator is still large. The PostgreSQL cardinality estimator makes more than 13 thousand estimates for the most complex query in JOB. Given this number of estimates, approaching perfect-(4) for a large range of queries seems to be exceedingly difficult. It is particularly challenging because perfect-(4) requires perfect-(3), perfect-(2), and so forth. While not every dataset and workload have the same properties as JOB, this is evidence of the challenges of standard approaches to improving cardinality estimation.

B. Selective Improvement of Cardinality Estimates

One suggestion to improve cardinality estimates is to discover cardinality estimation errors during query execution, and correct them in future executions of similar queries. LEO [4], is an example of this approach. To demonstrate the limitations of these techniques, we take a set of poorly performing queries and find the lowest join operator in the plan tree with cardinality estimates 32 times larger or smaller than the true cardinality. We set estimates for the join and all estimates below it in the plan tree to their true values then plan and execute the query again. We continue this process until no operator has an estimation error larger than the threshold. This is the best case for incremental improvement strategies

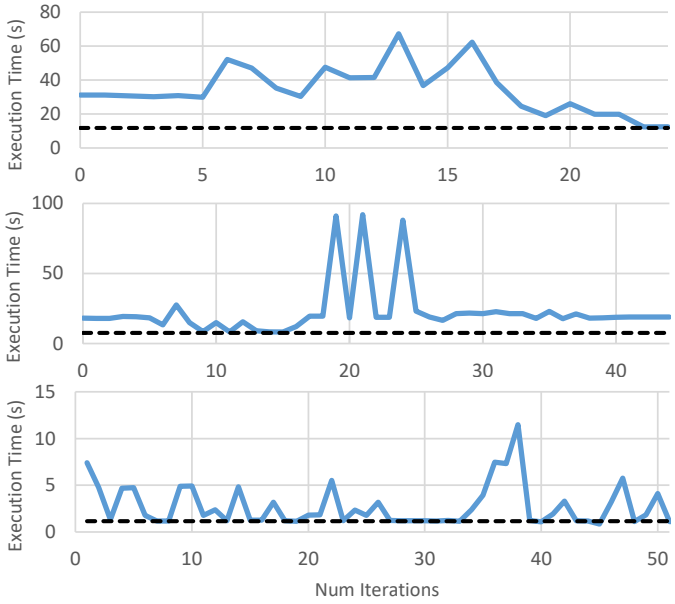

Fig. 2. Execution time of queries with iterative improvement of cardinality estimates. The dotted line is execution time with perfect estimates. From top to bottom, plots of queries 16b, 25c, and 30a.

since we execute the same query repeatedly and fix cardinality estimates perfectly. In practice, correcting cardinality estimates through executed queries is more challenging since queries or data may change over time.

In Figure 2, we plot the execution time of queries 16b, 25c, and 30a as we incrementally improve their cardinality estimates as described above. We compare their execution time to that chosen using perfect cardinality estimates. For some of the worst performing queries, many corrections to cardinality estimates are required before discovering a good plan. Query 16b is executed 24 times before an efficient plan is chosen. In queries 25c and 30a, the query optimizer chooses a plan near optimal with a small number of cardinality estimate corrections. However, continuing to improve estimates causes the optimizer to choose plans several times slower than optimal. This means that correcting only a subset of cardinality estimates can cause the optimizer to choose query plans significantly slower than the original plan, as seen after 19 iterations on query 25c. To arrive at a plan near optimal, all catastrophic plans must have higher costs than a good plan. This is difficult to guarantee when improving only a subset of cardinality estimates.

III. Re-optimization

We next explore how much of the benefit of perfect cardinality estimates can be realized with dynamic query re-optimization.

We simulate a simple query re-optimization scheme by examining the `EXPLAIN ANALYZE` output of each query and comparing the true cardinalities to the those predicted by the PostgreSQL cardinality estimator. For the lowest join operator in the query plan with a true cardinality a factor of n larger

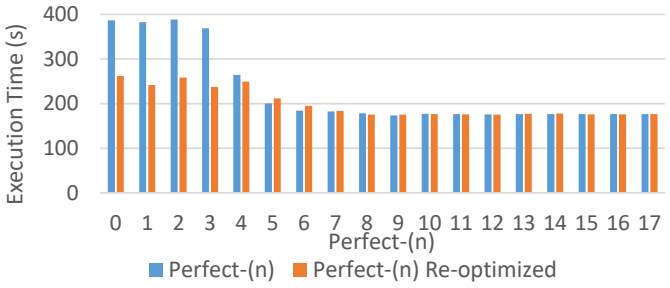

Fig. 3. Total execution time of JOB varying the quality of estimates from perfect-(0) to perfect-(17), with and without re-optimization

or smaller than the estimated cardinality, we rewrite this subquery to create a temporary table instead. For the remainder of the query, we replace all the tables in the poorly estimated join with the temporary table and re-plan. We repeat this procedure until no join operators in the query plan have a cardinality estimate a factor of n different from the true cardinality. While materialization of intermediate results may be expensive, it gives the optimizer a chance to correct itself for the rest of the execution.

We then measure the planning and execution time for the resulting "re-optimized" query. For the planning time, we sum the planing time of the original query and the planning time of the `SELECT` queries. We do not include the time required to plan the creation of temporary tables since this is already included in the original query planning time. Thus the re-optimized query is a series of `CREATE TEMPORARY TABLE` commands followed by a `SELECT` query to generate the final result.

Total execution time is measured by summing the execution time of each `CREATE TEMPORARY TABLE` command and the final `SELECT` query. We report execution and planning time as reported by PostgreSQL's `EXPLAIN ANALYZE` command. Unless otherwise noted, we report only the execution time of queries, and exclude parsing and optimization time. For many short-running queries in JOB, PostgreSQL planning time exceeds execution time. Clearly it does not pay off to re-optimize these queries.

We believe this is a reasonable approximation of a simplistic re-optimization scheme. Although it is possible that this approximation breaks a pipeline present in the original query plan, our simulation provides a reasonable approximation for the upper bound of the cost of re-optimization schemes, since it requires a full materialization of intermediate tables. More sophisticated re-optimization schemes like Rio [5], may perform better than our simulation.

A. Re-optimization Triggers

Clearly the decision of when to re-optimize is critical to a scheme's performance. In our setup we re-optimize when the relative cardinality estimate error crosses a threshold. If the true cardinality of a join is n times more or less than we expect, we materialize the result and re-optimize the rest of the query. That is, we re-optimize a query when the Q-error [6]

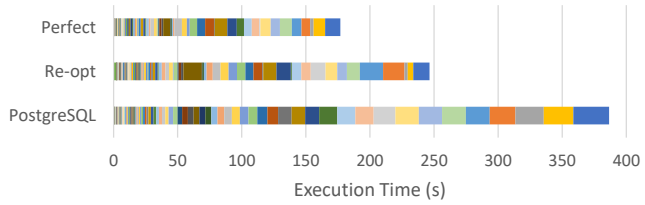

Fig. 4. Total execution time comparison of perfect, re-optimized, and PostgreSQL by query, ordered by execution time on PostgreSQL.

exceeds a threshold value. If the threshold is too low even good plans may be unnecessarily re-optimized. If set too high re-optimization will never be triggered. We experimented with varying the re-optimization threshold and observed the impact on planning and execution time, comparing to PostgreSQL and using perfect estimates. Based on these experiments, we found that setting the threshold to 32 gives the best query execution time improvement. Therefore we set the threshold to 32.

Surprisingly, even re-optimizing at a threshold of two, the lowest we tested, only increases the query planning time by about 42% over PostgreSQL while decreasing the execution time by 40%. It is worth noting that the planning time remains constant independent of the data size while the execution time will increase with the size of the data. While planning time is significant, sometimes exceeding execution time, this is a result of the small size of the IMDB dataset, rather than the excessive costs of planning. Frequent materialization of tables by setting the threshold to a relatively low value of two has a minimal impact on query execution time. A re-optimization threshold of two degrades performance only 10% compared to the lowest re-optimization execution time measured. This indicates that, at least for JOB, setting the re-optimization threshold too low is still better than not re-optimizing at all.

With longer running queries, the additional cost of planning is likely negligible. Because we expect that re-optimization will provide the most benefit for longer running queries taking several minutes, we report only execution time for the following experiments.

B. Re-optimization and Better Cardinality Estimates

We believe that without a redesign of the query optimizer, query re-optimization is most likely to lead to the biggest gains in end-to-end query latency. However, if a method to improve cardinalities approaches perfect-(3) or perfect-(4), re-optimization can still improve execution time. As cardinality estimates improve, the need to re-optimize decreases. In Figure 3, we compare perfect-(n) plus re-optimization for varying values of n. We see that re-optimization improves the latency of perfect-(n) estimates until perfect-(5). While re-optimizing perfect-(5) slows the execution of the workload, the risk is relatively small, The execution time of the benchmark is only 6% slower with re-optimization than only perfect-(5).

C. The Benefits of Re-optimization

In Figure 4 we compare the execution time of standard PostgreSQL to perfect-(17) and our re-optimization simulation. We visualize the total execution time by query, and order queries from shortest to longest in the original PostgreSQL execution. Note that this figure does not include planning time. While re-optimization does not significantly improve the execution time of most of the shortest queries, the impact of re-optimization on the longest running queries achieves much of the benefits of having perfect estimates. Thus, we find there is no need to re-optimize the shortest queries, particularly because the re-optimization may exceed query execution time.

D. The Risks of Re-optimization

While query re-optimization has a clear benefit for JOB it is not without risks. In several queries the execution time increases significantly from default PostgreSQL. In fact, the worst performing re-optimized query has more than 100 times worse execution time than the original query executed by PostgreSQL. However, since the original execution time of the query is only about ten milliseconds, the impact on the execution time of the whole benchmark is negligible. This can be avoided by re-optimizing only long-running queries.

IV. Conclusion

In this paper we showed that a simple re-optimization strategy achieves much of the benefit of perfect cardinality estimation in JOB. While we confirm that having perfect cardinality estimates is a clear way to improve query plans, we show that achieving estimates good enough to impact execution time of a workload is daunting.

Much re-optimization work was performed more than a decade ago. Our experiments show that re-optimization significantly improves query execution time compared to default PostgreSQL cardinality estimates on a single threaded row store for a single workload. However, modern query execution engines are heavily pipelined, compile queries, execute on many machines, or use columnar storage. Re-optimization becomes more complex with these developments. If a query is re-planned, this may require another expensive compilation phase. But the costs of more sophisticated schemes remain unclear. More work is needed to determine the benefits and feasibility of re-optimization for modern workloads on modern systems.

References

[1] 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 *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*. ACM, 1979, pp. 23–34.

[2] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, "How good are query optimizers, really?" *Proceedings of the VLDB Endowment*, vol. 9, no. 3, pp. 204–215, 2015.

[3] The Transaction Processing Council, "TPC-H Benchmark (Revision 2.16.0)," http://www.tpc.org/tpch/, June 2013.

[4] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil, "Leo-db2's learning optimizer," in *VLDB*, vol. 1, 2001, pp. 19–28.

[5] S. Babu, P. Bizarro, and D. DeWitt, "Proactive re-optimization," in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. ACM, 2005, pp. 107–118.

[6] G. Moerkotte, T. Neumann, and G. Steidl, "Preventing bad plans by bounding the impact of cardinality estimation errors," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 982–993, 2009.