Sampling-Based Query Re-Optimization

Wentao Wu Jeffrey F. Naughton Harneet Singh
Department of Computer Sciences, University of Wisconsin-Madison
{wentaowu, naughton, harneet}@cs.wisc.edu

ABSTRACT

Despite of decades of work, query optimizers still make mistakes on "difficult" queries because of bad cardinality estimates, often due to the interaction of multiple predicates and correlations in the data. In this paper, we propose a low-cost post-processing step that can take a plan produced by the optimizer, detect when it is likely to have made such a mistake, and take steps to fix it. Specifically, our solution is a sampling-based iterative procedure that requires almost no changes to the original query optimizer or query evaluation mechanism of the system. We show that this indeed imposes low overhead and catches cases where three widely used optimizers (PostgreSQL and two commercial systems) make large errors.

1. INTRODUCTION

Query optimizers rely on decent cost estimates of query plans. Cardinality/selectivity estimation is crucial for the accuracy of cost estimates. Unfortunately, although decades of research has been devoted to this area and significant progress has been made, cardinality estimation remains challenging. In current database systems, the dominant approach is to keep various statistics, primarily histograms, about the data. While histogram-based approaches have worked well for estimating selectivities of local predicates (i.e., predicates over a column of a base table), query optimizers still make mistakes on "difficult" queries, often due to the interaction of multiple predicates and correlations in the data [28].

Indeed, there is a great deal of work in the literature exploring selectivity estimation techniques beyond histogram-based ones (see Section 6). Nonetheless, histogram-based approaches remain dominant in practice because of its low overhead. Note that, query optimizers may explore hundreds or even thousands of candidates when searching for an optimal query plan, and selectivity estimation needs to be done for each candidate. As a result, a feasible solution has to improve cardinality estimation quality without significantly increasing query optimization time.

In this paper, we propose a low-cost post-processing step that can take a plan produced by the optimizer, detect when it is likely to have made such a mistake, and take steps to fix it. Specifically, our solution is a sampling-based iterative procedure that requires almost no changes to the original query optimizer or query evaluation mechanism of the system. We show that this indeed imposes low overhead and catches cases where three widely used optimizers (PostgreSQL and two commercial systems) make large errors.

In more detail, sampling-based approaches (e.g., [11, 20, 27]) automatically reflect correlation in the data and between multiple predicates over the data, so they can provide better cardinality estimates on correlated data than histogram-based approaches. However, sampling also incurs higher overhead. In previous work [39, 40, 41], the authors investigated the effectiveness of using sampling-based cardinality estimates to get better query running time predictions. The key observation is the following: while it is infeasible to use sampling for all plans explored by the optimizer, it is feasible to use sampling as a "post-processing" step after the search is finished to detect potential errors in optimizer's original cardinality estimates for the final chosen plan.

Inspired by this observation, our basic idea is simple: if significant cardinality estimation errors are detected, the optimality of the returned plan is then itself questionable, so we go one step further to let the optimizer re-optimize the query by also feeding it the cardinality estimates refined via sampling. This gives the optimizer second chance to generate a different, perhaps better, plan. Note that we can again apply the sampling-based validation step to this new plan returned by the optimizer. It therefore leads to an iterative procedure based on feedback from sampling: we can repeat this optimization-then-validation loop until the plan chosen by the optimizer does not change. The hope is that this re-optimization procedure can catch large optimizer errors *before* the system even begins executing the chosen query plan.

A couple of natural concerns arise regarding this simple query re-optimization approach. First, how efficient is it? As we have just said, sampling should not be abused given its overhead. Since we propose to run plans over samples iteratively, how fast does this procedure converge? To answer this question, we conduct a theoretical analysis as well as an experimental evaluation. Our theoretical study suggests that, the expected number of iterations can be bounded by $O(\sqrt{N})$, where N is the number of plans considered by the optimizer in its search space. In practice, this upper bound can rarely happen. Re-optimization for most queries tested in our experiments converges after only a few rounds of iteration, and the time spent on re-optimization is ignorable compared with the corresponding query running time.

Second, is it useful? Namely, does re-optimization really generate a better query plan? This raises the question of how to evaluate the effectiveness of re-optimization. Query optimizers appear to do well almost all of the time. But the experience of optimizer developers we have talked to is that there are a small number of "difficult" queries that cause them most of the pain. That is, most

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

 ISBN 978-1-4503-3531-7/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2882903.2882914

of the time the optimizer is very good, but when it is bad, it is very bad. Indeed, Lohman [28] recently gave a number of compelling real-world instances of optimizers that, while presumably performing well overall, make serious mistakes on specific queries and data sets. It is our belief that an important area for optimizer research is to focus precisely on these few "difficult" queries.

We therefore choose to evaluate re-optimization over those difficult, corner-case queries. Now the hard part is characterizing exactly what these "difficult" queries look like. This will inevitably be a moving target. If benchmarks were to contain "difficult" queries, optimizers would be forced to handle them, and they would no longer be "difficult," so we cannot look to the major benchmarks for examples. In fact, we implemented our approach in PostgreSQL and tested it on the TPC-H benchmark database, and we did observe significant performance improvement for certain TPC-H queries (Section 5.2). However, for most of the TPC-H queries, the re-optimized plans are exactly the same as the original ones. We also tried the TPC-DS benchmark and observed similar phenomena (Appendix A.2). Using examples of real-world difficult queries would be ideal, but we have found it impossible to find well-known public examples of these queries and the data sets they run on.

It is, however, well-known that many difficult queries are made difficult by correlations in the data — for example, correlations between multiple selections, and more likely correlations between selections and joins [28]. This is our target in this paper. It is in fact very easy to generate examples of these queries and data sets that confuse all the optimizers (PostgreSQL and two commercial RDBMS) that we tested — such examples are the basis for our "optimizer torture test" presented in Section 4. We observed that re-optimization becomes superior on these cases (Section 5.3). While original query plans often take hundreds or even thousands of seconds to finish, after re-optimization all queries can finish in less than 1 second. We therefore hope that our re-optimization technique can help cover some of those corner cases that are challenging to current query optimizers.

The idea of query re-optimization goes back to two decades ago (e.g. [25, 30]). The main difference between this line of work and our approach is that re-optimization was previously done *after* a query begins to execute whereas our re-optimization is done *before* that. While performing re-optimization during query execution has the advantage of being able to observe accurate cardinalities, it suffers from (sometimes significant) runtime overhead such as materializing intermediate results that have been generated. Meanwhile, runtime re-optimization frameworks usually require significant changes to query optimizer's architecture. Our compile-time re-optimization approach is more lightweight. The only additional cost is due to running tentative query plans over samples. The modification to the query optimizer and executor is also limited: our implementation in PostgreSQL needs only several hundred lines of C code. Furthermore, we should also note that our compile-time re-optimization approach actually does not conflict with these previous runtime re-optimization techniques: the plan returned by our re-optimization procedure could be further refined by using runtime re-optimization. It remains interesting to investigate the effectiveness of this combination framework.

The rest of the paper is organized as follows. We present the details of our iterative sampling-based re-optimization algorithm in Section 2. We then present a theoretical analysis of its efficiency in terms of the number of iterations it requires and the quality of the final plan it returns in Section 3. To evaluate the effectiveness of this approach, we further design a database (and a set of queries) with highly correlated data in Section 4, and we report experimental evaluation results on this database as well as the TPC-H benchmark databases in Section 5. We discuss related work in Section 6 and conclude the paper in Section 7.

2. THE RE-OPTIMIZATION ALGORITHM

In this section, we first introduce necessary background information and terminology, and then present the details of the re-optimization algorithm. We focus on using sampling to refine selectivity estimates for join predicates, which are the major source of errors in practice [28]. The sampling-based selectivity estimator we used is tailored for join queries [20], and it is our goal in this paper to study its effectiveness in query optimization when combined with our proposed re-optimization procedure. Nonetheless, sampling can also be used to estimate selectivities for other types of operators, such as aggregates (i.e., "Group By" clauses) that require estimation of the number of distinct values (e.g. [11]). We leave the exploration of integrating other sampling-based selectivity estimation techniques into query optimization as interesting future work.

2.1 Preliminaries

In previous work [39, 40, 41], the authors used a sampling-based selectivity estimator proposed by Haas et al. [20] for the purpose of predicting query running times. In the following, we provide an informal description of this estimator.

Let R_1, ..., R_K be K relations, and let R_k^s be the sample table of R_k for $1 \leq k \leq K$. Consider a join query $q = R_1 \bowtie \cdots \bowtie R_K$. The selectivity ρ_q of q can be estimated as

$$\hat{\rho}_q = \frac{|R_1^s \bowtie \cdots \bowtie R_K^s|}{|R_1^s| \times \cdots \times |R_K^s|}.$$

It has been shown that this estimator is both unbiased and strongly consistent [20]: the larger the samples are, the more accurate this estimator is. Note that this estimator can be applied to joins that are sub-queries of q as well.

2.2 Algorithm Overview

As mentioned in the introduction, cardinality estimation is challenging and cardinality estimates by optimizers can be erroneous. This potential error can be noticed once we apply the aforementioned sampling-based estimator to the query plan generated by the optimizer. However, if there are really significant errors in cardinality estimates, the optimality of the plan returned by the optimizer can be in doubt.

If we replace the optimizer's cardinality estimates with sampling-based estimates and ask it to re-optimize the query, what would happen? Clearly, the optimizer will either return the same query plan, or a different one. In the former case, we can just go ahead to execute the query plan: the optimizer does not change plans even with the new cardinalities. In the latter case, the new cardinalities cause the optimizer to change plans. However, this new plan may still not be trustworthy because the optimizer may still decide its optimality based on erroneous cardinality estimates. To see this, let us consider the following example.

EXAMPLE 1. *Consider the two join trees T_1 and T_2 in Figure 1. Suppose that the optimizer first returns T_1 as the optimal plan. Sampling-based validation can then refine cardinality estimates for the three joins: $A \bowtie B$, $A \bowtie B \bowtie C$, and $A \bowtie B \bowtie C \bowtie D$. Upon knowing these refined estimates, the optimizer then returns T_2 as the optimal plan. However, the join $C \bowtie D$ in T_2 is not observed in T_1 and its cardinality has not been validated.*

Hence, we can again apply the sampling-based estimator to this new plan and repeat the re-optimization process. This then leads to an iterative procedure.

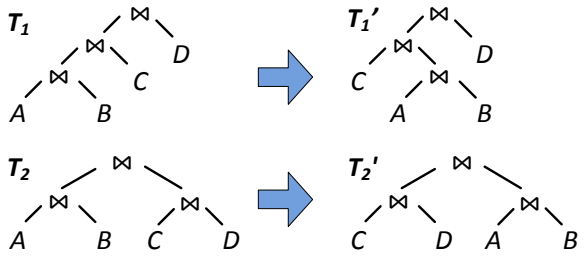


Figure 1: Join trees and their local transformations.

Algorithm 1 outlines the above idea. Here, we use Γ to represent the sampling-based cardinality estimates for joins that have been validated by using sampling. Initially, Γ is empty. In the round i ($i \geq 1$), the optimizer generates a query plan P_i based on the current information preserved in Γ (line 5). If P_i is the same as P_{i-1}, then we can terminate the iteration (lines 6 to 8). Otherwise, P_i is new and we invoke the sampling-based estimator over it (line 9). We use Δ_i to represent the sampling-based cardinality estimates for P_i, and we update Γ by merging Δ_i into it (line 10). We then move to the round $i+1$ and repeat the above procedure (line 11).

Algorithm 1: Sampling-based query re-optimization

Input: q, a given SQL query
Output: P_q, query plan of q after re-optimization

```
1  Γ ← ∅;
2  P_0 ← null;
3  i ← 1;
4  while true do
5  |  P_i ← GetPlanFromOptimizer(Γ);
6  |  if P_i is the same as P_{i-1} then
7  |  |  break;
8  |  end
9  |  Δ_i ← GetCardinalityEstimatesBySampling(P_i);
10 |  Γ ← Γ ∪ Δ_i;
11 |  i ← i + 1;
12 end
13 Let the final plan be P_q;
14 return P_q;
```

Note that this iterative process has as its goal improving the selected plan, not finding a new globally optimal plan. It is certainly possible that the iterative process misses a good plan because the iterative process does not explore the complete plan space — it only explores neighboring transformations of the chosen plan. Nonetheless, as we will see in Section 5, this local search is sufficient to catch and repair some very bad plans.

3. THEORETICAL ANALYSIS

In this section, we present an analysis of Algorithm 1 from a theoretical point of view. We are interested in two aspects of the re-optimization procedure:

- *Efficiency*, i.e., how many rounds of iteration does it require before it terminates?
- *Effectiveness*, i.e., how good is the final plan it returns compared to the original plan, in terms of the cost metric used by the query optimizer?

Our following study suggests that (i) the expected number of rounds of iteration in the worst case is upper-bounded by $O(\sqrt{N})$ where N is the number of query plans explored in the optimizer's search space (Section 3.3); and (ii) the final plan is guaranteed to be no worse than the original plan if sampling-based cost estimates are consistent with the actual costs (Section 3.4).

3.1 Local and Global Transformations

We start by introducing the notion of local/global transformations of query plans. In the following, we use $\mathrm{tree}(P)$ to denote the *join tree* of a query plan P. A join tree is the logical skeleton of a physical plan, which is represented as the set of *ordered* logical joins contained in P. For example, the representation of T_2 in Figure 1 is $T_2 = \{A \bowtie B, C \bowtie D, A \bowtie B \bowtie C \bowtie D\}$.

DEFINITION 1 (LOCAL/GLOBAL TRANSFORMATION). *Two join trees T and T' (of the same query) are* local *transformations of each other if T and T' contain the same set of* unordered *logical joins. Otherwise, they are global transformations.*

In other words, local transformations are join trees that subject to only exchanges of left/right subtrees. For example, $A \bowtie B$ and $B \bowtie A$ are different join trees, but they are local transformations. In Figure 1 we further present two join trees T_1' and T_2' that are local transformations of T_1 and T_2. By definition, a join tree is always a local transformation of itself.

Given two plans P and P', we also say that P' is a local/global transformation of P if $\mathrm{tree}(P')$ is a local/global transformation of $\mathrm{tree}(P)$. In addition to potential exchange of left/right subtrees, P and P' may also differ in specific choices of physical join operators (e.g., hash join vs. sort-merge join). Again, by definition, a plan is always a local transformation of itself.

3.2 Convergence Conditions

At a first glance, even the convergence of Algorithm 1 is questionable. Is it possible that Algorithm 1 keeps looping without termination? For instance, it seems to be possible that the re-optimization procedure might alternate between two plans $P1$ and $P2$, i.e., the plans generated by the optimizer are $P1$, $P2$, $P1$, $P2$, ... As we will see, this is impossible and Algorithm 1 is guaranteed to terminate. We next present a sufficient condition for the convergence of the re-optimization procedure. We first need one more definition regarding plan coverage.

DEFINITION 2 (PLAN COVERAGE). *Let P be a given query plan and $\mathcal{P}$ be a set of query plans. P is* covered *by $\mathcal{P}$ if*

$$\mathrm{tree}(P) \subseteq \bigcup\nolimits_{P' \in \mathcal{P}} \mathrm{tree}(P').$$

That is, all the joins in $\mathrm{tree}(P)$ are included in the join trees of $\mathcal{P}$. As a special case, any plan that belongs to $\mathcal{P}$ is covered by $\mathcal{P}$.

Let P_i ($i \geq 1$) be the plan returned by the optimizer in the i-th re-optimization step. We have the following convergence condition for the re-optimization procedure:

THEOREM 1 (CONDITION OF CONVERGENCE). *Algorithm 1 terminates after $n+1$ ($n \geq 1$) steps if P_n is covered by $\mathcal{P} = \{P_1, ..., P_{n-1}\}$.*

PROOF. If P_n is covered by $\mathcal{P}$, then using sampling-based validation will not contribute anything new to the statistics Γ. That is, $\Delta_n \cup \Gamma = \Gamma$. Therefore, P_{n+1} will be the same as P_n, because the optimizer will see the same Γ in the round $n+1$ as that in the round n. Algorithm 1 then terminates accordingly (by lines 6 to 8). □

Note that the convergence condition stated in Theorem 1 is sufficient by not necessary. It could happen that P_n is not covered by

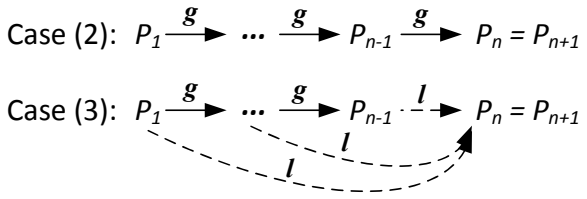


Figure 2: Characterization of the re-optimization procedure (g and l stand for global and local transformations, respectively). For ease of illustration, P_i is only noted as a global transformation of P_{i-1}, but we should keep in mind that P_i is also a global transformation of all the P_j's with $j < i$.

the previous plans $\{P_1, ..., P_{n-1}\}$ but $P_{n+1} = P_n$ after using the validated statistics (e.g., if there are no significant errors detected in cardinality estimates of P_n).

COROLLARY 1 (CONVERGENCE GUARANTEE). *Algorithm 1 is guaranteed to terminate.*

PROOF. Based on Theorem 1, Algorithm 1 would only continue if P_n is not covered by $\mathcal{P} = \{P_1, ..., P_{n-1}\}$. In that case, P_n should contain at least one join that has not been included by the plans in $\mathcal{P}$. Since the total number of possible joins considered by the optimizer is finite, $\mathcal{P}$ will eventually reach some fixed point if it keeps growing. The re-optimization procedure is guaranteed to terminate upon that fixed point, again by Theorem 1. □

Theorem 1 also implies the following special case:

COROLLARY 2. *The re-optimization procedure terminates after $n+1$ ($n \geq 1$) steps if $P_n \notin \mathcal{P}$ but P_n is a local transformation of some $P \in \mathcal{P} = \{P_1, ..., P_{n-1}\}$.*

PROOF. By definition, if P_n is a local transformation of some $P \in \mathcal{P}$, then $\mathrm{tree}(P_n)$ and $\mathrm{tree}(P)$ contain the same set of unordered logical joins. By Definition 2, P_n is covered by $\mathcal{P}$. Therefore, the re-optimization procedure terminates after $n+1$ steps, by Theorem 1. □

Also note that Corollary 2 has covered a common case in practice that P_n is a local transformation of P_{n-1}.

Based on Corollary 2, we next present an important property of the re-optimization procedure.

THEOREM 2. *When the re-optimization procedure terminates, exactly one of the following three cases holds:*

(1) It terminates after 2 steps with $P_2 = P_1$.

(2) It terminates after $n+1$ steps ($n > 1$). For $1 \leq i \leq n$, P_i is a global transformation of P_j with $j < i$.

(3) It terminates after $n+1$ steps ($n > 1$). For $1 \leq i \leq n-1$, P_i is a global transformation of P_j with $j < i$, and P_n is a local transformation of some $P \in \mathcal{P} = \{P_1, ..., P_{n-1}\}$.

That is, when the procedure does not terminate trivially (Case (1)), it can be characterized as a sequence of global transformations with *at most* one more local transformation before its termination (Case (2) or (3)). Figure 2 illustrates the latter two nontrivial cases. A formal proof of Theorem 2 is included in [1].

One might raise the question here that why we separate Case (3) from Case (2) in Theorem 2. Why would the fact that there might be one local transformation be interesting? The interesting point here is not that we may have one local transformation. Rather, it is that local transformation can occur at most once, and, if it occurs, it must be the last one in the transformation chain. Actually, we have observed in our experiments that re-optimization of some queries only involves (one) local transformation. This, however, by no means suggests that local transformations do not lead to any significant optimizations. For instance, sometimes even just replacing an index-based nested loop join by a hash join (or vice versa) can result in nontrivial performance improvement.

3.3 Efficiency

We are interested in how fast the re-optimization procedure terminates. As pointed out by Theorem 2, the convergence speed depends on the number of *global* transformations the procedure undergoes. In the following, we first develop a probabilistic model that will be used in our analysis, and then present analytic results for the general case and several important special cases.

3.3.1 A Probabilistic Model

Consider a queue of N balls. Originally all balls are not marked. We then conduct the following procedure:

PROCEDURE 1. *In each step, we pick the ball at the head of the queue. If it is marked, then the procedure terminates. Otherwise, we mark it and randomly insert it back into the queue: the probability that the ball will be inserted at the position i ($1 \leq i \leq N$) is uniformly $1/N$.*

We next study the expected number of steps that Procedure 1 would take before its termination.

LEMMA 1. *The expected number of steps that Procedure 1 takes before its termination is:*

$$S_N = \sum_{k=1}^{N} k \cdot (1 - \frac{1}{N}) \cdots (1 - \frac{k-1}{N}) \cdot \frac{k}{N}. \qquad (1)$$

We include a proof of Lemma 1 in [1]. We can further show that S_N is upper-bounded by $O(\sqrt{N})$.

THEOREM 3. *The expected number of steps S_N as presented in Lemma 1 satisfies $S_N = O(\sqrt{N})$.*

The proof of Theorem 3 is in [1]. In Figure 3, we plot S_N by increasing N from 1 to 1000. As we can see, the growth speed of S_N is very close to that of $f(N) = \sqrt{N}$.

3.3.2 The General Case

In a nutshell, query optimization can be thought of as picking a plan with the lowest estimated cost among a number of candidates. Different query optimizers have different search spaces, so in general we can only assume a search space with N different join trees that will be considered by an optimizer.[1] Let these trees be T_1, ..., T_N, ordered by their estimated costs. The re-optimization procedure can then be thought of as shuffling these trees based on their refined cost estimates. This procedure terminates whenever (or even before) the tree with lowest estimated cost reoccurs, that is, when some tree appears at the head position for the second time. Therefore, the probabilistic model in the previous section applies here. As a result, by Lemma 1, the expected number of steps for this procedure to terminate is S_N. We formalize this result as the following theorem:

[1]By "different" join trees, we mean join trees that are global transformations of each other. We use this convention in the rest of the paper unless specific clarifications are noted.

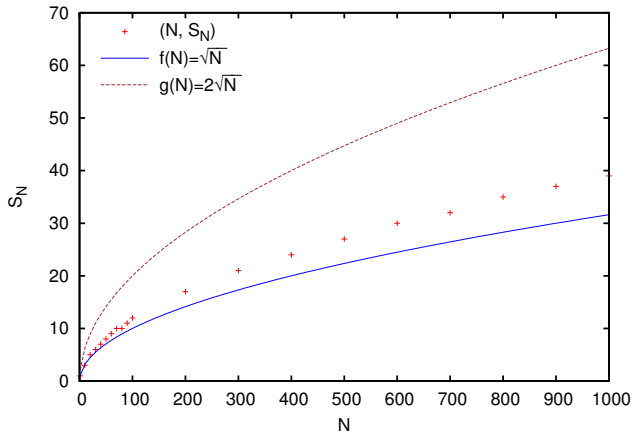


Figure 3: S_N with respect to the growth of N.

THEOREM 4. *Assume that the position of a plan (after sampling-based validation) in the ordered plans with respect to their costs is uniformly distributed. Let N be the number of different join trees in the search space.* [2] *The expected number of steps before the re-optimization procedure terminates is then S_N, where S_N is computed by Equation 1. Moreover, $S_N = O(\sqrt{N})$ by Theorem 3.*

We emphasize that the analysis here only targets worst-case performance, which might be too pessimistic. This is because Procedure 1 only simulates the Case (3) stated in Theorem 2, which is the worst one among the three possible cases. In our experiments, we have found that all queries we tested require less than 10 rounds of iteration, most of which require only 1 or 2 rounds.

Remark: The uniformity assumption in Theorem 4 may not be valid in practice. It is possible that a plan after sampling-based validation (or, a marked ball in terms of Procedure 1) is more likely to be inserted into the front/back half of the queue. Such cases imply that, rather than with an equal chance of overestimation or underestimation, the optimizer tends to overestimate/underestimate the costs of all query plans (for a particular query). This is, however, not impossible. In practice, significant cardinality estimation errors usually appear locally and propagate upwards. Once the error at some join is corrected, the errors in all plans that contain that join will also be corrected. In other words, the correction of the error at a single join can lead to the correction of errors in many candidate plans. In Appendix B, we further present analysis for two extreme cases: all local errors are overestimates/underestimates. To summarize, for left-deep join trees, we have the following two results:

- If all local errors are overestimates, then in the worst case the re-optimization procedure will terminate in at most $m + 1$ steps, where m is the number of joins contained in the query.
- If all local errors are underestimates, then in the worst case re-optimization is expected to terminate in $S_{N/M}$ steps. Here N is the number of different join trees in the optimizer's search space and M is the number of edges in the join graph.

Note that both results are better than the bound stated in Theorem 4. For instance, in the underestimation-only case, if $N = 1000$ and $M = 10$, we have $S_N = 39$ but $S_{N/M} = 12$.

However, in reality, overestimates and underestimates may co-exist. For left-deep join trees, by following the analysis in Appendix B, we can see that such cases sit in between the two extreme cases. Nonetheless, an analysis for plans beyond left-deep trees (e.g., bushy trees) seems to be challenging. We leave this as one possible direction for future work.

3.4 Optimality of the Final Plan

We can think of the re-optimization procedure as progressive adjustments of the optimizer's direction when it explores its search space. The search space depends on the algorithm or strategy used by the optimizer. So does the impact of re-optimization. But we can still have some general conclusions about the optimality of the final plan regardless of the search space.

ASSUMPTION 1. *The cost estimates of plans using sampling-based cardinality refinement are consistent. That is, for any two plans P_1 and P_2, if $\text{cost}^s(P_1) < \text{cost}^s(P_2)$, then $\text{cost}^a(P_1) < \text{cost}^a(P_2)$. Here, $\text{cost}^s(P)$ and $\text{cost}^a(P)$ are the estimated cost based on sampling and the actual cost of plan P, respectively.*

We have the following theorem based on Assumption 1. The proof is included in [1].

THEOREM 5. *Let P_1, ..., P_n be a series of plans generated during the re-optimization procedure. Then $\text{cost}^s(P_n) \leq \text{cost}^s(P_i)$, and thus, by Assumption 1, it follows that $\text{cost}^a(P_n) \leq \text{cost}^a(P_i)$, for $1 \leq i \leq n-1$.*

That is, the plan after re-optimization is guaranteed to be better than the original plan. Nonetheless, it is difficult to conclude that the plans are improved monotonically, namely, in general it is not true that $\text{cost}^s(P_{i+1}) \leq \text{cost}^s(P_i)$, for $1 \leq i \leq n-1$. However, we can prove that this is true if we only have overestimates during re-optimization (proof in [1]):

COROLLARY 3. *Let P_1, ..., P_n be a series of plans generated during the re-optimization procedure. If in the re-optimization procedure only overestimates occur, then $\text{cost}^s(P_{i+1}) \leq \text{cost}^s(P_i)$ for $1 \leq i \leq n-1$.*

Our last result on the optimality of the final plan is in the sense that it is the best among all the plans that are local transformations of the final plan (proof in [1]).

THEOREM 6. *Let P be the final plan the re-optimization procedure returns. For any P' such that P' is a local transformation of P, it holds that $\text{cost}^s(P) \leq \text{cost}^s(P')$.*

3.5 Discussion

We call the final plan returned by the re-optimization procedure the *fixed point* with respect to the initial plan generated by the optimizer. According to Theorem 5, this plan is a *local optimum* with respect to the initial plan. Note that, if $\mathcal{P} = \{P_1, ..., P_n\}$ covers the whole search space, that is, any plan P in the search space is covered by $\mathcal{P}$, then the locally optimal plan is also globally optimal. However, in general, it is difficult to give a definitive answer

[2] Theoretically N could be as large as $O(2^m)$ where m is the number of join operands, assuming that the optimizer uses the bottom-up dynamic programming search strategy. Nonetheless, this may not always be the case. For example, some optimizers use the Cascades framework that leverages a top-down search strategy [18]. An optimizer may even use different search strategies for different types of queries. For example, PostgreSQL will switch from the dynamic programming search strategy to a randomized search strategy based on genetic algorithm, when the number of joins exceeds a certain threshold (12 by default) [2]. For this reason, we choose to characterize the complexity of our algorithm in terms of N rather than m.

to the question that how far away the plan after re-optimization is from the *true* optimal plan. It depends on several factors, including the quality of the initial query plan, the search space covered by re-optimization, and the accuracy of the cost model and sampling-based cardinality estimates.

A natural question is then the impact of the initial plan. Intuitively, it seems that the initial plan can affect both the fixed point and the time it takes to converge to the fixed point. (Note that it is straightforward to prove that the fixed point must exist and be unique, with respect to the given initial plan.) There are also other related interesting questions. For example, if we start with two initial plans with similar cost estimates, would they converge to fixed points with similar costs as well? We leave all these problems as interesting directions for further investigation.

Moreover, the convergence of the re-optimization procedure towards a fixed point can also be viewed as a validation procedure of the costs of the plans $\mathcal{V}$ that can be covered by $\mathcal{P} = \{P_1, ..., P_n\}$. Note that $\mathcal{V}$ is a subset of the whole search space explored by the optimizer, and $\mathcal{V}$ is induced by P_1 — the initial plan that is deemed as optimal by the optimizer. It is also interesting future work to study the relationship between P_1 and $\mathcal{V}$, especially how much of the complete search space can be covered by $\mathcal{V}$.

4. OPTIMIZER "TORTURE TEST"

Evaluating the effectiveness of a query optimizer is challenging. As we mentioned in the introduction, query optimizers have to handle not only common cases but also difficult, corner cases. However, we found it impossible to find well-known public examples of these corner-case queries and the data sets they run on. Regarding this, in this section we create our own data sets and queries based on the well-known fact that many difficult queries are made difficult by correlation in the data [28]. We call it "optimizer torture test" (OTT), given that our goal is to sufficiently challenge the cardinality estimation approaches used by current query optimizers. We next describe the details of OTT.

4.1 Design of the Database and Queries

Since we target cardinality/selectivity estimation, we can focus on queries that only contain selections and joins. In general, a selection-join query q over K relations R_1, ..., R_K can be represented as

$$q = \sigma_F(R_1 \bowtie \cdots \bowtie R_K),$$

where F is a selection predicate as in relational algebra (i.e., a boolean formula). Moreover, we can just focus on equality predicates, i.e., predicates of the form $A = c$ where A is an attribute and c is a constant. Any other predicate can be represented by unions of equality predicates. As a result, we can focus on F of the form

$$F = (A_1 = c_1) \wedge \cdots \wedge (A_K = c_K),$$

where A_k is an attribute of R_k, and $c_k \in \mathrm{Dom}(A_k)$ $(1 \le k \le K)$. Here, $\mathrm{Dom}(A_k)$ is the domain of the attribute A_k.

Based on the above observations, our design of the database and queries is as follows:

(1) We have K relations $R_1(A_1, B_1)$, ..., $R_K(A_K, B_K)$.

(2) We use A_k's for selections and B_k's for joins.

(3) Let $R'_k = \sigma_{A_k=c_k}(R_k)$ for $1 \le k \le K$. The queries of our benchmark are then of the form:

$$R'_1 \bowtie_{B_1=B_2} R'_2 \bowtie_{B_2=B_3} \cdots \bowtie_{B_{K-1}=B_K} R'_K. \quad (2)$$

The remaining question is how to generate data for R_1, ..., R_K so that we can easily control the selectivities for the selection and join predicates. This requires us to consider the joint data distribution for $(A_1, ..., A_K, B_1, ..., B_K)$. A straightforward way could be to specify the contingency table of the distribution. However, there is a subtle issue of this approach: we cannot just generate a large table with attributes A_1, ..., A_K, B_1, ..., B_K and then split it into different relations $R_1(A_1, B_1)$, ..., $R_K(A_K, B_K)$. The reason is that we cannot infer the joint distribution $(A_1, ..., A_K, B_1, ..., B_K)$ based on the (marginal) distributions we *observed* on (A_1, B_1), ..., (A_K, B_K). In Appendix C we further provide a concrete example to illustrate this. This discrepancy between the observed and true distributions calls for a new approach.

4.2 The Data Generation Algorithm

The previous analysis suggests that we can only generate data for each $R_k(A_k, B_k)$ separately and independently, without resorting to their joint distribution. To generate correlated data, we therefore have to make A_k and B_k correlated, for $1 \le k \le K$. Because our goal is to challenge the optimizer's cardinality estimation algorithm, we choose to go to the extreme of this direction: let B_k be the same as A_k. Algorithm 2 presents the details of this idea.

Algorithm 2: Data generation for the OTT database

Input: $\Pr(A_k)$, the distribution of A_k, for $1 \le k \le K$
Output: $R_k(A_k, B_k)$: tables generated, for $1 \le k \le K$

```
for 1 ≤ k ≤ K do
    Pick a seed independently for the random number generator;
    Generate A_k with respect to Pr(A_k);
    Generate B_k = A_k;
end
return R_k(A_k, B_k), 1 ≤ k ≤ K;
```

We are now left with the problem of specifying $\Pr(A_k)$. While $\Pr(A_k)$ could be arbitrary, we should reconsider our goal of sufficiently challenging the optimizer. We therefore need to know some details about how the optimizer estimates selectivities/cardinalities. Of course, different query optimizers have different implementations, but the general principles are similar. In the following, we present the specific technique used by PostgreSQL, which is used in our experimental evaluation in Section 5.

4.2.1 PostgreSQL's Approaches

PostgreSQL stores the following three types of statistics for each attribute A in its `pg_stats` view [3], if the `ANALYZE` command is invoked for the database:

- the number of distinct values $n(A)$ of A;
- most common values (MCV's) of A and their frequency;
- an equal-depth histogram for the other values of A except for the MCV's.

The above statistics can be used to estimate the selectivity of a predicate over a single attribute in a straightforward manner. For instance, for the predicate $A = c$ in our OTT queries, PostgreSQL first checks if c is in the MCV's. If c is present, then the optimizer simply uses the (exact) frequency recorded. Otherwise, the optimizer assumes a uniform distribution over the non-MCV's and estimates the frequency of c based on $n(A)$.

The approach used by PostgreSQL to estimate selectivities for join predicates is more sophisticated. Consider an equal-join predicate $B_1 = B_2$. If MCV's for either B_1 or B_2 are not available, then the optimizer uses an approach first introduced in System R [35] by estimating the reduction factor as $1/\max\{n(B_1), n(B_2)\}$. If, on the other hand, MCV's are available for both B_1 and B_2, then PostgreSQL tries to refine its estimate by first "joining" the two lists of MCV's. For skewed data distributions, this can lead to much better estimates because the join size of the MCV's, which is accurate, will be very close to the actual join size. Other database systems, such as Oracle [7], use similar approaches.

To combine selectivities from multiple predicates, PostgreSQL relies on the well-known attribute-value-independence (AVI) assumption, which assumes that the distributions of values of different attributes are independent.

4.2.2 The Distribution $\Pr(A_k)$ And Its Impact

From the previous discussion we can see that whether $\Pr(A_k)$ is uniform or skewed will have little difference in affecting the optimizer's estimates if MCV's are leveraged, simply because MCV's have recorded the *exact* frequency for those skewed values. We therefore can just let $\Pr(A_k)$ be uniform. We next analyze the impact of this decision by computing the differences between the estimated and actual cardinalities for the OTT queries.

Let us first revisit the OTT queries presented in Equation 2. Note that for an OTT query to be non-empty, the following condition must hold: $B_1 = B_2 = \cdots = B_{K-1} = B_K$. Because we have intentionally set $A_k = B_k$ for $1 \leq k \leq K$, this then implies

$$A_1 = A_2 = \cdots = A_{K-1} = A_K. \quad (3)$$

The query size can thus be controlled by the values of the A's. The query is simply empty if Equation 3 does not hold. In Appendix D, we further present a detailed analysis of the query size when Equation 3 holds. To summarize, we are able to control the difference between the query sizes when Equation 3 holds or not. Therefore, we can make this gap as large as we wish. However, the optimizer will give the same estimate of the query size regardless of if Equation 3 holds or not. In our experiments (Section 5) we further used this property to generate instance OTT queries.

5. EXPERIMENTAL EVALUATION

We present experimental evaluation results of our proposed re-optimization procedure in this section.

5.1 Experimental Settings

We implemented the re-optimization framework in PostgreSQL 9.0.4. The modification to the optimizer is small, limited to several hundred lines of C code, which demonstrates the feasibility of including our framework into current query optimizers. We conducted our experiments on a PC with 2.4GHz Intel dual-core CPU and 4GB memory, and we ran PostgreSQL under Linux 2.6.18.

5.1.1 Databases and Performance Metrics

We used both the standard version and a skewed version [4] of the TPC-H benchmark database, as well as our own OTT database described in Section 4.

TPC-H Benchmark Databases. We used TPC-H databases at the scale of 10GB in our experiments. The generator for skewed TPC-H database uses a parameter z to control the skewness of each column by generating Zipfian distributions. The larger z is, the more skewed the generated data are. $z = 0$ corresponds to a uniform distribution. In our experiments, we generated a skewed database by setting $z = 1$.

OTT Database. We created an instance of the OTT database in the following manner. We first generated a standard 1GB TPC-H database. We then extended the 6 largest tables (*lineitem*, *orders*, *partsupp*, *part*, *customer*, *supplier*) by adding two additional columns A and B to each of them. As discussed in Section 4.2, we populated the extra columns with uniform data. The domain of a column is determined by the number of rows in the corresponding table: if the table contains r rows, then the domain is $\{0, 1, ..., r/100 - 1\}$. In other words, each distinct value in the domain appears roughly 100 times in the generated column. We further created an index on each added column.

Performance Metrics. In our experiments, we measured the following performance metrics for each query on each database:

(1) the original running time of the query;

(2) the number of iterations the re-optimization procedure requires before its termination;

(3) the time spent on the re-optimization procedure;

(4) the total query running time including the re-optimization time.

Based on studies in the previous work [40], in all of our experiments we set the sampling ratio to be 0.05, namely, 5% of the data were taken as samples.

5.1.2 Calibrating Cost Models

The previous work [40] has also shown that, after proper calibration of the cost models used by the optimizer, we could have better estimates of query running times. An interesting question is then: would calibration also improve query optimization?

In our experiments, we also investigated this problem. Specifically, we ran the offline calibration procedure (details in [40]) and replaced the default values of the five cost units (*seq_page_cost*, *random_page_cost*, *cpu_tuple_cost*, *cpu_index_tuple_cost*, and *cpu_operator_cost*) in `postgresql.conf` (i.e., the configuration file of PostgreSQL server) with the calibrated values. In the following, we also report results based on calibrated cost models.

5.2 Results on the TPC-H Benchmark

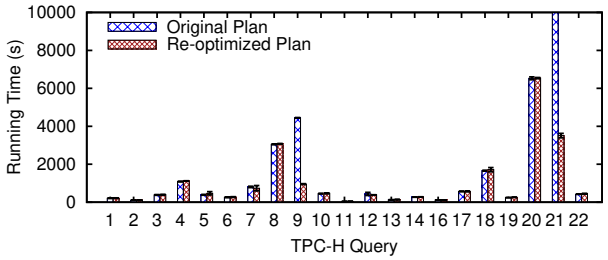
We tested 21 TPC-H queries. (We excluded Q15, which is not supported by our current implementation because it requires to create a view first.) For each TPC-H query, we randomly generated 10 instances. We cleared both the database buffer pool and the file system cache between each run of each query.

5.2.1 Results on Uniform Database

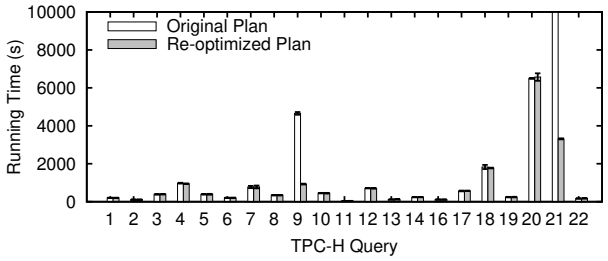
Figure 4 presents the average running times and their standard deviations (as error bars) of these queries over the uniform database.

We have two observations. First, while the running times for most of the queries almost do not change, we can see significant improvement for some queries. For example, as shown in Figure 4(a), even without calibration of the cost units, the average running time of Q9 drops from 4,446 seconds to only 932 seconds, a 79% reduction; more significantly, the average running time of Q21 drops from 20,746 seconds (i.e., almost 6 hours) to 3,508 seconds (i.e., less than 1 hour), a 83% reduction.

Second, calibration of the cost units can sometimes significantly reduce the running times for some queries. For example, comparing Figure 4(a) with Figure 4(b) we can observe that the average



(a) Without calibration of the cost units



(b) With calibration of the cost units

Figure 4: Query running time over uniform 10GB TPC-H database ($z = 0$).

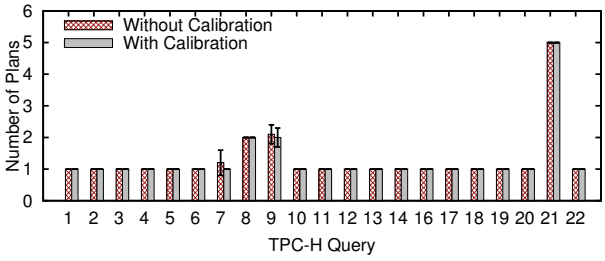


Figure 5: The number of plans generated during re-optimization over uniform 10GB TPC-H database.

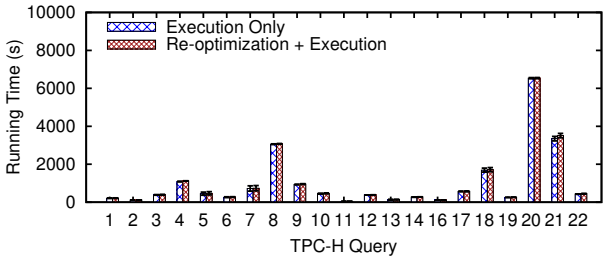
running time of Q8 drops from 3,048 seconds to only 339 seconds, a 89% reduction, by just using calibrated cost units without even invoking the re-optimization procedure.

We further studied the re-optimization procedure itself. Figure 5 presents the number of plans generated during re-optimization. It substantiates our observation in Figure 4: for the queries whose running times were not improved, the re-optimization procedure indeed picked the same plans as those originally chosen by the optimizer. Figure 6 further compares the query running time excluding/including the time spent on re-optimization. For all the queries we tested, re-optimization time is ignorable compared to query execution time, which demonstrates the low overhead of our re-optimization procedure.

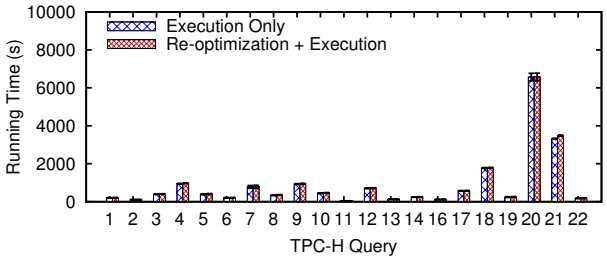
5.2.2 Results on Skewed Database

On the skewed database, we have observed results similar to that on the uniform database. Figure 7 presents the running times of the queries, with or without calibration of the cost units.[3] While it looks quite similar to Figure 4, there is one interesting phenomenon

[3]We notice that Q17 in Figure 7 has a large error bar. The error bars represent variance due to different instances of the query template (different constants in the query). Q17 therefore has a large variance, because we used a skewed TPC-H database. The variance is much smaller when a uniform database is used (see Figure 4).



(a) Without calibration of the cost units



(b) With calibration of the cost units

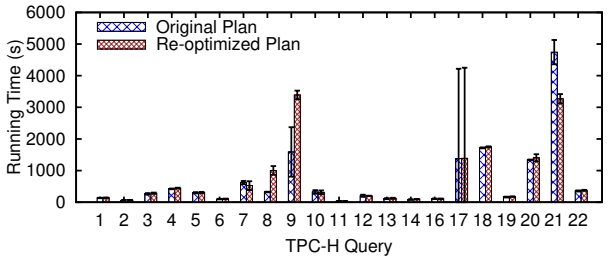
Figure 6: Query running time excluding/including re-optimization time over uniform 10GB TPC-H database ($z = 0$).

not shown before. In Figure 7(a) we see that, without using calibrated cost units, the average running times for Q8 and Q9 actually increase after re-optimization. Recall that in Section 3.4 we have shown the local optimality of the plan returned by the re-optimization procedure (Theorem 5). However, that result is based on the assumption that sampling-based cost estimates are consistent with actual costs (Assumption 1). Here this seems not the case. Nonetheless, after using calibrated cost units, both the running times of Q8 and Q9 were significantly improved (Figure 7(b)).

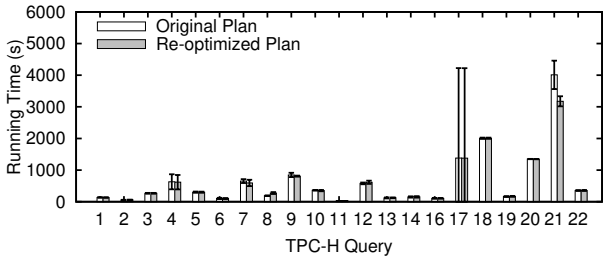
We further present the number of plans considered during re-optimization in Figure 8. Note that re-optimization seems to be more active on skewed data. Figure 9 shows the running times excluding/including the re-optimization times of the queries. Again, the additional overhead of re-optimization is trivial.

5.2.3 Discussion

While one might expect the chance for re-optimization to generate a better plan is higher on skewed databases, our experiments show that this may not be the case, at least for TPC-H queries. There are several different situations, though. First, if a query is too simple, then there is almost no chance for re-optimization. For example, Q1 contains no join, whereas Q16 and Q19 involve only one join so only local transformations are possible. Second, the final plan returned by the re-optimization procedure heavily relies on the initial plan picked by the optimizer, which is the seed or starting point where re-optimization originates. Note that, even if the optimizer has picked an inefficient plan, re-optimization cannot help if the estimated cost of that plan is not significantly erroneous. One question is if this is possible: the optimizer picks an inferior plan whose cost estimate is correct? This actually could happen because the optimizer may (incorrectly) overestimate the costs of the other plans in its search space. Another subtle point is that the inferior plan might be robust to certain degree of errors in cardinality estimates. Previous work has reported this phenomenon by noticing that the plan diagram (i.e., all possible plans and their governed optimality areas in the selectivity space) is dominated by just a couple of query plans [33].



(a) Without calibration of the cost units



(b) With calibration of the cost units

Figure 7: Query running time over skewed 10GB TPC-H database ($z = 1$).

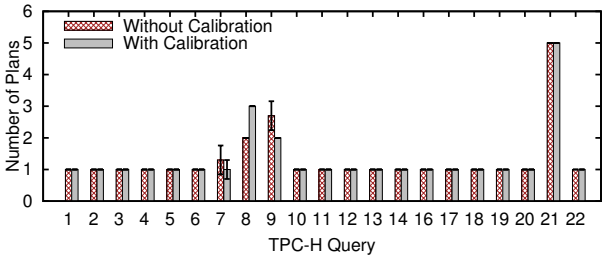


Figure 8: The number of plans generated during re-optimization over skewed 10GB TPC-H database.

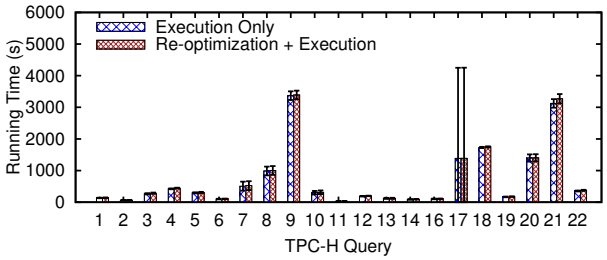
In summary, the effectiveness of re-optimization depends on factors that are out of the control of the re-optimization procedure itself. Nevertheless, we have observed intriguing improvement for some long-running queries by applying re-optimization, especially after calibration of the cost units.

5.3 Results of the Optimizer Torture Test

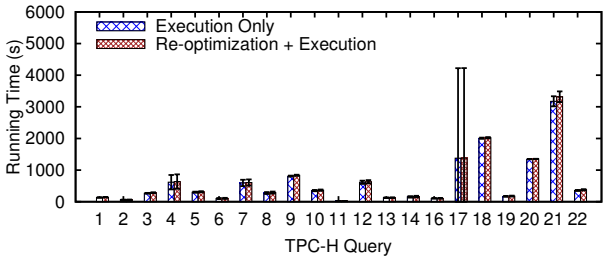
We created queries following our design of the OTT in Section 4.1. Specifically, if a query contains n tables (i.e., $n-1$ joins), we let m of the selections be $A = 0$ ($A = 1$), and let the remaining $n - m$ selections be $A = 1$ ($A = 0$). We generated two sets of queries: (1) $n = 5$ (4 joins), $m = 4$; and (2) $n = 6$ (5 joins), $m = 4$. Note that the maximal non-empty sub-queries then contain 3 joins over 4 tables with result size of roughly $100^4 = 10^8$ rows.[4] However, the size of each (whole) query is 0. So we would like to see the ability of the optimizer as well as the re-optimization procedure to identify the empty/non-empty sub-queries.

Figure 10 and 11 present the running times of the 4-join and 5-join queries, respectively. We generated in total 10 4-join queries and 30 5-join queries. Note that the y-axes are in log scale and we do not show queries that finish in less than 0.1 second. As we can see, sometimes the optimizer failed to detect the existence of empty sub-queries: it generated plans where empty join predicates were evaluated after the non-empty ones. The running times of these queries were then hundreds or even thousands of seconds. On the other hand, the re-optimization procedure did an almost perfect job in detecting empty joins, which led to very efficient query plans where the empty joins were evaluated first: all the queries after re-optimization finished in less than 1 second.

[4]Recall that a non-empty query must have equal A's (Equation 3) and we generated data with roughly 100 rows per distinct value (Section 5.1).



(a) Without calibration of the cost units



(b) With calibration of the cost units

Figure 9: Query running time excluding/including re-optimization time over skewed 10GB TPC-H database ($z = 1$).

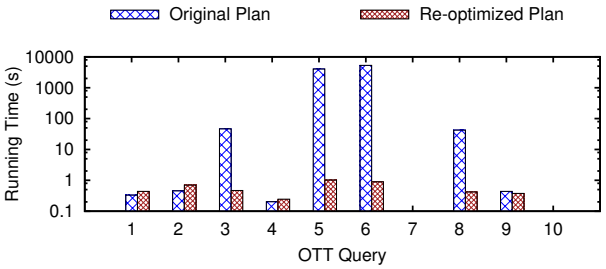
We also did similar studies regarding the number of plans generated during re-optimization and the time it consumed. Due to space constraints, we refer the readers to Appendix A.1 for the details.

One might argue that the OTT queries are really contrived: these queries are hardly to see in real-world workloads. While this might be true, we think these queries serve our purpose as exemplifying extremely hard cases for query optimization. Note that hard cases are not merely long-running queries: queries as simple as sequentially scanning huge tables are long-running too, but there is nothing query optimization can help with. Hard cases are queries where efficient execution plans do exist but it might be difficult for the optimizer to find them. The OTT queries are just these instances. Based on the experimental results of the OTT queries, re-optimization is helpful to give the optimizer second chances if it initially made a bad decision.

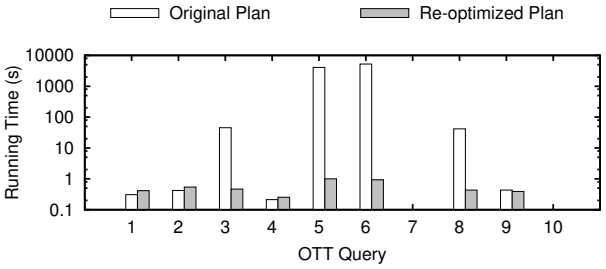
Another concern is if commercial database systems could do a better job on the OTT queries. In regard of this, we also ran the OTT over two major commercial database systems. The performance is very similar to that of PostgreSQL (Figure 12 and 13). We therefore speculate that commercial systems could also benefit from our re-optimization technique proposed in this paper.

5.3.1 A Note on Multidimensional Histograms

Note that even using multidimensional histograms (e.g., [10, 31, 32]) may not be able to detect the data correlation presented in the OTT queries, unless the buckets are so fine-grained that the exact joint distributions are retained. To understand this, let us consider the following example.

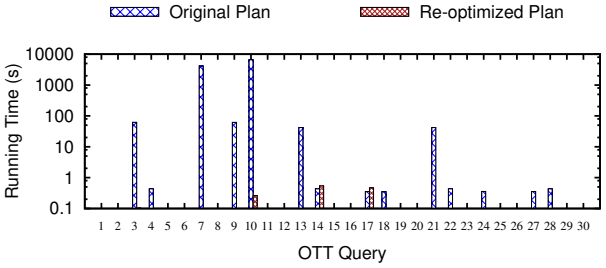


(a) Without calibration of the cost units

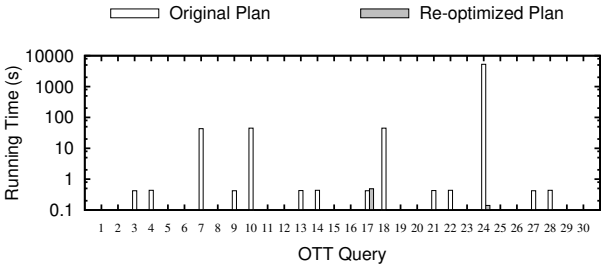


(b) With calibration of the cost units

Figure 10: Query running times of 4-join queries.



(a) Without calibration of the cost units



(b) With calibration of the cost units

Figure 11: Query running times of 5-join queries.

EXAMPLE 2. *Following our design of OTT, suppose that now we only have two tables $R_1(A_1, B_1)$ and $R_2(A_2, B_2)$. Moreover, suppose that each A_k (and thus B_k) contains $m = 2l$ distinct values, and we construct (perfect) 2-dimensional histograms on (A_k, B_k) $(k = 1, 2)$. Each dimension is evenly divided into $\frac{m}{2} = l$ intervals, so each histogram contains l^2 buckets. The joint distribution over (A_k, B_k) estimated by using the histogram is then:*

$$\begin{cases} \Pr(2r-2 \le A_k < 2r, 2r-2 \le B_k < 2r) = \frac{1}{l}, & 1 \le r \le l; \\ \Pr(a_l \le A_k < a_2, b_1 \le B_k < b_2) = 0, & \text{otherwise.} \end{cases}$$

For instance, if $m = 100$, then $l = 50$. So we have $\Pr(0 \le A_k < 2, 0 \le B_k < 2) = \cdots = \Pr(98 \le A_k < 100, 98 \le B_k < 100) = \frac{1}{50}$, while all the other buckets are empty. On the other hand, the actual joint distribution is

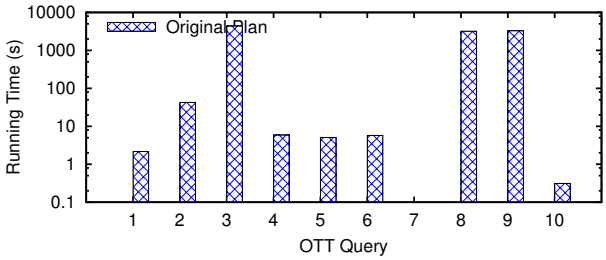
$$\Pr(A_k = a, B_k = b) = \begin{cases} \frac{1}{m}, & \text{if } a = b; \\ 0, & \text{otherwise.} \end{cases}$$

Now, let us consider the selectivities for two OTT queries:

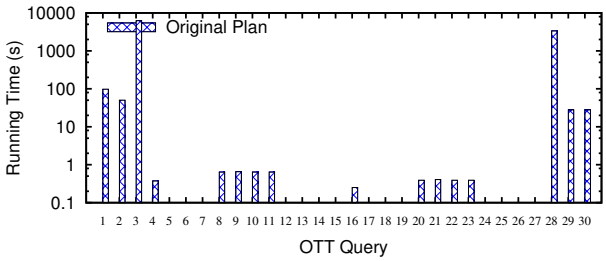
(q_1) $\sigma_{A_1=0 \wedge A_2=1 \wedge B_1=B_2}(R_1 \times R_2)$;

(q_2) $\sigma_{A_1=0 \wedge A_2=0 \wedge B_1=B_2}(R_2 \times R_2)$.

We know that q_1 is empty but q_2 is not. However, the estimated selectivity (and thus cardinality) of q_1 and q_2 is the same by using the

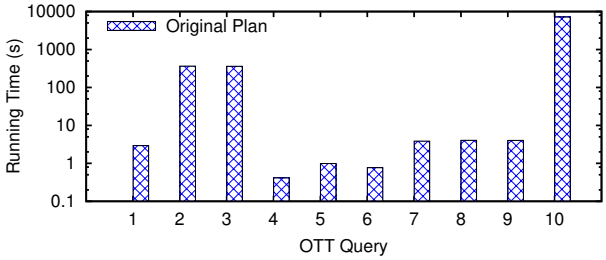


(a) 4-join OTT queries

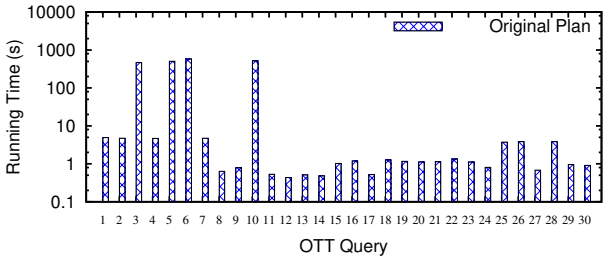


(b) 5-join OTT queries

Figure 12: Query running times of the OTT queries on the commercial database system A.



(a) 4-join OTT queries



(b) 5-join OTT queries

Figure 13: Query running times of the OTT queries on the commercial database system B.

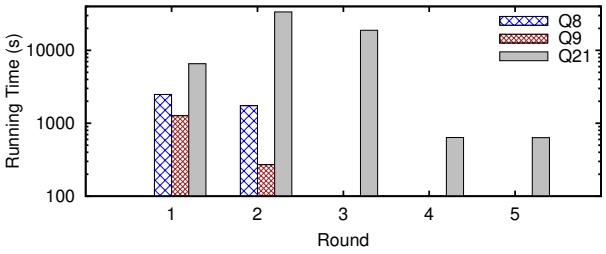


Figure 14: Running time of plans generated in re-optimization for TPC-H queries ($z = 0$) without calibration of cost units.

2-dimensional histogram, because of the assumption used by histograms that data inside each bucket is uniformly distributed.[5] *With the setting used in our experiments, $m = 100$ and thus $l = 50$. So each 2-dimensional histogram contains $l^2 = 2,500$ buckets. However, even such detailed histograms cannot help the optimizer distinguish empty joins from nonempty ones. Furthermore, note that our conclusion is independent of m, while the number of buckets increases quadratically in m. For instance, when $m = 10^4$ which means we have histograms containing 2.5×10^7 buckets, the optimizer still cannot rely on the histograms to generate efficient query plans for OTT queries.*

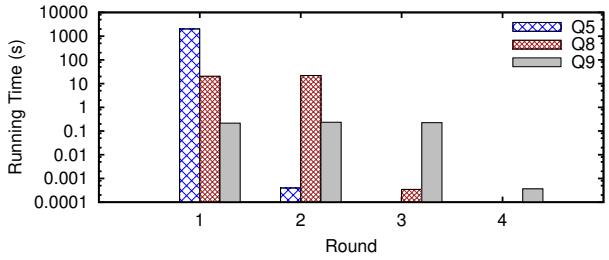
5.4 Effectiveness of Iteration

One interesting further question is how much benefit iteration brings in. Since running plans over samples incurs additional cost, we may wish to stop the iteration as early as possible rather than wait until its convergence. To investigate this, we also tested execution times for plans generated during re-optimization on the original databases. We focus on queries for which at least two plans were generated. Figure 14 presents typical results for TPC-H queries Q8, Q9, and Q21, and Figure 15 presents typical results for the OTT queries. For each query, the plan in the first round is the original one returned by the optimizer.

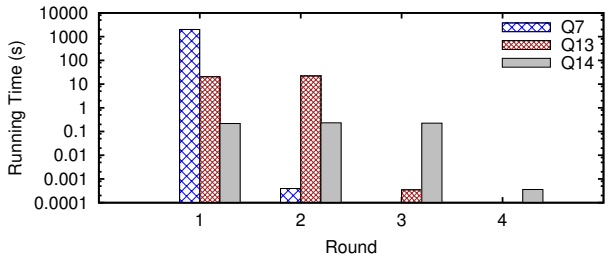
We have the following observations. While the plan returned in the second round of iteration (i.e., the first different plan returned by re-optimization) often dramatically reduces the query execution time, it is not always the case. Sometimes it takes additional rounds before an improved plan is found while the plans generated in between have similar execution times (e.g., 4-join OTT query Q8 and 5-join OTT query Q13). In the case of TPC-H Q21, the execution times of intermediate plans generated during re-optimization are even not non-increasing. The plans returned by the optimizer in the second and third round are even much worse than the original plan. (Note that the y-axis of Figure 14 is in log scale.) Fortunately, by continuing with the iteration, we can eventually reach a plan that is much better than the original one.

The above observations give rise to the question why worse plans might be generated during re-optimization. To understand this, note that in the middle of re-optimization, the plans returned by the optimizer are based on statistics that are *partially* validated by sampling. The optimizer is likely to generate an "optimal" plan due to underestimating the costs of joins that have not been covered by plans generated in previous rounds. This phenomenon has also been observed in previous work [13]. It is the essential reason that we cannot make Theorem 5 stronger. As long as there exist uncovered joins, there is no guarantee on the plan returned by the optimizer. Only upon the convergence of re-optimization we can say that the final plan is locally optimal.

[5]It is easy to verify that the selectivity estimates are $\hat{s}_1 = \hat{s}_2 = \frac{1}{8l^2}$.



(a) 4-join queries



(b) 5-join queries

Figure 15: Running time of plans generated in re-optimization for the OTT queries without calibration of cost units.

Nevertheless, in practice we can still have various strategies to control the overhead of re-optimization. For example, we can stop re-optimization if it does not converge after a certain number of rounds, or if the time spent on re-optimization has reached some timeout threshold. We then simply return the best plan among the plans generated so far, based on their cost estimates by using refined cardinality estimates from sampling [40]. As another option, it might even be worth considering not doing re-optimization at all if the estimated query execution time is shorter than some threshold, or only doing it if we run that plan and get past that threshold without being close to done.

6. RELATED WORK

Query optimization has been studied for decades, and we refer the readers to [12] and [22] for surveys in this area.

Cardinality estimation is a critical problem in cost-based query optimization, and has triggered extensive research in the database community. Approaches for cardinality estimation in the literature are either static or dynamic. Static approaches usually rely on various statistics that are collected and maintained periodically in an off-line manner, such as histograms (e.g., [23, 32]), samples (e.g., [11, 20, 27]), sketches (e.g., [5, 34]), or even graphical models (e.g. [17, 37]). In practice, approaches based on histograms are dominant in the implementations of current query optimizers. However, histogram-based approaches have to rely on the notorious attribute-value-independence (AVI) assumption, and they often fail to capture data correlations, which could result in significant errors in cardinality estimates. While variants of histograms (in particular, multidimensional histograms, e.g., [10, 31, 32]) have been proposed to overcome the AVI assumption, they suffer from significantly increased overhead on large databases. Meanwhile, even if we can afford the overhead of using multidimensional histograms, they are still insufficient in many cases, as we discussed in Section 5.3.1. Compared with histogram-based approaches, sampling is better at capturing data correlation. One reason for this is that sampling evaluates queries on real rather than summarized

data. There are many sampling algorithms, and in this paper we just picked a simple one (see [38] for a recent survey). We do not try to explore more advanced sampling techniques (e.g. [16]), which we believe could further improve the quality of cardinality estimates.

On the other hand, dynamic approaches further utilize information gathered during query runtime. Approaches in this direction include dynamic query plans (e.g., [14, 19]), parametric query optimization (e.g. [24]), query feedback (e.g., [8, 36]), mid-query re-optimization (e.g. [25, 30]), and quite recently, plan bouquets [15]. The ideas behind dynamic query plans and parametric query optimization are similar: rather than picking one single optimal query plan, all possible optimal plans are retained and the decision is deferred until runtime. Both approaches suffer from the problem of combinatorial explosion and are usually used in contexts where expensive pre-compilation stages are affordable. The recent development of plan bouquets [15] is built on top of parametric query optimization so it may also incur a heavy query compilation stage.

Meanwhile, approaches based on query feedback record statistics of past queries and use this information to improve cardinality estimates for future queries. Some of these approaches have been adopted in commercial systems such as IBM DB2 [36] and Microsoft SQL Server [8]. Nonetheless, collecting query feedback incurs additional runtime overhead as well as storage overhead of ever-growing volume of statistics.

The most relevant work in the literature is the line along mid-query re-optimization [25, 30]. The major difference is that re-optimization was previously carried out at runtime over the actual database rather than at query compilation time over the samples. The main issue is the trade-off between the overhead spent on re-optimization and the improvement on the query plan. In the introduction, we have articulated the pros and cons of both techniques. In some sense, our approach can be thought of as a "dry run" of runtime re-optimization. But it is much cheaper because it is performed over the sampled data. As we have seen, cardinality estimation errors due to data correlation can be caught by the sample runs. So it is perhaps an overkill to detect these errors at runtime by running the query over the actual database. Sometimes optimizers make mistakes that involve a bad ordering that results in a larger than expected result from a join or a selection. It is true that runtime re-optimization can detect this, but this may require the query evaluator to do a substantial amount of work before it is detected. For example, if the data is fed into an operator in a non-random order, because of an unlucky ordering the fact that the operator has a much larger than expected result may not be obvious until a substantial portion of the input has been consumed and substantial system resources have been expended. Furthermore, it is non-trivial to stop an operator in mid-flight and switch to a different plan — for this reason most runtime query re-optimization deals with changing plans at pipeline boundaries. Running a bad plan until a pipeline completes might be very expensive.

Nonetheless, we are not claiming that we should just use our approach. Rather, our approach is complementary to runtime re-optimization techniques: it is certainly possible to apply those techniques on the plan returned by our approach. The hope is that, after applying our approach, the chance that we need to invoke more costly runtime re-optimizaton techniques can be significantly reduced. This is similar to the "alerter" idea that has been used in physical database tuning [9], where a lightweight "alerter" is invoked to determine the potential performance gain before the more costly database tuning advisor is called. There is also some recent work on applying runtime re-optimization to Map-Reduce based systems [26]. The "pilot run" idea explored in this work is close to our motivation, where the goal is also to avoid starting with a bad plan by collecting more accurate statistics via scanning a small amount of samples. However, currently "pilot run" only collects statistics for leaf tables. It is then interesting to see if "pilot run" could be further enhanced by considering our technique.

In some sense, our approach sits between existing static and dynamic approaches. We combine the advantage of lower overheads from static approaches and the advantage of more optimization opportunities from dynamic approaches. This compromise leads to a lightweight query re-optimization procedure that could bring up better query plans. However, this also unavoidably leads to the co-existence of both histogram-based and sampling-based cardinality estimates during re-optimization, which may cause inconsistency issues [29]. Somehow, this is a general problem when different types of cardinality estimation techniques are used together. For example, an optimizer that also uses multi-dimensional histograms (in addition to one-dimensional histograms) may also have to handle inconsistencies. Previous runtime re-optimization techniques are also likely to suffer similar issues. In this sense, this is an orthogonal problem and we do not try to address it in this paper. In spite of that, it is interesting future work to see how much more improvement we can get if we further incorporate the approach based on the maximum entropy principle [29] into our re-optimization framework to resolve inconsistency in cardinality estimates.

Finally, we note that we are not the first that investigates the idea of incorporating sampling into query optimization. Ilyas et al. proposed using sampling to detect data correlations and then collecting joint statistics for those correlated data columns [21]. However, this seems to be insufficient if data correlation is caused by specific selection predicates, such as those OTT queries used in our experiments. Babcock and Chaudhuri also investigated the usage of sampling in developing a robust query optimizer [6]. While robustness is another interesting goal for query optimizer, it is beyond the scope of this paper.

7. CONCLUSION

In this paper, we studied the problem of incorporating sampling-based cardinality estimates into query optimization. We proposed an iterative query re-optimization procedure that supplements the optimizer with refreshed cardinality estimates via sampling and gives it second chances to generate better query plans. We show the efficiency and effectiveness of this re-optimization procedure both theoretically and experimentally.

There are several directions worth further exploring. First, as we have mentioned, the initial plan picked by the optimizer may have great impact on the final plan returned by re-optimization. While it remains interesting to study this impact theoretically, it might also be an intriguing idea to think about varying the way that the query optimizer works. For example, rather than just returning one plan, the optimizer could return several candidates and let the re-optimization procedure work on each of them. This might make up for the potentially bad situation currently faced by the re-optimization procedure that it may start with a bad seed plan. Second, the re-optimization procedure itself could be further improved. As an example, note that in this paper we let the optimizer unconditionally accept cardinality estimates by sampling. However, sampling is by no means perfect. A more conservative approach is to consider the uncertainty of the cardinality estimates returned by sampling as well. The previous work [41] has investigated the problem of quantifying uncertainty in sampling-based query running time estimation. It is very interesting to study a combination of that framework with the re-optimization procedure proposed in this paper. We leave all these as promising areas for future work.

Acknowledgements. We thank the anonymous reviewers for their valuable comments, and we thank Heng Guo for his help with the proof of Theorem 3. This work was supported in part by a Google Focus Award.

8. REFERENCES

[1] http://arxiv.org/abs/1601.05748.
[2] http://www.postgresql.org/docs/9.0/static/runtime-config-query.html.
[3] http://www.postgresql.org/docs/9.0/static/view-pg-stats.html.
[4] Skewed tpc-h data generator. ftp://ftp.research.microsoft.com/users/viveknar/TPCDSkew/.
[5] N. Alon, P. B. Gibbons, Y. Matias, and M. Szegedy. Tracking join and self-join sizes in limited storage. In *PODS*, 1999.
[6] B. Babcock and S. Chaudhuri. Towards a robust query optimizer: A principled and practical approach. In *SIGMOD*, pages 119–130, 2005.
[7] W. Breitling. Joins, skew and histograms. http://www.centrexcc.com/Joins,SkewandHistograms.pdf.
[8] N. Bruno and S. Chaudhuri. Exploiting statistics on query expressions for optimization. In *SIGMOD*, pages 263–274, 2002.
[9] N. Bruno and S. Chaudhuri. To tune or not to tune? A lightweight physical design alerter. In *VLDB*, pages 499–510, 2006.
[10] N. Bruno, S. Chaudhuri, and L. Gravano. Stholes: A multidimensional workload-aware histogram. In *SIGMOD*, pages 211–222, 2001.
[11] M. Charikar, S. Chaudhuri, R. Motwani, and V. R. Narasayya. Towards estimation error guarantees for distinct values. In *PODS*, pages 268–279, 2000.
[12] S. Chaudhuri. An overview of query optimization in relational systems. In *PODS*, pages 34–43, 1998.
[13] S. Chaudhuri, V. R. Narasayya, and R. Ramamurthy. A pay-as-you-go framework for query execution feedback. *PVLDB*, 1(1):1141–1152, 2008.
[14] R. L. Cole and G. Graefe. Optimization of dynamic query evaluation plans. In *SIGMOD*, pages 150–160, 1994.
[15] A. Dutt and J. R. Haritsa. Plan bouquets: query processing without selectivity estimation. In *SIGMOD*, 2014.
[16] C. Estan and J. F. Naughton. End-biased samples for join cardinality estimation. In *ICDE*, 2006.
[17] L. Getoor, B. Taskar, and D. Koller. Selectivity estimation using probabilistic models. In *SIGMOD*, 2001.
[18] G. Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.
[19] G. Graefe and K. Ward. Dynamic query evaluation plans. In *SIGMOD Conference*, pages 358–366, 1989.
[20] P. J. Haas, J. F. Naughton, S. Seshadri, and A. N. Swami. Selectivity and cost estimation for joins based on random sampling. *J. Comput. Syst. Sci.*, 52(3):550–569, 1996.
[21] I. F. Ilyas, V. Markl, P. J. Haas, P. Brown, and A. Aboulnaga. CORDS: automatic discovery of correlations and soft functional dependencies. In *SIGMOD*, pages 647–658, 2004.
[22] Y. E. Ioannidis. Query optimization. *ACM Comput. Surv.*, 28(1):121–123, 1996.
[23] Y. E. Ioannidis. The history of histograms (abridged). In *VLDB*, pages 19–30, 2003.
[24] Y. E. Ioannidis, R. T. Ng, K. Shim, and T. K. Sellis. Parametric query optimization. In *VLDB*, pages 103–114, 1992.
[25] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD*, pages 106–117, 1998.
[26] K. Karanasos, A. Balmin, M. Kutsch, F. Ozcan, V. Ercegovac, C. Xia, and J. Jackson. Dynamically optimizing queries over large scale data platforms. In *SIGMOD*, pages 943–954, 2014.
[27] R. J. Lipton, J. F. Naughton, and D. A. Schneider. Practical selectivity estimation through adaptive sampling. In *SIGMOD*, pages 1–11, 1990.
[28] G. Lohman. Is query optimization a "solved" problem? http://wp.sigmod.org/?p=1075.
[29] V. Markl, P. J. Haas, M. Kutsch, N. Megiddo, U. Srivastava, and T. M. Tran. Consistent selectivity estimation via maximum entropy. *VLDB J.*, 16(1):55–76, 2007.
[30] V. Markl, V. Raman, D. E. Simmen, G. M. Lohman, and H. Pirahesh. Robust query processing through progressive optimization. In *SIGMOD*, pages 659–670, 2004.
[31] M. Muralikrishna and D. J. DeWitt. Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. In *SIGMOD*, pages 28–36, 1988.
[32] V. Poosala and Y. E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *VLDB*, 1997.
[33] N. Reddy and J. R. Haritsa. Analyzing plan diagrams of database query optimizers. In *VLDB*, 2005.
[34] F. Rusu and A. Dobra. Sketches for size of join estimation. *ACM Trans. Database Syst.*, 33(3), 2008.
[35] 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*, 1979.
[36] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO - DB2's learning optimizer. In *VLDB*, 2001.
[37] K. Tzoumas, A. Deshpande, and C. S. Jensen. Lightweight graphical models for selectivity estimation without independence assumptions. *PVLDB*, 4(11):852–863, 2011.
[38] D. Vengerov, A. C. Menck, M. Zaït, and S. Chakkappen. Join size estimation subject to filter conditions. *PVLDB*, 8(12):1530–1541, 2015.
[39] W. Wu, Y. Chi, H. Hacigümüs, and J. F. Naughton. Towards predicting query execution time for concurrent and dynamic database workloads. *PVLDB*, 6(10):925–936, 2013.
[40] W. Wu, Y. Chi, S. Zhu, J. Tatemura, H. Hacigümüs, and J. F. Naughton. Predicting query execution time: Are optimizer cost models really unusable? In *ICDE*, pages 1081–1092, 2013.
[41] W. Wu, X. Wu, H. Hacigümüs, and J. F. Naughton. Uncertainty aware query execution time prediction. *PVLDB*, 7(14):1857–1868, 2014.

APPENDIX

A. ADDITIONAL RESULTS

In this section, we present additional experimental results for the OTT queries. We also performed the same experiments for TPC-DS queries, and we report the results here.

A.1 Additional Results on OTT Queries

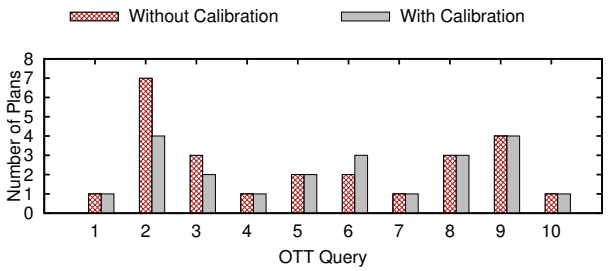
For OTT queries, we also did similar studies regarding the number of plans generated during re-optimization and the time it consumed. Figure 16 presents the number of plans generated during the re-optimization procedure for the 4-join and 5-join OTT queries. Figure 17 and 18 further present the comparisons of the running times by excluding or including re-optimization times for these queries (the y-axes are in log scale).

A.2 Results on the TPC-DS Benchmark

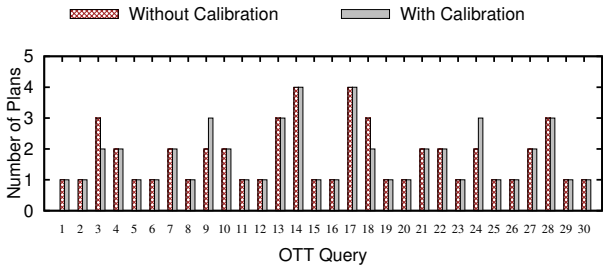
We conducted the same experiments on the TPC-DS benchmark. Specifically, we use a subset containing 29 TPC-DS queries that are supported by PostgreSQL and our current implementation, and we use a 10GB TPC-DS database. The experiments were performed on a PC with 3.4GHz Intel 4-core CPU and 8GB memory, and we ran PostgreSQL under Linux 2.6.32.

We again use a sampling ratio of 5%. Figure 19 presents the execution time of each TPC-DS query. The time spent on running query plans over the samples is again trivial and therefore is not plotted. Figure 20 further presents the number of plans generated during re-optimization.[6]

[6]The results we show here are based on the TPC-DS database with only indexes on primary keys of the tables. The TPC-DS benchmark specification does not recommend any indexes, which potentially could have impact on the optimizer's choices. Regarding this,

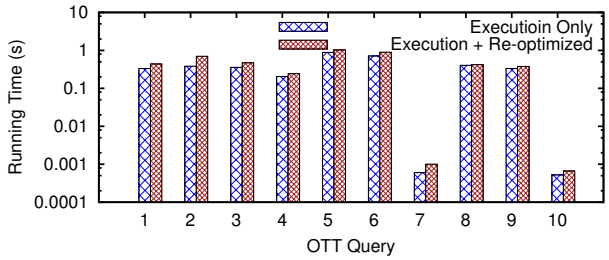


(a) 4-join queries

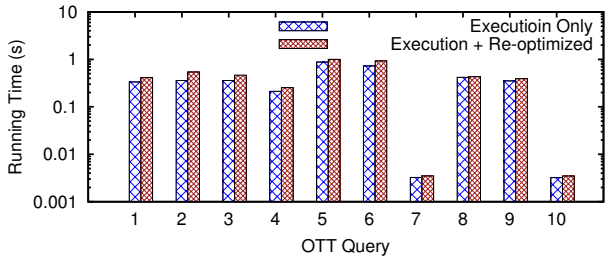


(b) 5-join queries

Figure 16: The number of plans generated during re-optimization for the OTT queries.

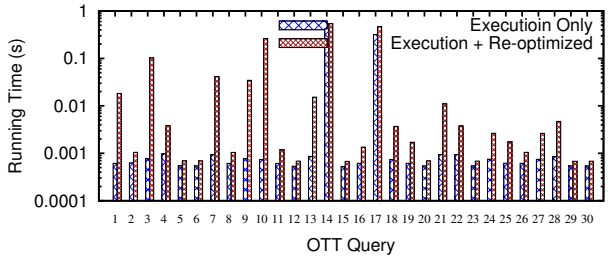


(a) Without calibration of the cost units

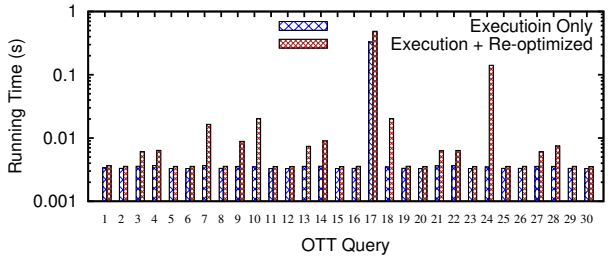


(b) With calibration of the cost units

Figure 17: Query running time excluding/including re-optimization time for the 4-join OTT queries.



(a) Without calibration of the cost units



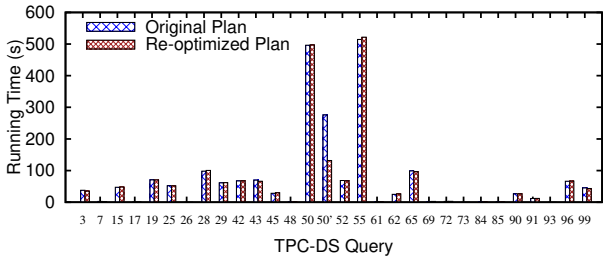
(b) With calibration of the cost units

Figure 18: Query running time excluding/including re-optimization time for the 5-join OTT queries.

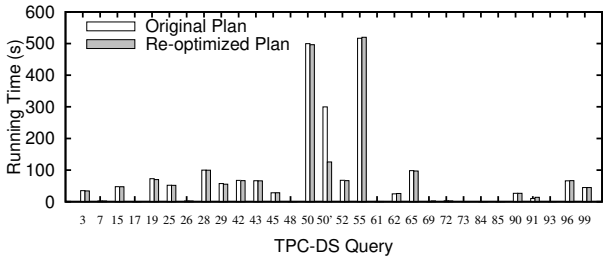
Unfortunately, we do not observe remarkable improvement over the TPC-DS queries we tested by using re-optimization. There are several reasons. First, the majority of the queries are short-running ones, though some of them are quite complicated in terms of their query plans. There is little room for improvement on these queries. Second, for the four relatively long-running queries Q28, Q50, Q55, and Q65 (with running time above 100 seconds), three of them (Q28, Q55, and Q65) are a bit trivial for re-optimization to have impact. Q28 only accesses one table, while Q55 joins a fact table with two small dimension tables. Q65 involves joins between a fact table and three small dimension tables, but its execution time is dominated by a self-join over the fact table. Meanwhile, after checking the plans, we found that all cardinality estimates are on track. We therefore cannot observe a plan change for any of these three queries (see Figure 20). Third, even if there are significant errors in cardinality estimates, they may not lead to significant improvement on the query plan. This is the case for Q50, which involves joins over two fact tables and three dimension tables. The execution time is then dominated by the join between the two fact tables. The optimizer's cardinality estimate for that join is indeed inaccurate, and the error was successfully detected by using sampling. As a result, re-optimization did generate different access paths for the rest of the plan. However, the execution time of the plan remains to be dominated by that huge join, and the remaining execution time after that dominant join was done is trivial.

We actually further tweaked Q50 by modifying the predicates over the dimension tables so that the selectivity estimates were changed. By doing this, we did identify cases where re-optimization significantly reduced query execution times. We show one such variant of Q50 (i.e., Q50') in Figure 19. The execution time of Q50' dropped from around 300 seconds to around 130 seconds after re-optimization, a 57% reduction. Tweaking the other queries may also give more opportunities for re-optimization. Another way is to further change the distribution that governs the underlying data generation. We do not pursue these options any further, for it remains debatable that these variants might just be contrived. A more systematic way could be to identify a particular class of queries that the current optimizers do not handle well, figure out how to handle these queries, then move on to find other kinds of queries that are still not handled well. We suspect (and hope) that this is one way optimizers will improve over time — our work on creating the "optimizer torture test" in Section 4 is one step in this direction. We leave the study of other types of "difficult" queries as important direction for future work.

we actually further tried to build indexes on columns that are referenced by the queries. However, the results we obtained are very similar to the ones we present here.



(a) Without calibration of the cost units



(b) With calibration of the cost units

Figure 19: Query running time over 10GB TPC-DS database.

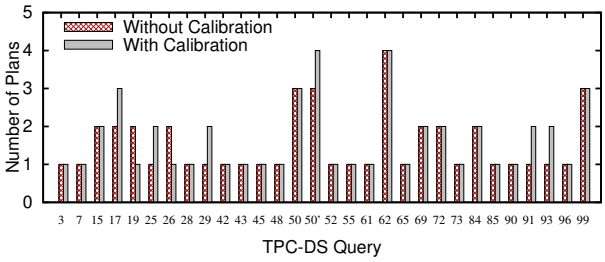


Figure 20: The number of plans generated during re-optimization over 10GB TPC-DS database.

B. ADDITIONAL ANALYSIS

Continuing with our study of the efficiency of the re-optimization procedure in Section 3.3, we now consider two special cases where the optimizer significantly overestimates or underestimates the selectivity of a particular join in the returned optimal plan. We focus our discussion on left-deep join trees. Before we proceed, we need the following assumption of cost functions.

ASSUMPTION 2. *Cost functions used by the optimizer are monotonically non-decreasing functions of input cardinalities.*

We are not aware of a cost function that does not conform to this assumption, though.

B.1 Overestimation Only

Suppose that the error is an overestimate, that is, the actual cardinality is smaller than the one estimated by the optimizer. Let that join operator be O, and let the corresponding subtree rooted at O be $\mathrm{tree}(O)$. Now consider a candidate plan P in the search space, and let $\mathrm{cost}(P)$ be the estimated cost of P. Note that the following property holds:

LEMMA 2. $\mathrm{cost}(P)$ *is affected (i.e., subject to change) only if* $\mathrm{tree}(O)$ *is a subtree of* $\mathrm{tree}(P)$.

Moreover, under Assumption 2, the refined cost estimate $\mathrm{cost}'(P)$ satisfies $\mathrm{cost}'(P) < \mathrm{cost}(P)$ because of the overestimate. Therefore, if we let the set of all such plans P be $\mathcal{P}$, then the next optimal plan picked by the optimizer must be from $\mathcal{P}$. We thus can prove the following result:

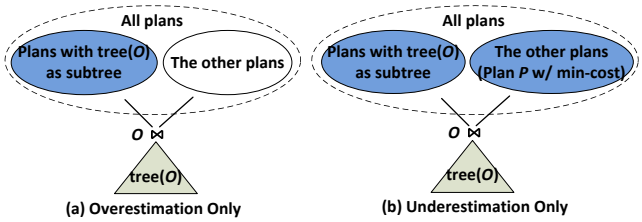


Figure 21: Comparison of the space of candidate optimal plans when local estimation errors are overestimation-only or underestimation-only. Candidate optimal plans are shaded.

THEOREM 7. *Suppose that we only consider left-deep join trees. Let m be the number of joins in the query. If all estimation errors are overestimates, then in the worst case the re-optimization procedure will terminate in at most $m+1$ steps.*

The proof of Theorem 7 is included in [1]. We emphasize that $m+1$ is a worst-case upper bound. The intuition behind Theorem 7 is that, for left-deep join trees, the validated subtree belonging to the final re-optimized plan can grow by at least one more level (i.e., with one more validated join) in each re-optimization step.

B.2 Underestimation Only

Suppose that, on the other hand, the error is an underestimate, that is, the actual cardinality is larger than the one estimated by the optimizer. Then things become more complicated: not only those plans that contain the subtree rooted at the erroneous node but the plan with the lowest estimated cost in the rest of the search space (which is not affected by the estimation error) also has the potential of being the optimal plan, after the error is corrected (see Figure 21). We next again focus on left-deep trees and present some analysis in such context.

Suppose that the join graph G contains M edges. We partition the join trees based on their first joins, which must be the edges of G. So we have M partitions in total.

Let s_i be the state when a plan returned by the optimizer uses the i-th edge in G ($1 \le i \le M$) as its first join. The re-optimization procedure can then be thought of as transitions between these M states, i.e., a Markov chain. Assume that the equilibrium state distribution $\pi(s_i)$ of this Markov chain exists, subject to

$$\sum_{i=1}^{M} \pi(s_i) = 1.$$

$\pi(s_i)$ therefore represents the probability that a plan generated in the re-optimization procedure would have its first join in the partition i ($1 \le i \le M$). We can then estimate the expected number of steps before the procedure terminates as

$$S = \sum_{i=1}^{M} \pi(s_i) \cdot N_i,$$

where N_i is the number of steps/transitions confined in the partition i before termination. Since we only consider left-deep trees, $N_i = N_j$ for $1 \le i, j \le M$. As a result, S can be simplified as $S = N_i$. So we are left with estimating N_i.

N_i is the expected number of steps before termination if the transitions (i.e., re-optimization steps) are confined in the partition i. We can divide this process into stages, and each stage contains one more join than the previous stage. If there are m joins, then we will have m stages. In each stage, we go through every plan until

we finish, and the average number of plans we considered is S_{K_j} as computed by Equation 1, where K_j is the number of join-trees (actually subtrees) in the stage j ($1 \leq j \leq m$). So

$$S = N_i = \sum_{j=1}^{m} S_{K_j}.$$

As an upper bound, we have $S \leq S_{N/M}$, where N is the total number of different join trees considered by the optimizer, and $S_{N/M}$ is computed by Equation 1. This is obtained by applying Theorem 4 to each partition. $S_{N/M}$ is usually much smaller than S_N, according to Figure 3. Again, we emphasize that the analysis here only targets worst-case expectations, which might be too pessimistic.

C. JOINT/MARGINAL DISTRIBUTIONS

To see why the straightforward approach that first generates a relation $(A_1, ..., A_K, B_1, ..., B_K)$ based on the joint distribution and then splits it into different relation $R_1(A_1, B_1)$, ..., $R_K(A_K, B_K)$ is incorrect, let us consider the following example.

EXAMPLE 3. *Suppose that we need to generate binary values for two attributes A_1 and A_2, with the following joint distribution:*

(1) $p_{00} = \Pr(A_1 = 0, A_2 = 0) = 0.1$*;*

(2) $p_{01} = \Pr(A_1 = 0, A_2 = 1) = 0$*;*

(3) $p_{10} = \Pr(A_1 = 1, A_2 = 0) = 0$*;*

(4) $p_{11} = \Pr(A_1 = 1, A_2 = 1) = 0.9$.

Assume that we generate 10 tuples (A_1, A_2) according to the above distribution. Then we will expect to obtain a multiset of tuples: $\{1 \cdot (0, 0), 9 \cdot (1, 1)\}$. Here the notation $1 \cdot (0, 0)$ means there is one $(0, 0)$ in the multiset. If we project the tuples onto A_1 and A_2 (without removing duplicates), we have $A_1 = A_2 = \{1 \cdot 0, 9 \cdot 1\}$. Now what is the joint distribution of (A_1, A_2) once we see such a database? Note that we have no idea about the true *joint distribution that governs the generation of the data, because we are only allowed to see the* marginal *distributions of A_1 and A_2. A natural inference of the joint distribution may be to consider the cross product $A_1 \times A_2$. In this case we have*

$$\begin{aligned} A_1 \times A_2 &= \{1 \cdot 0, 9 \cdot 1\} \times \{1 \cdot 0, 9 \cdot 1\} \\ &= \{1 \cdot (0, 0), 9 \cdot (0, 1), 9 \cdot (1, 0), 81 \cdot (1, 1)\}. \end{aligned}$$

Hence, the "observed" joint distribution of A_1 and A_2 is:

(1) $p'_{00} = \Pr'(A_1 = 0, A_2 = 0) = 1/100 = 0.01$*;*

(2) $p'_{01} = \Pr'(A_1 = 0, A_2 = 1) = 9/100 = 0.09$*;*

(3) $p'_{10} = \Pr'(A_1 = 1, A_2 = 0) = 9/100 = 0.09$*;*

(4) $p'_{11} = \Pr'(A_1 = 1, A_2 = 1) = 81/100 = 0.81$.

The "observed" joint distribution is indeed "the" joint distribution when tables are joined. It might be easier to see this if we rewrite the query in Equation 2 as

$$\sigma_{A_1=c_1 \wedge \cdots \wedge A_K=c_K \wedge B_1=B_2 \wedge \cdots \wedge B_{K-1}=B_K}(R_1 \times \cdots \times R_K). \quad (4)$$

The previous example shows that there is information loss when marginalizing out attributes, which is somewhat similar to the notion of lossless/lossy joins in database normalization theory.

D. ANALYSIS OF THE OTT QUERIES

We now compute the query size when Equation 3 holds. Consider a specific query q where the constants in the selection predicates are fixed. Let us compute the cardinality of q, which is equivalent to computing the selectivity of the predicate in Equation 4. In probabilistic sense, the selectivity s can be interpreted as the chance that a (randomly picked) tuple from $R_1 \times \cdots \times R_K$ making the selection predicate true. That is,

$$s = \Pr(A_1 = c_1, \cdots, A_K = c_K, B_1 = \cdots = B_K).$$

LEMMA 3. *When Equation 3 holds, we have*

$$s = \prod_{k=1}^{K} \frac{1}{n(A_k)}.$$

The proof of Lemma 3 is included in [1]. As a result, the size of the query q is

$$|q| = s \cdot \prod_{k=1}^{K} |R_k| = \prod_{k=1}^{K} \frac{|R_k|}{n(A_k)}.$$

To summarize, we have

$$|q| = \begin{cases} \prod_{k=1}^{K} \frac{|R_k|}{n(A_k)}, & \text{if } c_1 = \cdots = c_K; \\ 0, & \text{otherwise.} \end{cases}$$

Now what will be the query size estimated by the optimizer? Again, let us compute the estimated selectivity $\hat{s}$, assuming that the optimizer knows the exact histograms of the A_k and B_k ($1 \leq k \leq K$). Note that this assumption is stronger than the case in Section 4.2.1, where the optimizer possesses exact histograms only for MCV's. We have the following result:

LEMMA 4. *Suppose that the AVI assumption is used. Assuming that $\mathrm{Dom}(B_k)$ is the same for $1 \leq k \leq K$ and $|\mathrm{Dom}(B_k)| = L$, the estimated selectivity $\hat{s}$ is then*

$$\hat{s} = \frac{1}{L^{K-1}} \prod_{k=1}^{K} \frac{1}{n(A_k)}.$$

The proof of Lemma 4 is included in [1]. Hence, the estimated size of the query q is

$$\widehat{|q|} = \hat{s} \cdot \prod_{k=1}^{K} |R_k| = \frac{1}{L^{K-1}} \prod_{k=1}^{K} \frac{|R_k|}{n(A_k)}.$$

Note that this is regardless of if Equation 3 holds or not. In our experiments (Section 5) we further used this property to generate instance OTT queries.

Comparing the expressions of $|q|$ and $\widehat{|q|}$, if we define

$$d = \mid |q| - \widehat{|q|} \mid,$$

it then follows that

$$d = \begin{cases} \left(1 - \frac{1}{L^{K-1}}\right) \cdot \prod_{k=1}^{K} \frac{|R_k|}{n(A_k)}, & \text{if } c_1 = \cdots = c_K; \\ \frac{1}{L^{K-1}} \cdot \prod_{k=1}^{K} \frac{|R_k|}{n(A_k)}, & \text{otherwise.} \end{cases}$$

Let us further get some intuition about how large d could be by considering the following specific example.

EXAMPLE 4. *For simplicity, assume that $M = |R_k|/n(A_k)$ is the same for $1 \leq k \leq K$. M is then the number of tuples per distinct value of A_k ($1 \leq k \leq K$). In this case,*

$$d = \begin{cases} \left(1 - 1/L^{K-1}\right) \cdot M^K, & \textit{if } c_1 = \cdots = c_K; \\ M^K / L^{K-1}, & \textit{otherwise.} \end{cases}$$

If $L = 100$ and $M = 100$, it then follows that

$$d = \begin{cases} 100^K - 100, & \textit{if } c_1 = \cdots = c_K; \\ 100, & \textit{otherwise.} \end{cases}$$

For $K = 4$ (i.e., just 3 joins), $d \approx 10^8$ if it happens to be that $c_1 = \cdots = c_K$. Moreover, if $c_1 = \cdots = c_K$, then $|q| > \widehat{|q|}$. Therefore, the optimizer will significantly underestimate the size of the join (by 10^8). So it is likely that the optimizer would pick an inefficient plan that it thought were efficient.