Cardinality Estimation Done Right: Index-Based Join Sampling

Viktor Leis, Bernhard Radke, Andrey Gubichev[†,*], Alfons Kemper, Thomas Neumann
Technische Universität München
{leis,radke,kemper,neumann}@in.tum.de
Google[†]
gubichev@google.com[†]

ABSTRACT

After four decades of research, today's database systems still suffer from poor query execution plans. Bad plans are usually caused by poor cardinality estimates, which have been called the "Achilles Heel" of modern query optimizers. In this work we propose *index-based join sampling*, a novel cardinality estimation technique for main-memory databases that relies on sampling and existing index structures to obtain accurate estimates. Results on a real-world data set show that this approach significantly improves estimation as well as overall plan quality. The additional sampling effort is quite low and can be configured to match the desired application profile. The technique can be easily integrated into most systems.

1. INTRODUCTION

Cost-based query optimization is fundamentally based on cardinality estimation. Virtually all industrial-strength systems estimate cardinalities by combining some fixed-size, per-attribute summary statistics (histograms) with strong assumptions (uniformity, independence, inclusion, ad hoc constants). In other words, most databases try to approximate an arbitrarily large database in a constant amount of space. It is therefore not surprising that estimators have a very hard time detecting complex patterns like join-crossing correlations. For real-world data sets, cardinality estimation errors are large and occur frequently [20, 15]. These errors lead to slow queries and unpredictable performance.

One promising alternative to histogram-based estimation is sampling, as it can detect arbitrary correlations and therefore produces much more accurate estimates. However, despite being studied for decades [19, 28], few systems actually use sampling in production. One reason is that in the past, when main memory capacity was small, sampling was too expensive to be practical due to the high cost of random disk I/O operations. For example, using a conventional disk, sampling only 10 random tuples may take 30 ms. Today, many databases fully reside in RAM; assuming a tuple access time of 100 ns, in the same 30 ms it is possible to sample 300,000 random rows, which makes sampling much more practical.

[*]This work was primarily done while affiliated with TU München.

8th Biennial Conference on Innovative Data Systems Research (CIDR '17) January 8-11, 2017 , Chaminade, California, USA.

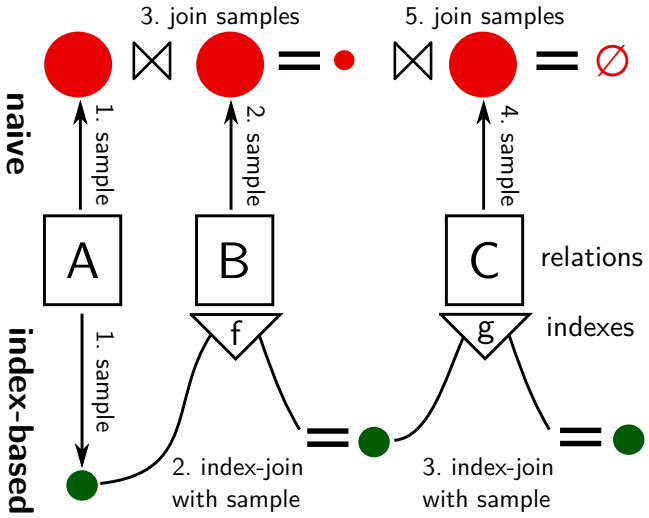

Figure 1: Naive (top) vs. index-based (bottom) join sampling

A second reason why sampling is not yet widely used is that even a relatively long sampling time may not be sufficient to estimate joins. Consider, for example, the join $A \bowtie_{A.p=B.f} B$, where p is a primary key and f is an (undeclared) uniformly distributed foreign key[1]. A standard technique for estimating joins using sampling is illustrated in the top half of Figure 1. To estimate $A \bowtie B$, the independent random samples of A and B are joined. It is easy to derive (cf. Appendix A) that for this join to produce a certain expected number of result tuples n, the number of sampled tuples must be $\sqrt{n|A|}$. For example, if $|A| = 10{,}000{,}000$ and we would like to retain $n = 1{,}000$ samples (after the join), we have to sample about 100,000 rows from both A and B. Any additional join (e.g., with relation C in the figure) will quickly reduce the sample size further. Given the large sample sizes required, systematically estimating many intermediate results is simply not feasible.

In this work we propose a novel cardinality estimation technique that produces accurate results but is much cheaper than joining random samples. The basic building block is an efficient sampling operator that utilizes existing index structures: As shown in the lower half of Figure 1, to get an estimate for $A \bowtie_{A.p=B.f} B$, we obtain a random sample of A and then look up the samples' join partners

[1]We use this example to keep the math simple. If the key/foreign key relationship is declared, a database system can, of course, estimate this particular join easily. Estimation is much harder in the presence of selections, undeclared keys, or non primary key joins.

in the index for $B.f$ (we could also start with B using an index on $A.p$). The resulting sample for $A \bowtie B$ can be used as a starting point for obtaining a sample for $A \bowtie B \bowtie C$ by using an index on the join attribute $C.g$ and so on. Index-based join sampling is much cheaper than joining independent samples as the sample size stays roughly constant after each additional join.

Cheaply obtaining samples for some intermediate results is, however, only part of the solution. Large queries can have hundreds or even thousands of intermediate results, and—even with cheap, index-based sampling—it is not feasible to sample all of them. The question therefore is: Which intermediate results should be sampled? We argue that one should proceed bottom-up, i.e., to compute accurate estimates for 2-way joins, then for 3-way joins, and so on. We then inject these accurate estimates into a traditional query optimizer that exhaustively enumerates all join orders—but with much better information about the true costs. Other sampling-based approaches either greedily determine the join order during sampling [12] or only use sampling to locally improve existing query plans [30]. Our approach, in contrast, utilizes the strength of exhaustive enumeration but puts it on much firmer ground with accurate, sampling-based estimates.

While sampling in main memory is quite cheap, the additional sampling phase nevertheless increases query optimization time. Queries with many joins have many intermediate results, each of which should ideally be estimated by a separate sampling step. We therefore set a time budget for the sampling phase and fall back to traditional estimation after that. As a result, our approach has very low overhead, which can be configured depending on the application profile and query type.

The rest of this paper is organized as follows. After contrasting our approach against other sampling-based proposals in Section 2, Section 3 describes index-based join sampling and its integration into database systems. In Section 4 we extensively evaluate our approach using a real-world data set and a large set of multi-join queries. The results show that index-based join sampling produces much better plans than traditional estimators and other sampling methods. We show that the sampling overhead of our technique is quite low, which makes it highly practical for complex analytical queries that often suffer from bad query plans. In Section 5 we describe query optimization techniques that have similar goals but are not based on sampling. Finally, Section 6 summarizes the paper and discusses future research directions.

2. BACKGROUND: SAMPLING-BASED QUERY OPTIMIZATION

Theoretical [11] as well well as empirical [15] work has shown that estimation errors increase exponentially with the number of joins. A large body of work therefore aims at improving estimation, plan quality, and robustness. In the following we focus on some recent proposals that, like our approach, rely on sampling. A more complete discussion of related work can be found in Section 5.

Sampling-based query re-optimization [30] first obtains a query plan using conventional estimates. However, instead of executing this plan, it draws independent samples from each table of the query and then joins these samples. Since the samples are independent, the sample size must be quite large to reduce the risk of empty join results (the paper uses 5% of the relation size). The cardinality estimates obtained by sampling are injected into the query optimizer, which is run again to compute a new query plan. The process repeats until the plan does not change any more. Sampling-based query re-optimization sometimes avoids bad query plans, but suffers from high sampling overhead (due to large, $O(n)$ sample sizes) and often misses good plans due to the greedy exploration strategy.

CS2 [31], in contrast, uses materialized samples instead of sampling for each query. It pre-computes one small sample for each relation and joins these samples when estimating cardinalities. The main idea of CS2 is to overcome the problem of empty results that often occur when small samples are joined by using "correlated" samples. In a star schema, for example, the fact table would serve as a starting point ("source relation") for which a normal sample is computed. A tuple in a dimension table only becomes part of the dimension table's sample if it has a join partner in the fact table's sample. While CS2 works very well for star- or snowflake-like schemas, it is not clear how to apply it *automatically* to more complex schemas.

Despite being proposed for a non-relational language, the Runtime Optimizer for XQueries (ROX) [12] has most similarities with our approach. ROX also relies on existing index structures in order to cheaply compute accurate samples for multi-join queries. ROX, however, uses front-biased cutoff sampling instead of our unbiased sampling method. Furthermore, in contrast to the exhaustive, bottom-up approach which we propose, ROX greedily picks edges to sample and this sampling process directly determines the join order. This has the major disadvantage that some good plans are simply never enumerated and thus cannot be chosen.

The sampling-based approaches discussed above have weaknesses that preclude their use in industrial-strength systems. Any technique that requires human intervention is very unlikely to be adopted widely. The same is true for techniques that have high overhead, because a significant number of plans are actually pretty close to the optimum[2]—despite large estimation errors. Greedy algorithms will loose significant performance from not fully enumerating the search space—often slowing down queries that had good plans in the first place. Our index-based sampling approach, in contrast, has low (and configurable) overhead, is fully automatic, and integrates with exhaustive join enumeration without resorting to greedy algorithms.

3. INDEX-BASED JOIN SAMPLING

Our approach for improving cardinality estimates consists of two components. The first is an index-based sampling operator that cheaply computes a sample for a join result. The second component is a join enumeration strategy, which systematically explores the important intermediate results of a query using the index-based sampling operator and ensures that the overall sampling time is limited. We first describe these two components before discussing how our approach fits into the architecture of a typical database system.

3.1 The Index-Based Sampling Operator

To estimate the cardinalities of a query, we first compute random samples of constant, configurable size for each base relation in the query. To estimate $\sigma_{A.x=1}(A)$, for example, we pick random tuples (without replacement) from A and apply the selection $A.x = 1$ to obtain a sample and accurate estimate. If an index on $A.x$ exists, it is also possible to directly sample from the index (as described later for joins).

The basic idea of join sampling is quite simple: There are two ways to compute a sample for the expression $\sigma_{A.x=1}(A) \bowtie_{A.p=B.f} B$. Using a sample of B, one can probe in the index $A.p$, and finally apply the selection $A.x = 1$. Or, if an index on $B.f$ exists, one can start

[2]The claim that many plans are good may seem contradictory with our previous statement that bad plans are common. However, both statements are true (cf., Figure 6). Indeed, the fairly good quality of the "average" plan is one of the main hurdles for the adoption of any advanced (and therefore more expensive) estimation technique.

Algorithm 1 Index-based join sampling

sampleIndex(S, I, n)
Input: sample S of intermediate result T,
suitable index I of relation A,
maximum sample size n
Output: sample for $T \bowtie A$

cpt = empty sequence of $(tuple, count)$ pairs
for each $t \in S$
  append $(t, I.lookup(t).count)$ to cpt
$sum = \sum_i cpt[i].count$ // total join size
S_{out} = empty sequence of tuples
sid = sample non-negative integers $< sum$, $|sid| = \min\{sum, n\}$
for each $id \in sid$
  $chosen = \max\{i | (\sum_{j=1}^{i} cpt[j].count) \leq id\}$
  $t_S = cpt[chosen].tuple$
  $offset = id - \sum_{i=1}^{chosen-1} cpt[i].count$
  $t_A = (I.lookup(t_S))[offset]$
  append $(t_S \circ t_A)$ to S_{out}
return S_{out}

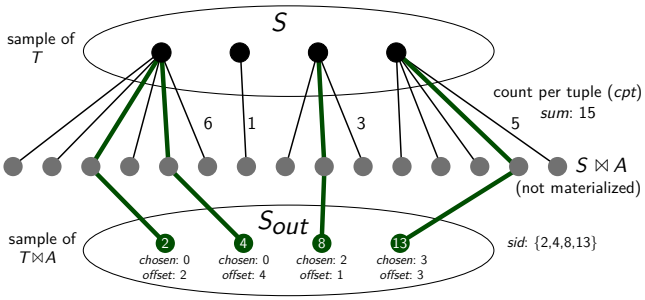

Figure 2: Illustration of Algorithm 1. Using a sample S, which consists of 4 tuples, and a suitable index, the algorithm first counts the number of join partners for each tuple in S (6, 1, 3, and 5). To compute the final sample, it then draws 4 random tuples (without replacement) from these 15 candidates

with the previously computed sample for $\sigma_{A.x=1}(A)$ and probe in the index $B.f$. While both approaches typically compute accurate results, they differ in how many samples they may produce. If $B.f$ is a foreign key, the number of samples after the join may be much larger than the original sample. Large samples are undesirable, as they increase sampling time.

Our sampling strategy, shown in Algorithm 1, therefore directly samples such that the result sample size does not exceed the given maximum sample size (denoted as n). The algorithm consists of two phases and is illustrated in Figure 2. First, for each tuple of the input sample S, the number of matching tuples in the index is counted without materializing this join of S and A. The overall count, of course, can be much larger than the desired sample size. Therefore, in the second step, the algorithm randomly draws and materializes n tuples (without replacement) from this (potentially large) set of matching tuples.

The algorithm assumes that indexes allow one to cheaply ($O(1)$ or $O(\log n)$) count the number of matches of a particular key and to select the k-th value of a particular key. This is, for instance, trivially the case for hash indexes that associate each key with a vector of values. For tree indexes (e.g., B-Trees), implementing these operations efficiently is slightly more complicated, but can be achieved by augmenting each node with an order statistic (cf. [5, Chapter 14]).

3.2 Greedy vs. Bottom-Up Enumeration

The index-based sampling operator efficiently computes a fixed-size sample for one intermediate join result. This allows one to sample multiple intermediate results for a given query. For queries with less than 8 joins it is even realistic to sample *all* intermediate results (cf. Section 4.3). (We do not sample cross products, as most join enumeration algorithms ignore them anyway.)

Unfortunately, the number of intermediate results grows exponentially with the number of joins in a query. In the JOB benchmark [15], for example, queries with 7 joins have 84–107 intermediate join results, and queries with 13 joins already have 1,517–2,032 intermediate results (again ignoring cross products). For large queries, we therefore set a limit on the time spent in the sampling phase and fall back to traditional estimation after that. For queries with many intermediate results, sampling is not exhaustive and the sampling *order* becomes important.

Intuitively, one might be inclined to follow a ROX-style greedy approach, i.e., to start with a small result and extend it one-by-one until all relations in the query graph are covered. The advantage is that one quickly obtains accurate estimates for large intermediate results. The disadvantage is that many small intermediate results are not sampled and thus have to be estimated using traditional estimation. It is well known that—due to the independence assumption—traditional estimators tend to *underestimate* result sizes. Therefore, when this mix of (accurate) sampling-based estimates and traditional (under-)estimates are injected into a query optimizer, it will often pick a plan based on the traditional estimates (as they appear to be very cheap). This phenomenon has been called "fleeing from knowledge to ignorance" [21] and—paradoxically—causes additional, accurate information to *decrease* plan quality.

To avoid the "fleeing from knowledge" issue we compute intermediate results in a bottom-up fashion, i.e., we first sample all 2-way joins, then all 3-way joins, and so on until the we run out of budget. This will result in accurate estimates for smaller intermediate results (and, under a limited budget, traditional estimation for larger results). Our experiments indeed show that it is better to spend the sampling time on the smaller intermediate results. This is because it may often be feasible to exhaustively sample all 2-way, 3-way, and 4-way joins, but not larger results. A cost-based query optimizer will thus have precise knowledge of the costs for the early (and often crucial) joins.

3.3 Enumeration Algorithm

Algorithm 2 shows the pseudo code for our bottom-up enumeration approach, which is superficially similar to System R's dynamic programming algorithm that enumerates plans by size and does not consider bushy trees. However, in contrast to traditional dynamic programming, our algorithm does not determine the join order, but only computes samples and (not shown in the pseudo code) cardinality estimates. In other words, it samples each intermediate result instead of finding the cheapest alternative for that result. Also note that, because index-based sampling requires at least one base relation as input, the enumeration ignores bushy trees (though this only affects estimation and the eventual plan *can* be bushy).

The algorithm also keeps track of the number of index lookups during sampling and stops once the given budget (e.g., 100,000 index lookups) is exhausted. As previously mentioned, we assume that sampling a tuple from an index is cheap. Thus, the number of index lookups correlates strongly with the sampling time. However,

Algorithm 2 Sampling-based estimation with a budget

```
estimateQuery(G,b,n)
Input:  query graph G (relations: vertices, predicates: edges),
        sampling budget b,
        maximum sample size n
Output: table of samples

samples = structure that maps expressions to samples
for each R ∈ G.getRelations()
  samples[{R}] = sampleRelation(R,n)
budget = b
for each size from 1 to G.getRelations().size() − 1
  for each (exp_in, s_in) ∈ samples.getEntriesOfSize(size)
    for each R ∈ G.getNeighbours(exp_in)
      exp_out = exp_in ∪ {R}
      if (samples[exp_out].size() < n/10)∧
          (R.hasIndex(exp_in) ∨ |R| ≤ n)
        S_out = sampleIndex(s_in, R.getIndex(exp_in), n)
        samples[exp_out] = S_out
        budget = budget − sampleCost(s_in, S_out, R)
        if budget < 0
          return samples
return samples
```

it would also be possible use a time limit directly (e.g., 100 ms) as a budget.

For long-running reporting queries the budget can be set to a high value to ensure that a good plan is found. A fairly high budget is also fine for interactive, ad hoc queries, because from the perspective of a user it does not really matter if the query takes 1 ms or 100 ms (both are perceived as instantaneous). What matters is that bad query plans (e.g., ones that take 10 minutes) are avoided. For simple queries that require low latency, on the other hand, the sampling budget can be set to a very low value (or even 0). Another approach, which we plan to investigate in the future, is to adaptively set the budget depending on the estimated query cost.

Besides managing the budget, the algorithm also contains two additional optimizations. Previously, we have discussed how the index-based sampling operator avoids the problem of too large sample sizes. However, there is also the inverse problem, i.e., too small samples, which can occur for selective queries. To reduce the impact of this problem, we therefore sometimes revisit (and therefore sample) an expression multiple times if the sample size is below a threshold (e.g., $n/10$). The second optimization in our algorithm is to directly join small relations with a sample if no index exists. If a relation has less than n tuples, we simply (hash) join it in its entirety with the sample, since this is cheap anyway and increases the number of expressions for which one can obtain accurate estimates.

3.4 Database System Integration

Let us close this section by discussing the interplay of index-based sampling with the rest of the database system. Figure 3 shows that index-based sampling is performed before the traditional query optimizer as an additional phase that computes accurate cardinality estimates. The resulting estimates, which may be incomplete, are injected into the existing cardinality estimation component of the query optimizer. No changes to the cost model or plan space enumeration algorithm are necessary. As a result, index-based join sampling can be utilized by bottom-up (e.g., [23, 25]) as well as top-down (e.g., [8]) join enumeration algorithms.

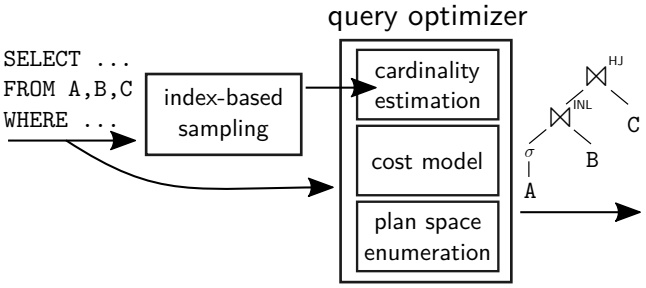

Figure 3: Integration of index-based join sampling into a traditional query optimizer

Except for the addition of the index-based sample operator, our approach does not require any changes to the query engine. In contrast to adaptive query processing techniques (cf. Section 5), there is also no feedback between the query execution and query optimization. In other words, the plan produced by the optimizer is executed as is (without any additional overhead at execution time). As a consequence, index-based sampling integrates well with modern compilation-based query engines like HyPer, which compile the query plan into machine code. Also note that such query engines already have compilation overhead on the order of 100 ms [27]. In such a setting a typical sampling overhead of, e.g., 10 ms will barely be noticed.

Our approach performs additional index lookups during query optimization time. In a concurrent setting these additional lookups must use appropriate synchronization techniques (e.g., latches). However, the total number of lookups is fairly low, and with modern, optimistic synchronization protocols like Optimistic Lock Coupling [18] the synchronization overhead is very low and readers do not negatively affect writers.

Any query optimizer change that increases the performance for the vast majority of queries, will also decrease performance for some queries, which is very undesirable in production systems. Existing database systems are therefore very conservative with query optimizer changes. Thus, one could use our approach as an optional tuning feature for queries that are slower than expected. In other words, if a user is not satisfied with the performance of a particular query, to get better performance she may turn on index-based sampling only for that query.

4. EVALUATION

In this section we investigate index-based join sampling experimentally. We show that (1) the estimates are much improved, (2) the sampling overhead incurred is low, (3) the plan quality is much better than that of alternative approaches, and that (4) our technique works well over a wide spectrum of access path configurations.

4.1 Experimental Setup

In prior work [15] we have shown that cardinality estimation for synthetic benchmarks like TPC-H is unrealistically easy. Our experiments therefore use the Join Order Benchmark (JOB) [15], which is based on the Internet Movie Database. This real-world data set is around 3.6 GB and has a complex schema. The JOB workload consists of 113 queries with 3 to 16 joins. Because of the bad quality of commercial cardinality estimates, query optimizers often do not find good plans for JOB, in particular, when many indexes are available [15].

The experiments were performed in a single-threaded in-

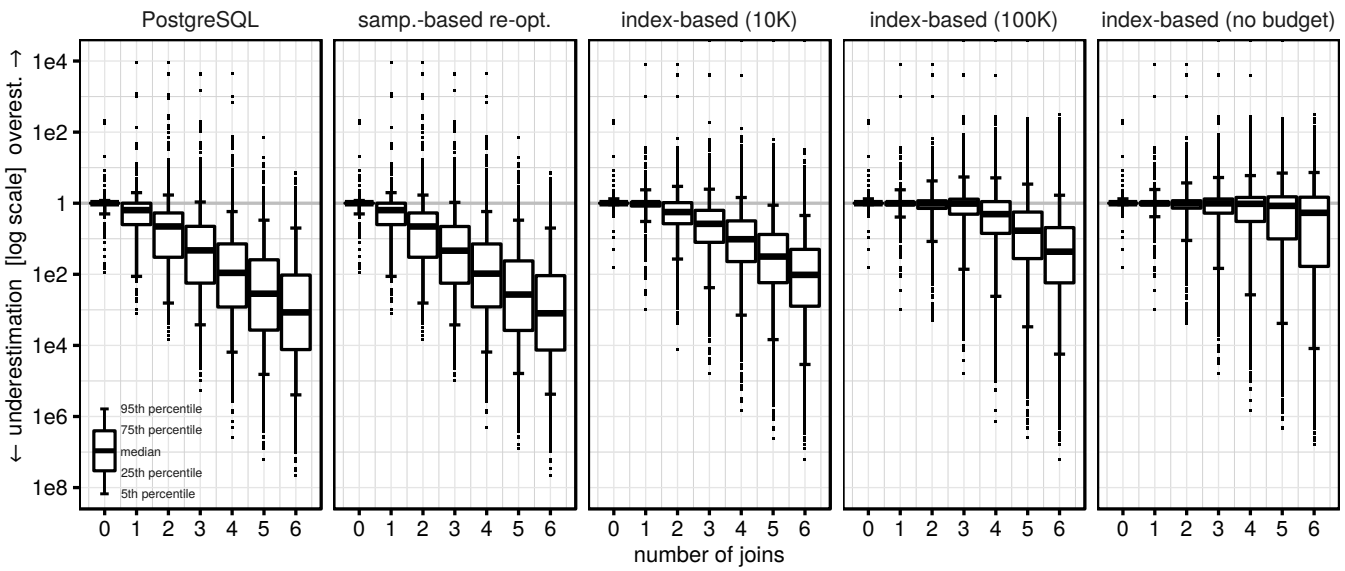

Figure 4: Quality of cardinality estimates for multi-join queries in comparison with the true cardinalities. Each boxplot summarizes the error distribution of all subexpressions with a particular size (over all queries in the workload)

memory prototype similar to MonetDB (column-wise storage, full materialization after each operator) developed for query optimization experimentation. For simplicity, indexes are implemented as hash tables; efficient order-preserving data structures like ART [16] would also be suitable. For most experiments, we create indexes on all primary key and foreign key columns, which is the worst case in terms of relative plan quality (more indexes make it harder to find the optimal plan) and sampling overhead (more indexes allow for more sampling steps). Despite not being heavily optimized for query performance, our prototype is about 5 times faster than PostgreSQL.

Like most systems, our prototype uses cost-based dynamic programming to determine the join order and the join algorithm (hash or index-nested-loop join). As the cost function we use C_{mm} [15], which, in effect, counts the number of tuples that pass through each operator. By default, cardinalities are estimated using index-based sampling and a sample size of 1,000. As a sampling-based competitor, we implemented *sampling-based query re-optimization* [30].

In order to compare our approach with traditional estimation, we support injecting estimates from other systems. We inject PostgreSQL's cardinality estimates, which were obtained using the EXPLAIN command. Note that we only use cardinality estimates (not costs), which avoids any issues stemming from incompatible cost models and allows one to compare different estimation techniques in a fair way. The quality of cardinality estimates of most commercial systems is similar to PostgreSQL [15].

We also experimented with ROX-style [12] join ordering. The algorithm starts with the best join edge (i.e., the one that yields the smallest result), and then samples join edges in breadth-first search (BFS) manner. Once the BFS is finished, the path (i.e., sequence of joins) that reduces the size of the input relation most, is executed. Starting at this partial result, sampling and execution alternate until all the query graph edges (i.e., relations) are covered. Despite improved cardinality estimation due to sampling, in our experiments ROX-style join ordering was not competitive (cf., Table 1) due to its greedy nature.

4.2 Does Sampling Improve Estimation?

The most important quality of any cardinality estimator is, of course, the quality of its estimates. Figure 4 compares the quality of index-based sampling with PostgreSQL's estimates for all intermediate results in our query set. The figure summarizes the distribution of the cardinality estimates in comparison with the true cardinalities. As observed in prior work [15], we see two major trends: First, with increasing join sizes, the errors increase exponentially (note the logarithmic scale), as evidenced by the increasing heights of the box plots. Second, underestimation is much more common than overestimation and is getting more pronounced with each additional join. The estimates of sampling-based re-optimization are very similar to PostgreSQL, as it samples only a small number of intermediate results.

Index-based join sampling, in contrast, does much better than PostgreSQL, in particular with a higher budget. Underestimation occurs later because smaller intermediate results are estimated accurately using sampling. Large underestimation errors only occur once we run out of budget, which is not surprising because we fall back to PostgreSQL's join estimation formula in that case. Also note that even with the low budget of 10,000, which only allows one to sample only a few joins, the estimates of larger joins are better than PostgreSQL's because larger results are computed using smaller, more accurate ones.

4.3 How Expensive is Sampling?

Let us next look at the sampling overhead. After all, sampling is only practical if it is cheap. For each of the 113 queries in the JOB workload, Figure 5 shows the sampling time for sampling-based re-optimization (blue circles) and for our index-based sampling technique (green symbols) under three budget settings: a very low budget of 10,000 index lookups, a medium budget of 100,000 index lookups, and an unrestricted budget. (Note that PostgreSQL is not shown in the figure because its fairly simple cardinality estimation process is very cheap.)

Even without a budget, for queries with up to 8 joins, the sampling overhead is quite low (less than 20 ms). This means that for

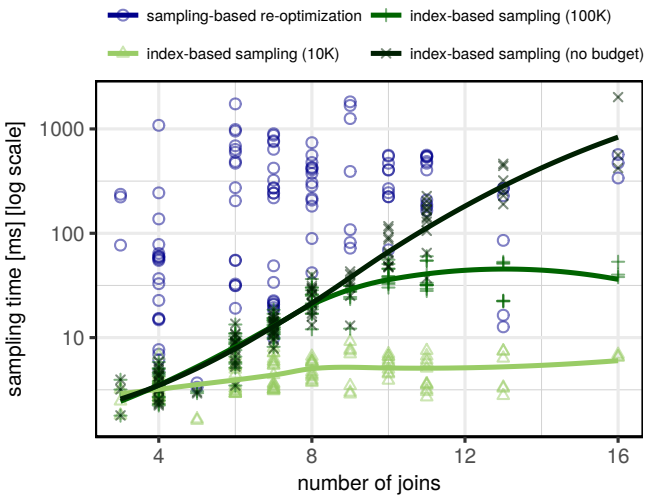

Figure 5: Overhead of sampling

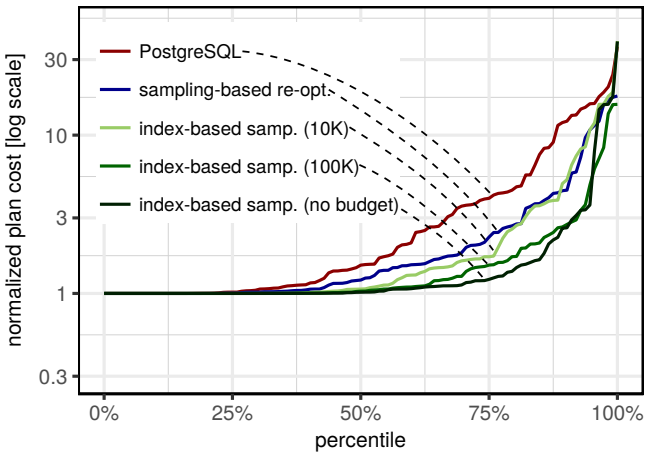

Figure 6: Plan quality in comparison with the optimal plan (determined using true cardinalities)

medium-sized queries, it is possible to fully sample all intermediate results within a time span that lies below the human perception threshold (around 100 ms). For larger queries, a sampling budget effectively limits the sampling overhead (to less than 10 ms and less than 60 ms respectively). These results show that index-based sampling in RAM is cheap enough for interactive applications.

The sampling overhead of sampling-based re-optimization is generally much larger (note the logarithmic scale). The reason is that it uses large sample sizes (5% of each relation), which results in up to 1.8 s sampling overhead. In addition, sampling-based re-optimization requires running the join ordering algorithm for each re-optimization step (which is not included in the measurements). Another important difference between the two sampling techniques is that index-based sampling systematically explores the space of cardinality estimates. As a result, its sampling overhead, which can effectively be bounded using a budget, mainly depends on the number of joins but *not* on the database size. The sampling overhead of sampling-based re-optimization, in contrast, grows linearly with the database size (not shown in the graph).

4.4 Does Sampling Improve Plan Quality?

The actual purpose of query optimization is to find good query

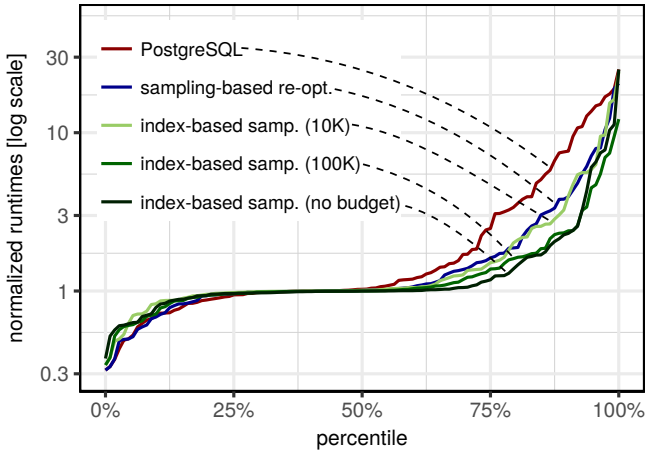

Figure 7: Runtimes in comparison with the optimal plan (determined using true cardinalities)

Table 1: Geometric means of query runtimes and sampling times (ms)

	runtime	sampling overh.
PostgreSQL	57.1	-
ROX	120.0	1.6
sampling-based re-optimization	48.0	144.0
index-based sampling (10k)	48.4	4.3
index-based sampling (100k)	43.3	13.3
index-based sampling (no budget)	43.4	19.3
true cardinalities	35.8	-

plans. Figure 6 shows the plan quality for different cardinality estimation techniques. The cost of each query is normalized by the cost of the optimal plan that would have been chosen if the true cardinalities were known. Using PostgreSQL's estimates, only around one quarter of the plans are close to the optimum. 42% of the plans are off by a factor of 2 or more and 12% are off by a factor of 10 or more. Sampling-based re-optimization slightly improves the plan quality in comparison with PostgreSQL but the gap to the optimum is still large.

Index-based sampling achieves better results. Even a very small budget of 10,000 index lookups achieves results similar to sampling-based re-optimization—using only a tiny fraction of the sampling overhead. A more realistic budget of 100,000 index lookups further improves performance for many queries: Only 17% of the queries are off by a factor of 2 or more and only 3% of the queries are off by a factor of 10 or more. Generally, the higher the sampling budget, the better the plan quality, but even a low budget results in plans better than the other techniques.

We observe similar results, shown in Figure 7, when we look at actual runtimes rather than the costs. One difference is that some plans are actually faster (up to a factor of 3) with inaccurate estimates than with the true cardinalities. This effect is caused by cost model errors rather than inaccurate cardinalities and explains the hesitation of many commercial database systems to change their query optimizers. Any optimizer change that will improve performance for the vast majority of queries, will inevitably slow down some queries. Nevertheless, the upside of the improved estimates is clearly visible.

Table 1 reports the geometric mean of all runtimes in our workload and compares them with the sampling times. Let us note again that the sampling overhead of our technique, which is already usu-

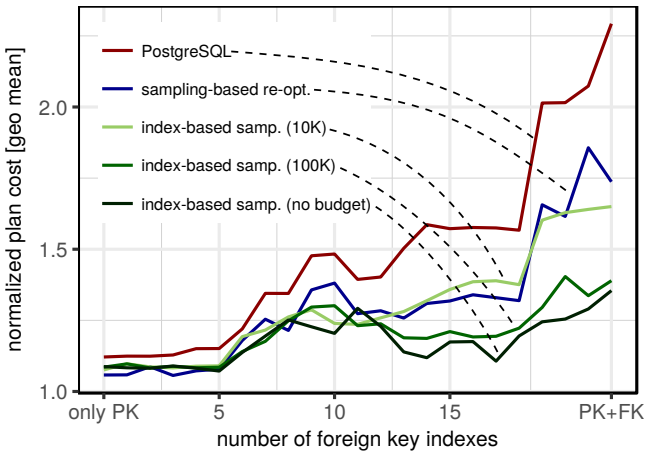

Figure 8: Query runtime under different index configurations

ally much lower than the query runtime on our moderately sized data set, is independent of the data size. On larger data sets it would indeed be negligible.

Table 1 also shows that the ROX algorithm performs worse than PostgreSQL. This may be surprising, as ROX was designed to pick up on join-crossing correlated predicates prevalent in our benchmark. To be conservative on the sample exploration cost, ROX is a greedy algorithm, which explores the space only from the cheapest first execution step (typically a selection). Our theory is that in the extensive DBLP 3-way join evaluation in [12] where anti-correlated predicates play a role, the best plan is typically to join the relation with the smallest selection with its anti-correlated counterpart. In the DBLP experiments, ROX not only succeeds in identifying these but also works on a physical infrastructure with partitioned structural indexes, where these anti-correlated joins can be executed very efficiently (as discussed in Section 4.4 of [15]). Thus, the greedy nature of ROX is not a hindrance to its performance there, whereas the situation in the JOB benchmark appears to be quite different.

4.5 What If There are Few Indexes?

So far, all results were obtained with indexes on all primary and foreign key attributes. In more realistic settings, only some of the foreign key attributes will have indexes. Given our strong reliance on index structures, one obvious question is how well our technique works if fewer applicable indexes exist. We therefore created other index configurations by enabling foreign key indexes one-by-one (in a random order). Figure 8 shows the normalized geometric mean (relative to the optimal plan with true cardinalities) for each index configuration. Looking at the curve of PostgreSQL's estimates, we see that—despite the large estimation errors—the average plan quality is quite good when there are few indexes. As the number of indexes increases, however, the plans chosen by the optimizer are much further away from the optimal plan (as previously observed [15]). Index-based sampling improves plan quality across all configurations, but especially in those cases where plan quality is worst—namely, when multiple indexes are available.

5. RELATED WORK

While sampling is a highly promising technique, many other approaches for improving plan quality and robustness have been proposed. In the following, we describe some of the major paradigms, most of which diverge from the traditional first-optimization-then-execution model.

One intuitive idea is to detect cardinality estimation errors at runtime, as in the LEO project [29]. Successive executions of the same (or a very similar) query can utilize the true cardinalities determined in an earlier run instead of inaccurate estimates. The issue of non-consistent estimates, which occur when estimates from different sources are combined (true cardinalities vs. histogram-based estimates), is addressed using a maximum entropy technique [21].

In the adaptive query processing [6, 2] paradigm, the traditionally separate optimization and execution phases are merged or interleaved. The Eddies [1] algorithm, for example, continuously reorders operators by dynamically routing incoming tuples to operators until a tuple has visited all operators. The Plan Bouquet [7] and SpillBound [13] algorithms are based on the observation that if the (uncertain) selectivities of a query are very high, the plan is likely relatively cheap (because index scans are highly effective). The approach therefore starts by executing a plan that would be optimal if the selectivities are high, and—if it turns out to take longer than expected—switches to plans, which assume lower selectivities.

Eddies and Plan Bouquet are a very radical departure from the traditional query optimization model and—by moving many of the optimizer decisions to runtime—can cause significant runtime overheads. This is a problem, because most query plans picked by traditional optimizers are actually quite good—despite the frequently large cardinality misestimates. Therefore, a number of techniques have been proposed [26, 22, 3] that strive to detect optimizer mistakes using limited runtime adaptivity. Thus, these techniques have less overhead and are easier to integrate into existing systems.

Another pragmatic approach is to design the query operators in a such a way that they do not rely on cardinality estimates. Many operators adhering to this philosophy have been proposed. Smooth Scan [4], for example, is a general, intelligent access path that avoids the need to pick whether a full table scan or an index scan should be used. In a similar vein, there has been work on the join [10, 14], the window [17], and aggregation [24, 14] operators. While using intelligent operators is certainly beneficial, some important decisions (e.g., the join order) have be made at optimization time.

6. CONCLUSIONS AND FUTURE WORK

We have shown that index-based sampling in main-memory databases is cheap. This allows one to systematically sample many intermediate query results in a bottom-up fashion and inject the resulting estimates into a traditional query optimizer. Our technique finds better plans in comparison with traditional, histogram and assumption-based estimators as well as alternative sampling proposals. Given that the query plans of state-of-the-art optimizers are very fragile, we believe that index-based sampling is a highly promising and practical technique.

In the future, we plan to integrate index-based join sampling into HyPer, which already estimates the selectivity of base relation selection predicates using sampling. We will also investigate possible adaptations of our technique for databases that are larger than RAM. The fact that even a low sampling budget improves plan quality indicates that our approach might also be beneficial for databases that do not fully reside in RAM—but e.g., on modern SSDs, which have reasonable random access times.

While our fixed-size sampling strategy is simple, has low overhead, and is quite effective, it is certainly worthwhile to investigate whether more sophisticated sampling strategies (e.g., [19, 9]) would be beneficial for our approach.

7. REFERENCES

[1] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD*, 2000.

[2] S. Babu and P. Bizarro. Adaptive query processing in the looking glass. In *CIDR*, 2005.

[3] S. Babu, P. Bizarro, and D. J. DeWitt. Proactive re-optimization. In *SIGMOD*, 2005.

[4] R. Borovica-Gajic, S. Idreos, A. Ailamaki, M. Zukowski, and C. Fraser. Smooth scan: Statistics-oblivious access paths. In *ICDE*, 2015.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3. ed.)*. 2009.

[6] A. Deshpande, Z. G. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1), 2007.

[7] A. Dutt and J. R. Haritsa. Plan bouquets: query processing without selectivity estimation. In *SIGMOD*, 2014.

[8] P. Fender and G. Moerkotte. Counter strike: Generic top-down join enumeration for hypergraphs. *PVLDB*, 6(14), 2013.

[9] S. Ganguly, P. B. Gibbons, Y. Matias, and A. Silberschatz. Bifocal sampling for skew-resistant join size estimation. In *SIGMOD*, 1996.

[10] G. Graefe. A generalized join algorithm. In *BTW*, 2011.

[11] Y. E. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *SIGMOD*, 1991.

[12] R. A. Kader, P. A. Boncz, S. Manegold, and M. van Keulen. ROX: run-time optimization of XQueries. In *SIGMOD*, 2009.

[13] S. Karthik, J. R. Haritsa, S. Kenkre, and V. Pandit. Platform-independent robust query processing. In *ICDE*, 2016.

[14] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*, 2014.

[15] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *PVLDB*, 9(3), 2015.

[16] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE*, 2013.

[17] V. Leis, K. Kundhikanjana, A. Kemper, and T. Neumann. Efficient processing of window functions in analytical SQL queries. *PVLDB*, 8(10), 2015.

[18] V. Leis, F. Scheibner, A. Kemper, and T. Neumann. The ART of practical synchronization. In *DaMoN*, 2016.

[19] R. J. Lipton, J. F. Naughton, and D. A. Schneider. Practical selectivity estimation through adaptive sampling. In *SIGMOD*, 1990.

[20] G. Lohman. Is query optimization a "solved" problem? http://wp.sigmod.org/?p=1075, 2014.

[21] V. Markl, N. Megiddo, M. Kutsch, T. M. Tran, P. J. Haas, and U. Srivastava. Consistently estimating the selectivity of conjuncts of predicates. In *VLDB*, 2005.

[22] V. Markl, V. Raman, D. E. Simmen, G. M. Lohman, and H. Pirahesh. Robust query processing through progressive optimization. In *SIGMOD*, 2004.

[23] G. Moerkotte and T. Neumann. Dynamic programming strikes back. In *SIGMOD*, 2008.

[24] I. Müller, P. Sanders, A. Lacurie, W. Lehner, and F. Färber. Cache-efficient aggregation: Hashing is sorting. In *SIGMOD*, 2015.

[25] T. Neumann. Query simplification: graceful degradation for join-order optimization. In *SIGMOD*, 2009.

[26] T. Neumann and C. A. Galindo-Legaria. Taking the edge off cardinality estimation errors using incremental execution. In *BTW*, 2013.

[27] T. Neumann and V. Leis. Compiling database queries into machine code. *IEEE Data Eng. Bull.*, 37(1), 2014.

[28] F. Olken and D. Rotem. Random sampling from database files: A survey. In *SSDBM*, 1990.

[29] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO - DB2's learning optimizer. In *VLDB*, 2001.

[30] W. Wu, J. F. Naughton, and H. Singh. Sampling-based query re-optimization. In *SIGMOD*, 2016.

[31] F. Yu, W. Hou, C. Luo, D. Che, and M. Zhu. CS2: a new database synopsis for query estimation. In *SIGMOD*, 2013.

Appendix A: Sample Size When Joining Independent Samples

In the following, we derive the expected sample size when joining two independent relation samples for the common key/foreign key scenario. Consider the expression $A \bowtie_{A.p=B.f} B$, where p is a primary key and f is a uniformly distributed foreign key. If we sample m tuples from A without replacement, the probability of any tuple from B having a join partner (we assume $m \leq |A|$) is

$$\frac{m}{|A|}.$$

Because we also sample m tuples from B, the expected number of join results n is

$$n = m\frac{m}{|A|}.$$

Solving for m, we obtain the formula stated in Section 1:

$$m = \sqrt{n|A|}$$