Predicting Query Execution Time: Are Optimizer Cost Models Really Unusable?

Wentao Wu [†1] Yun Chi [‡2] Shenghuo Zhu [‡3] Junichi Tatemura [‡4]
Hakan Hacıgümüş [‡5] Jeffrey F. Naughton [†6]

[†]*Computer Sciences Department, University of Wisconsin, Madison, WI, USA*
[1,6]{wentaowu,naughton}@cs.wisc.edu

[‡]*NEC Laboratories America, Cupertino, CA, USA*
[2,3,4,5]{ychi,zsh,tatemura,hakan}@nec-labs.com

***Abstract*—Predicting query execution time is useful in many database management issues including admission control, query scheduling, progress monitoring, and system sizing. Recently the research community has been exploring the use of statistical machine learning approaches to build predictive models for this task. An implicit assumption behind this work is that the cost models used by query optimizers are insufficient for query execution time prediction. In this paper we challenge this assumption and show while the simple approach of scaling the optimizer's estimated cost indeed fails, a properly calibrated optimizer cost model is surprisingly effective. However, even a well-tuned optimizer cost model will fail in the presence of errors in cardinality estimates. Accordingly we investigate the novel idea of spending extra resources to refine estimates for the query plan after it has been chosen by the optimizer but before execution. In our experiments we find that a well calibrated query optimizer model along with cardinality estimation refinement provides a low overhead way to provide estimates that are always competitive and often much better than the best reported numbers from the machine learning approaches.**

I. Introduction

Predicting query execution time has always been desirable if somewhat elusive capability for database management systems. This capability has received a flurry of attention recently, perhaps because it has become increasingly important in the context of offering databases as a service (DaaS). A DaaS provider has to manage infrastructure costs as well as honor service level agreements (SLAs), and many system management decisions can benefit from prediction of query execution time, including:

- *Admission Control*: Knowing the execution time of an incoming query can enable cost-based decisions on admission control [28], [31].
- *Query Scheduling*: Knowing the query execution time is crucial in deadline and latency aware scheduling [9], [14].
- *Progress Monitoring*: Knowing the execution time of an incoming query can help avoid "rogue queries" that are submitted in error and take an unreasonably long time to execute [24].
- *System Sizing*: Knowing query execution time as a function of hardware resources can help in system sizing [30].

[1] The work was done while the author was at NEC Laboratories America.

Recent work on predicting query execution time [4], [12], [28], [31] has focused on various machine learning techniques, which treat the database system as a black box and try to learn a query running time prediction model. This move toward black box machine learning techniques is implicitly and sometimes explicitly motivated by a belief that query optimizers' cost estimations are not good enough for run time prediction. For example, in [12], the authors found that using linear regression to map the cost from Neoview's commercial query optimizer to the actual running was not effective (see Figure 17 of [12]). In [4], the same approach was used to map PostgreSQL's estimate to the actual execution time, and similar disappointing results were obtained (see Figure 5 of [4]).

It is clear from this previous work that post-processing the optimizer cost estimate is not effective. However, we argue in this paper that this does not imply that optimizer estimates are not useful — to the contrary, our experiments show that if the optimizer's internal cost model is tuned before making the estimate, the optimizer's estimates are competitive with and often superior to those obtained by more complex approaches. In more detail, for specificity consider the cost model used by the PostgreSQL query optimizer:

Example 1 (PostgreSQL's Cost Models): PostgreSQL's optimizer uses a vector of five parameters (referred to as *cost units*) in its cost model: $\mathbf{c} = (c_s, c_r, c_t, c_i, c_o)^\intercal$, defined as follows:

1) c_s: *seq_page_cost*, the I/O cost to sequentially access a page.
2) c_r: *random_page_cost*, the I/O cost to randomly access a page.
3) c_t: *cpu_tuple_cost*, the CPU cost to process a tuple.
4) c_i: *cpu_index_tuple_cost*, the CPU cost to process a tuple via index access.
5) c_o: *cpu_operator_cost*, the CPU cost to perform an operation such as hash or aggregation.

The cost C_O of an operator O in a query plan is then computed by a linear combination of c_s, c_r, c_t, c_i, and c_o:

$$C_O = \mathbf{n}^\intercal \mathbf{c} = n_s \cdot c_s + n_r \cdot c_r + n_t \cdot c_t + n_i \cdot c_i + n_o \cdot c_o. \quad (1)$$

The values $\mathbf{n} = (n_s, n_r, n_t, n_i, n_o)^\intercal$ here represent *the number of pages sequentially scanned*, *the number of pages randomly*

accessed, and so forth, during the execution of the operator O. The total estimated cost of a query plan is then simply the sum of the costs of the individual operators in the query plan. □

The accuracy of C_O hence depends on both the accuracy of the c's and the n's. In PostgreSQL, by default, $c_s = 1.0$, $c_r = 4.0$, $c_t = 0.01$, $c_i = 0.005$, and $c_o = 0.0025$. The cost C_O in Equation (1) is thus reported in units of sequential page I/O cost (since $c_s = 1.0$). Note that these cost units were somewhat arbitrarily set by the optimizer designers with no knowledge of the system on which the query is actually being run. Using linear regression to map an estimate so obtained will only work if the ratios among these units are correct, and not surprisingly, these default ratios were far from correct on our systems.

Of course, the accuracy of C_O also depends on the quantities n_s, n_r, n_t, n_i, and n_o. Determining accurate values for these quantities is not a matter of calibration — rather, it is a matter of good cardinality estimation in the optimizer. Hence one could say that we have reduced the problem of query time prediction to the previously unsolved problem of cardinality estimation. In a sense this is true, but further reflection reveals that we are solving a subtly but significantly different problem.

In their traditional role, cardinality estimates are required for every cardinality encountered as the optimizer searches thousands or tens of thousands of alternative plans. This of course means that the estimation process itself must be extremely efficient, or long optimization times will result. But our problem is different: we must determine cardinalities for the single plan that the optimizer has already chosen. The fact that we are working on a single plan means we can afford to spend some extra time to improve the original optimizer estimates. Specifically, in this paper we consider using sampling-based approaches to refine these estimates. We believe that although sampling-based approaches may be too expensive to be used while *searching* for good query plans, they can be practically used for *correcting* the erroneous cardinality estimates in a ready-to-be-executed query plan.

Our experiments show that if we correctly calibrate the constants in the optimizer's cost model, it yields good query execution time estimates when the cardinality estimates are good (as is the case in, for example, the uniformly distributed data set variants of the TPC-H benchmark.) Furthermore, "expensive" techniques such as sampling can be effectively used to improve the cardinality estimates for the chosen plan. Putting the two together yields cost estimates that are as good or better than those obtained by previously studied "black box" machine learning approaches.

The rest of the paper is organized as follows. We first give an overview of our cost-model based approach in Section II. We then discuss the two error-correction steps, i.e., calibrating cost units and cardinality estimates, in Sections III and IV, respectively. We further conduct extensive experimental evaluations and present our results in Section V. We summarize related work in Section VI and conclude the paper in Section VII.

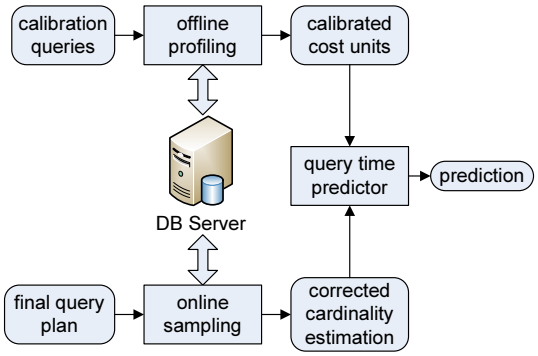


Fig. 1. The architecture of our framework.

II. Our Framework

Example 1 demonstrates that errors in $\mathbf{c}$ and $\mathbf{n}$ might prevent us from leveraging $C_O = \mathbf{n}^\intercal\mathbf{c}$ to predict query execution time. Our basic idea is simply to correct these errors in a principled fashion.

As illustrated in Figure 1, our framework consists of two error-correction stages, namely, an *offline* profiling stage to calibrate the $\mathbf{c}$, and an *online* sampling stage to refine the $\mathbf{n}$:

- *Offline profiling to calibrate* $\mathbf{c}$:
  The errors in $\mathbf{c}$ reflect an inaccurate view of the underlying hardware and database system. To correct this, instead of using the default values assigned by the query optimizer, we calibrate them by running a family of profiling queries on the system on which the queries will be run. Note that this profiling stage is offline. Moreover, it only needs to be run once as long as the underlying hardware and software configuration does not change.
  We describe the details of the profiling stage in Section III.
- *Online sampling to refine* $\mathbf{n}$:
  The errors in $\mathbf{n}$ reflect errors in cardinality estimation. Once the query plan has been chosen by the query optimizer, we re-estimate cardinalities, if necessary, using a sampling-based approach. Although using sampling for cardinality estimation is well-known, current DBMS optimizers exclude sampling from their implementations, perhaps due to the additional overhead sampling incurs. However, since we only have to estimate cardinalities for one plan, the overhead of sampling is affordable in practice. We describe the details of the sampling stage in Section IV. We note that the important idea is that there is an opportunity to spend extra time refining cardinality estimates once a query plan has been chosen; sampling is one example of how that could be done, and finding other techniques is an interesting area for future work.

The advantages of our framework for predicting query execution time include the following:

- *Lightweight*: The profiling step is fast and so can be conducted in a new hardware environment quickly. The sampling step, as will be shown, introduces small (usually $< 10\%$) and tunable overhead.

- *No training data needed*: Unlike machine-learning-based approaches, which heavily rely on training data representative of the actual workload, our framework does not rely on such a training data set and so can handle *ad hoc* queries well.
- *White-box approach*: Instead of sophisticated statistical models (e.g., SVM and KCCA), which are often difficult to understand for non-experts, our framework adopts an intuitive white-box approach that fits naturally into the existing paradigm of query optimization and evaluation in relational database systems.

III. Calibrating Cost Units

In this section we consider the task of calibrating the cost units in the optimizer cost model to match the true performance of the hardware and software on which the query will be run. It turns out that closely related problems have been studied in the context of heterogeneous DBMS [10], DB resource virtualization [26], and storage type selection [32]. Previous work, however, has focused either on cost models for particular operators (e.g., selections and 2-way joins in [10]), or on a subset of cost units dedicated to a particular subsystem of the DBMS (e.g., CPU in [26] and I/O in [32]). We build on this previous work following their technique of using a set of *calibration queries*. The basic idea is that for each cost unit to be calibrated, one designs some particular query that isolates this parameter from the others. In practice, this is not completely straightforward in that not every cost unit can be isolated in a single query.

A. Guiding Principles

Ideally, we wish to have one calibration query for each parameter. However, this is not always possible. For instance, there is no SQL query which involves the *cpu_operator_cost* but not the *cpu_tuple_cost*. A natural generalization is then to use k calibration queries for k parameters, as was done in [26]. The following example illustrates this idea.

Example 2 (Calibration Query): Suppose `R` is some relation that is buffer pool resident. We can use the following two calibration queries to obtain the parameters *cpu_tuple_cost* and *cpu_operator_cost*:

- Q1: `SELECT * FROM R`
- Q2: `SELECT COUNT(*) FROM R`

Since `R` is memory resident, there is no I/O cost for Q1 or Q2. Q1 only involves the parameter *cpu_tuple_cost*, while Q2 involves both *cpu_tuple_cost* and *cpu_operator_cost* (due to the `COUNT` aggregation). Suppose the execution time of Q1 and Q2 are t_1 and t_2, respectively. Since the overhead due to *cpu_tuple_cost* for Q1 and Q2 are the same, we can then infer *cpu_operator_cost* with respect to the execution time $t_2 - t_1$.

Specifically, let the number of tuples processed be n_t, and the number of CPU operations be n_o, as in Equation (1). In PostgreSQL's cost model, a CPU operation means things like *adding two integers*, *hashing an attribute*, and so on. n_o is thus the number of such operations performed. On the other hand, n_t is the number of input *tuples* (sometimes the output tuples or the sum of both, depending on the specific operator). Here, for Q1, the cost model will only charge one c_t per tuple, since the CPU merely reads in the tuple without any further processing. For Q2, in addition to charging one c_t per tuple for reading it, the cost model will also charge one c_o per tuple for doing the aggregation (i.e., `COUNT`), and hence the total CPU cost is estimated to be $n_t c_t + n_o c_o$. Note that for this particular query Q2, we coincidentally have $n_t = n_o = |\mathtt{R}|$. In general, n_t and n_o could be different. For example, for the *sort* operator, n_t is the number of input tuples, and n_o is the number of comparisons made. In this case, $n_o = n_t \log n_t$.

Now suppose that T_t is the time for the CPU to process one tuple, and T_o is the time for one CPU operation. We then have

$$\begin{aligned} t_1 &= T_t \cdot n_t, \\ t_2 &= T_t \cdot n_t + T_o \cdot n_o. \end{aligned}$$

Solving these two equations gives us the values of T_t and T_o, in turn determines c_t and c_o in Equation (1). □

In general, with a set of calibration queries Q, we first estimate $\mathbf{n}_i$ for each $q_i \in Q$ and then measure its execution time t_i. With $\mathbf{N} = (\mathbf{n}_1, \cdots, \mathbf{n}_k)^\intercal$ and $\mathbf{t} = (t_1, \cdots, t_k)^\intercal$, we can solve the following equation for $\mathbf{c}$:

$$\mathbf{Nc} = \mathbf{t},$$

which is just a system of k equations.

The next question is then, given a set of optimizer cost units $\mathbf{c}$, how to design a set of calibration queries Q. Our goal is to design a set with the following properties:

- *Completeness*: Each cost unit in $\mathbf{c}$ should be covered by at least one calibration query $q \in Q$.
- *Conciseness*: Each query $q \in Q$ should be necessary to guarantee completeness. In other words, any subset $Q' \subset Q$ is not complete.
- *Simplicity*: Each query $q \in Q$ should be as simple as possible when Q is both complete and concise.

Clearly "completeness" is mandatory, while the others are just "desirable". Since the possible number of SQL queries on a given database is infinite, we restrict our attention to concise complete subsets. However, there is still infinite number of sets Q that are both complete and concise. Simpler queries are preferred over more complex, because it is easier to obtain correct values for the cardinalities such as n_t and n_o in Example 2 (getting exact values for such cardinalities may be difficult for operators embedded in deep query trees).

B. Implementation

We designed the 5 calibration queries for the PostgreSQL optimizer as follows. We chose queries q_i for Q by introducing individual cost units one by one:

- *cpu_tuple_cost*: We use query q_1:
  `SELECT * FROM R`
  as the calibration query. The relation `R` is first paged into the buffer pool, and hence there is no I/O cost involved: $\mathbf{n}_1 = (0, 0, n_{t1}, 0, 0)^\intercal$.

- *cpu_operator_cost*: We use query q_2:
  `SELECT COUNT(*) FROM R`
  as another calibration query. We then use the method illustrated in Example 2. Again, `R` is memory resident: $\mathbf{n}_2 = (0, 0, n_{t2}, 0, n_{o2})^\intercal$ and $n_{t2} = n_{t1}$.
- *cpu_index_tuple_cost*: We use query q_3:
  `SELECT * FROM R WHERE R.A < a`
  where `R.A` has a clustered index and we pick `a` so that the optimizer will choose an index scan. This query involves *cpu_tuple_cost*, *cpu_index_tuple_cost*, and *cpu_operator_cost*. Once again, `R` is memory resident: $\mathbf{n}_3 = (0, 0, n_{t3}, n_{i3}, n_{o3})^\intercal$.
- *seq_page_cost*: We use query q_4:
  `SELECT * FROM R`
  as the calibration query. This query will be executed in a sequential scan, and the cost model only involves overhead in terms of *seq_page_cost* and *cpu_tuple_cost*: $\mathbf{n}_4 = (n_{s4}, 0, n_{t4}, 0, 0)^\intercal$.
- *rand_page_cost*: We use query q_5:
  `SELECT * FROM R where R.B < b`
  as the calibration query. Here `R.B` is some attribute of the relation `R` on which an *unclustered* index is built. The values of `B` are uniformly generated, and we pick `b` so that the optimizer chooses an index scan. Ideally, we would like that the qualified tuples were *completely* randomly distributed so that we could isolate the parameter *rand_page_cost*. However, in practice, *pure* random access is difficult to achieve, since the execution subsystem can first determine the pages that need to be accessed based on the qualified tuples before it actually accesses the pages. In this sense, local sequential accesses are unavoidable, and the query plan involves more or less overhead in terms of *seq_page_cost*. In fact, a typical query plan of this query will contain all the five parameters: $\mathbf{n}_5 = (n_{s5}, n_{r5}, n_{t5}, n_{i5}, n_{o5})^\intercal$.

Notice that $\mathbf{n}_i$ can be estimated relatively accurately due to simplicity of q_i. Furthermore, the 5 equations generated by the 5 queries are *independent*, which guarantees the existence of a unique solution for $\mathbf{c}$. This can be easily seen by observing the matrix $\mathbf{N} = (\mathbf{n}_1, \cdots, \mathbf{n}_5)^\intercal$, namely,

$$\mathbf{N} = \begin{pmatrix} 0 & 0 & n_{t1} & 0 & 0 \\ 0 & 0 & n_{t2} & 0 & n_{o2} \\ 0 & 0 & n_{t3} & n_{i3} & n_{o3} \\ n_{s4} & 0 & n_{t4} & 0 & 0 \\ n_{s5} & n_{r5} & n_{t5} & n_{i5} & n_{o5} \end{pmatrix}.$$

Note that the determinant $|\mathbf{N}|$ satisfies $|\mathbf{N}| \neq 0$, since by rearranging the columns of $\mathbf{N}$, we can make it a triangular matrix.

To make this approach more robust, our implementation uses multiple queries for each q_i and finds the best-fitting of $\mathbf{c}$. This is done by picking different relations `R` and different values for the `a` in the predicates of the form `R.A < a`.

IV. Refining Cardinality Estimation

We discuss how to refine $\mathbf{n}$ in this section. To make this paper self-contained, we first discuss how the optimizer obtains $\mathbf{n}$ for a given query plan. We then propose a sampling-based method of refining the cardinality estimates (and hence the $\mathbf{n}$) of the final plan chosen by the optimizer. We describe the details of the algorithm and our current implementation.

A. Optimizer's Estimation of $\mathbf{n}$

The optimizer estimates query execution cost by aggregating the cost estimates of the operators in the query plan. To distinguish blocking and non-blocking operators, this cost model comprises of the *start_cost* and *total_cost* of each operator:

- *start_cost* (sc) is the cost before the operator can produce its first output tuple;
- *total_cost* (tc) is the cost after the operator generates all of its output tuples.

Note that the cost of an operator includes the cost of its child operators.

As an example, we show how $\mathbf{n}$ is derived for the *in-memory sort* and *nested-loop join* operators in PostgreSQL. These operators are representative of blocking and non-blocking operators, respectively. In the following illustration, *run_cost* (rc for short) is defined as $rc = tc - sc$, and N_t is the (estimated) number of input tuples for the operator. Observe that the costs are given as linear combinations of $\mathbf{c}$.

Example 3 (In-Memory Sort): Quick sort is used for tables that optimizer estimates can be completely held in memory. The values *sc* and *rc* are estimated as follows:

$$\begin{aligned} sc &= 2 \cdot c_o \cdot N_t \cdot \log N_t + tc \text{ of child}, \\ rc &= c_t \cdot N_t. \end{aligned}$$

□

Example 4 (Nested-Loop Join): The *nested-loop join* operator joins two input relations. The *sc* and *rc* are estimated as follows:

$$\begin{aligned} sc &= sc \text{ of outer child} + sc \text{ of inner child}, \\ rc &= c_t \cdot N_t^o \cdot N_t^i + N_t^o \cdot rc \text{ of inner child}. \end{aligned}$$

Here N_t^o and N_t^i are the number of input tuples from the outer and inner child operator, respectively. □

Notice that the main uncertainty in $\mathbf{n}$ comes from the estimated input cardinality N_t in both of these examples. In general, the logic flow in the cost models of PostgreSQL optimizer can be summarized with five steps:

1) estimate the input/output cardinality;
2) compute the CPU cost based on cardinality estimates;
3) estimate the number of accessed pages according to the cardinality estimates;
4) compute the I/O cost based on estimates of accessed pages;
5) compute the total cost as the sum of CPU and I/O cost.

Hence, our main task in calibrating $\mathbf{n}$ is to refine the input/output cardinalities for each operator.

B. Cardinality Refinement

As mentioned in the introduction, traditionally cardinality estimation has had to satisfy strict performance constraints because it is done for every plan considered by the optimizer. This has led to compromises that may produce inaccuracies in estimates that are too large for run time estimation.

Our goal is to *refine* the cardinality estimates for the plan chosen by the optimizer. Clearly, this will increase the overhead of the optimization phase. However, the key insight here is that because the refinement procedure only needs to be performed *once* per query, rather than once per plan, we can afford to spend more time than is possible for traditional cardinality estimation.

C. A Sampling-Based Approach

In principle, any approach that can improve cardinality estimation can be applied. We use a generalized version of the *sequential-sampling* estimator proposed in [15] for the following two reasons:

- It incorporates a tunable trade-off between efficiency (i.e., the number of samples taken) and effectiveness (i.e., the precision of the estimates);
- It can simultaneously estimate cardinalities for multiple operators in the query plan.

In this paper, we extend the framework in [15] in the following two aspects:

- The estimator described in [15] is for join queries. We generalize the framework to queries with arbitrary number of selections and joins. We prove that this extension preserves the two key properties, namely, *unbiasedness* and *strong consistency*, of the original estimator.
- The framework described in [15] uses random disk accesses to take samples. In comparison, we propose to take samples offline, store them as materialized views, and directly use them at runtime. This greatly reduces the runtime overhead and requires very minimal changes to the database engine (e.g., a few hundred lines of C code in the case of PostgreSQL). We further show that this offline sampling preserves the semantics of the original online sampling.

We next first describe the estimator in its generalized form, and then describe our cardinality refinement algorithm and its implementation details.

D. The Estimator

Let $\mathcal{D}$ be a database consisting of K relations R_1, ..., R_K. Suppose that R_k is partitioned into m_k blocks each with size N_k, namely, $|R_k| = m_k N_k$. Consider the two basic relational operators: *selection* σ_F (F is a boolean formula representing the selection condition), and *cross-product* $\times$. For σ_F, we define the output of an input block B to be $\sigma_F(B)$. For $\times$, we define the output of the two input blocks B and B' to be $B \times B'$. Instead of estimating the cardinality of the output relation directly, the estimator will estimate the *selectivity* of the operator, which is defined as the output cardinality divided by the input cardinality. Specifically, the selectivity of the selection operator σ_F is $\rho_R = |\sigma_F(R)|/|R|$ where R is the input relation. Moreover, the selectivity of σ_F on a particular block B of R is $\rho_B = |\sigma_F(B)|/|B|$. On the other hand, the selectivity of the cross-product operator $\times$ is always 1. It is then straightforward to obtain the output cardinality once we know the selectivity of the operator[1].

In the following, without loss of generality, we will assume that the query considered is over relations R_1, ..., R_K, and we use the notation $\mathbf{R} = R_1 \times \cdots \times R_K$. Let $B(k, j)$ be the j-th block of relation k ($1 \leq j \leq m_k$, and $1 \leq k \leq K$). We use $\mathbf{B}(L_{1,i_1}, ..., L_{K,i_K})$ to represent $B(1, L_{1,i_1}) \times \cdots \times B(K, L_{K,i_K})$, where $B(k, L_{k,i_k})$ is the block (with index L_{k,i_k}) randomly picked from the relation R_k in the i_k-th sampling step. Moreover, we use the notation $\mathbf{B}_i$ if $i_1 = i_2 = \cdots = i_K = i$.

Lemma 1: Consider $\sigma_F(\mathbf{R})$. Let $\mathbf{B}_1$, ..., $\mathbf{B}_n$ be a sequence of n random samples (with replacement) from $\mathbf{R}$. Define $\rho_{\mathbf{B}_i} = |\sigma_F(\mathbf{B}_i)|/|\mathbf{B}_i|$ ($1 \leq i \leq n$). Then $E\big[\rho_{\mathbf{B}_i}\big] = \rho_{\mathbf{R}}$.

Due to space limitations we defer the proofs of our results to the full version of this paper [1].

Define

$$\tilde{\rho}_{\mathbf{R}} = \frac{1}{n} \sum_{i=1}^{n} \rho_{\mathbf{B}_i}.$$

Then it is easy to see from Lemma 1 that $E\big[\tilde{\rho}_{\mathbf{R}}\big] = \rho_{\mathbf{R}}$. Moreover, since the random variables $\rho_{\mathbf{B}_i}$ are i.i.d., by the strong law of large numbers, we have $Pr\big[\lim_{n\to\infty} \tilde{\rho}_{\mathbf{R}} = \rho_{\mathbf{R}}\big] = 1$. We summarize this result in the following lemma:

Lemma 2: $E\big[\tilde{\rho}_{\mathbf{R}}\big] = \rho_{\mathbf{R}}$, and $Pr\big[\lim_{n\to\infty} \tilde{\rho}_{\mathbf{R}} = \rho_{\mathbf{R}}\big] = 1$.

Lemma 2 can be generalized to queries with arbitrary number of selections and joins:

Theorem 1: Let q be any query involving only selections and joins over $\mathbf{R}$, and let ρ_q be the selectivity of q. Then $E\big[\tilde{\rho}_q\big] = \rho_q$, and $Pr\big[\lim_{n\to\infty} \tilde{\rho}_q = \rho_q\big] = 1$.

In statistical terminology, the estimator $\tilde{\rho}_q$ is *unbiased*, and *strongly consistent* for ρ_q: the more samples we take, the closer $\tilde{\rho}_q$ is to ρ_q. This gives us a way to control the trade-off between the estimation accuracy and the number of samples we take.

The estimator we just described takes samples from each relation uniformly and independently (called *independent sampling* [15]). Therefore, after n steps, we have n observations in total. In [15], the authors further discussed another alternative called *cross-product* sampling. The idea is that, at the i-th step, assuming the K blocks taken from the K relations are $B(1, L_{1,i})$, ..., $B(K, L_{K,i})$, we can actually join each $B(k, L_{k,i})$ with each $B(k', L_{k',i'})$ such that $1 \leq i' \leq i$ and

[1]Note that the input cardinality is already known before the estimation procedure runs. It is simply the product of the cardinalities of the underlying relations that are input to the operator, which can be directly obtained from the statistics stored in system catalogs.

$k' \neq k$ (note that in the case of independent sampling, we only join among the blocks with $i' = i$). In this way, we can obtain n^K observations after n steps.

Define

$$\tilde{\rho}_{\mathbf{R}}^{cp} = \frac{1}{n^K} \sum_{i_1=1}^{n} \cdots \sum_{i_K=1}^{n} \rho_{\mathbf{B}(L_{1,i_1},\cdots,L_{K,i_K})}. \quad (2)$$

From Lemma 1, it is clear that $\tilde{\rho}_{\mathbf{R}}^{cp}$ is still unbiased, i.e., $E\big[\tilde{\rho}_{\mathbf{R}}^{cp}\big] = \rho_{\mathbf{R}}$. However, since now the $\rho_{\mathbf{B}(L_{1,i_1},\cdots,L_{K,i_K})}$'s are no longer independent, we cannot directly apply the strong law of large numbers to show the strong consistency of $\tilde{\rho}_{\mathbf{R}}^{cp}$, although it still holds here:

Lemma 3: $E\big[\tilde{\rho}_{\mathbf{R}}^{cp}\big] = \rho_{\mathbf{R}}$, and $Pr\big[\lim_{n\to\infty} \tilde{\rho}_{\mathbf{R}}^{cp} = \rho_{\mathbf{R}}\big] = 1$.

Therefore, Theorem 1 still holds in the case of cross-product sampling. It is also shown that cross-product sampling is always superior to independent sampling because of its lower sample variance (see Theorem 2 of [15]). Therefore, our cardinality refinement algorithm discussed next is based on cross-product sampling instead of independent sampling.

E. The Cardinality Refinement Algorithm

There are several considerations when designing our refinement algorithm based on the sampling-based estimator.

First, the estimator needs to access disk to take samples. However, since samples should be randomly taken, this means significant random reads may be required during the sampling phase, which may be too costly in practice. To overcome this issue, as has been suggested in previous applications of sampling in DBMS (e.g., [25]), we take samples offline and store them as materialized views in the database. We found in our experiments that the number of samples required is quite small and therefore can be cached in memory during runtime.

Second, the estimator we discussed so far focuses on estimating the selectivity (or equivalently, cardinality) for a single operator. However, in practice, a query plan may contain more than one operator, and for our purpose of refining the cost estimates of this plan, we need to estimate the cardinality for each operator. Another good property of the estimator is that, for a query plan with a fixed join order, which is always the case when refinement is performed, we can estimate all selection and join operators in the plan simultaneously. Consider, for example, a three-way join query $q = R_1 \bowtie R_2 \bowtie R_3$. We need to estimate the cardinality for both $q' = R_1 \bowtie R_2$ and q. However, after we are done with q', we can estimate for q by directly evaluating $q' \bowtie R_3$. This means, we can estimate the cardinality for each operator by simply invoking the query plan q over the sample relations and then apply the estimator to each operator (see Theorem 2 below).

Theorem 2: Let $q = \sigma_F(R_1 \times \cdots \times R_K)$ be an arbitrary query with only selections and joins. For every subquery $q_i = \sigma_{F_i}(R_1 \times \cdots \times R_i)$ ($1 \leq i \leq K$, and F_i is the selection condition only involving R_1, ..., R_i), $E\big[\tilde{\rho}_{q_i}\big] = \rho_{q_i}$ and $Pr\big[\lim_{n\to\infty} \tilde{\rho}_{q_i} = \rho_{q_i}\big] = 1$.

Third, while the estimator discussed above is both unbiased and strongly consistent, it only works for queries involving selections and joins. In practice, SQL queries can contain additional operators. A particularly common class of such operators we encountered in TPC-H queries is *aggregates*, for which we need to estimate the number of distinct values in the input relation. Aggregates basically *collapse* the underlying data distribution, so the estimator cannot work for queries containing aggregates. As a result, we can only apply the sampling-based estimator to the part of the query plan that does not involve aggregates. For aggregate operators, we simply rely on PostgreSQL's models for estimating output cardinalities. However, note that, since the refinement phase may change the input estimates for the aggregate, the output estimates for the aggregate may change as well. We observed in our experimental evaluation (see Section V) that the current approach already leads to promising prediction of execution time in practice. We leave the problem of further integrating state-of-the-art estimators (e.g., the GEE estimator in [6]) for estimating the number of distinct values as future work.

Our cardinality refinement algorithm is illustrated in Algorithm 1. For the input query q, we first call the optimizer to obtain its query plan P_q (line 30). We then modify P_q by replacing the relations it touches with the corresponding sample relations (i.e., materialized views), and run P_q over the sample relations (line 31 to 34). After that, we call the procedure $RecomputeCardinality$ to refine the cardinality estimation for each operator in P_q (line 36).

The procedure $RecomputeCardinality$ (line 11 to 27) works as follows. It first invokes $EstimateCardinality$ on the child operators (if any) of the current operator O (line 12 to 17). It then checks the flag *HasAgg*, which indicates whether O has any *descendant* operator that is an aggregate. If the flag is set, then it simply calls the optimizer's own model to do cardinality estimation for O (line 18 to 19), since our estimator refinement cannot be applied in this case, as discussed above. If, on the other hand, the flag is not set, then it further checks whether O is itself an aggregate. If so, it again calls the optimizer's cardinality estimation model for O, and sets the flag *HasAgg* (line 21 to 23). If not, it invokes $EstimateCardinality$ to estimate the cardinality of O (line 25). Note that due to the order that $EstimateCardinality$ is invoked, the estimator is applied to each operator in a bottom-up manner. This guarantees that the input cardinality of any operator will be estimated after its child operators (if any). While this is not necessary if we only need to refine the cardinalities, it is necessary since we need to further estimate based on the cardinality (for example, the number of pages accessed), which enforces the same bottom-up ordering here since the cost of an operator covers the cost of its child operators as well (recall Example 3 and 4).

The procedure $EstimateCardinality$ (line 3 to 9) implements the estimator. $GetSubPlan$ returns the subtree P_O of the query plan with the current operator O as the root. N_s is the product of the cardinalities of the sample relations involved in P_O, and E_s is the exact output cardinality of O when

Algorithm 1: Cardinality Refinement

Input: q, a SQL query
Output: P_q, query plan of q with refined cardinalities

HasAgg $\leftarrow$ *False*;

EstimateCardinality(O):
$P_O \leftarrow GetSubPlan(O)$;
$N_s \leftarrow \prod_{R^s \in SampleRelations(P_O)} |R^s|$;
$E_s \leftarrow CardinalityBySampling(O)$;
$N_O \leftarrow \prod_{R_O \in Relations(P_O)} |R_O|$;
$E_O \leftarrow N_O \cdot \frac{E_s}{N_s}$;
Treat E_O as the cardinality estimate for O;

RecomputeCardinality(O):
if *O has left child O_{lc}* **then**
  $EstimateCardinality(O_{lc})$;
end
if *O has right child O_{rc}* **then**
  $EstimateCardinality(O_{rc})$;
end
if *HasAgg* **then**
  Use optimizer's model to estimate for O;
else
  if *O is aggregate* **then**
    Use optimizer's model to estimate for O;
    HasAgg $\leftarrow$ *True*;
  else
    $EstimateCardinality(O)$;
  end
end

Main:
$P_q \leftarrow GetPlanFromOptimizer(q)$;
foreach $R \in Relations(P_q)$ **do**
  Replace R with its sample relation R^s;
end
Run the plan P_q over the sample relations;
$O \leftarrow GetRootOperator(P_q)$;
$RecomputeCardinality(O)$;
return P_q;

the plan is evaluated over sample relations. Therefore, the estimated selectivity is $\frac{E_s}{N_s}$, and the output cardinality of O over the original relations is then $N_O \cdot \frac{E_s}{N_s}$, where N_O is the product of the cardinalities of the original input relations. In Theorem 3, we further show that $EstimateCardinality$ is a particular implementation of the estimator conforming to the semantics of cross-product sampling, with a special *tuple-level* partitioning scheme, where each block contains only a single tuple of the relation.

Theorem 3: The procedure $EstimateCardinality$ estimates the cardinality of the operator O according to the semantics of cross-product sampling.

V. Evaluation

In this section, we describe our experimental settings and report our results. Due to space limitations, some of the results are omitted. The complete experimental results are included in [1].

A. Experimental Settings

We implemented Algorithm 1 inside PostgreSQL 9.0.4 by modifying the query optimizer. In addition, we added instrumentation code to the optimizer to collect the input cardinalities for each operator. Our software setup was PostgreSQL on Linux Kernel 2.6.18, and we tested our method on both TPC-H 1GB and 10GB databases.

Our experiments were conducted on two different hardware configurations:

- *PC1*: configured with a 1-core 2.27GHz Intel CPU and 2GB memory;
- *PC2*: configured with an 8-core 2.40GHz Intel CPU and 16GB memory.

We randomly drew 10 queries from each of the 21 query templates[2], and we ran each query 5 times. Our error metric is computed based on the mean execution time of the queries. We cleared both the filesystem and DB buffers between each run of each query.

Since the original TPC-H database generator uses uniform distributions, to test the robustness of different approaches on different data distributions, we also used a skewed TPC-H database generator [2]. This database generator populates a TPC-H database using a Zipf distribution. This distribution has a parameter z that controls the degree of skewness. $z = 0$ generates a uniform distribution, and as z increases, the data becomes more and more skewed. We created skewed databases generated using $z = 1$.

B. Calibrating Cost Units

We use the approach described in Section III to generate calibration queries. The calibrated values for the 5 PostgreSQL optimizer parameters on PC1 and PC2 are shown in Table I and II, respectively. Except for *rand_page_cost*, the cost units show very small variance when profiled under different relations.

Calibrating the *rand_page_cost* is more difficult. As discussed in Section III-B, achieving purely random reads in a query appears difficult in practice. In addition, the number of random pages accessed as estimated by optimizer is based on statistics about correlations between the order of keys stored in the unclustered index and their actual order in the corresponding data file. Therefore, there is some inherent uncertainty in this estimation. It is interesting future work to see whether the *rand_page_cost* could be calibrated more accurately with different methods than the one described in this paper.

[2]We excluded the template Q15 because it creates a view before the query runs, which is not supported by our current implementation of Algorithm 1.

Optimizer Parameter	Calibrated $\mu \pm \sigma$ (ms)	Default
seq_page_cost	5.53e-2 ± 3.09e-3	1.0
rand_page_cost	6.50e-2 ± 2.32e-2	4.0
cpu_tuple_cost	1.67e-4 ± 5.83e-6	0.01
cpu_index_tuple_cost	3.41e-5 ± 2.30e-5	0.005
cpu_operator_cost	1.12e-4 ± 1.30e-6	0.0025

TABLE I
ACTUAL VALUES OF POSTGRESQL OPTIMIZER PARAMETERS ON PC1

Optimizer Parameter	Calibrated $\mu \pm \sigma$ (ms)	Default
seq_page_cost	5.03e-2 ± 3.82e-3	1.0
rand_page_cost	4.89e-1 ± 7.44e-2	4.0
cpu_tuple_cost	1.41e-4 ± 1.35e-5	0.01
cpu_index_tuple_cost	3.34e-5 ± 3.85e-5	0.005
cpu_operator_cost	7.10e-5 ± 1.52e-5	0.0025

TABLE II
ACTUAL VALUES OF POSTGRESQL OPTIMIZER PARAMETERS ON PC2

Note that the default settings of the parameters fail to accurately reflect the actual relative magnitudes. For example, on PC1, the ratio of calibrated *cpu_tuple_cost* to *seq_page_cost* is about 0.003 instead of 0.01.

Clearly, the overhead of this profiling stage depends on how many calibration queries we use. In our experiments on the TPC-H database, we used the 5 largest relations as the `R` in `SELECT * FROM R` and `SELECT COUNT(*) FROM R`, respectively. For `SELECT * FROM R WHERE R.A < a`, we used the largest relation (*lineitem*), and generated 10 queries where the predicate `R.A < a` had different selectivities in each. Under this setting, the profiling stage usually finishes in less than an hour, which is substantially less than the long training stage of machine-learning-based approaches.

Moreover, our profiling stage was conducted on top of the uniform TPC-H database. Note that we do not need to run it again for the skewed TPC-H database. This is because the values of the cost units only depend on the specific hardware configuration. After the cost units are calibrated, they can be used as long as the hardware configuration does not change. On the other hand, machine-learning-based approaches usually need to collect new training data and rebuild the predictive model if the underlying data distribution significantly changes.

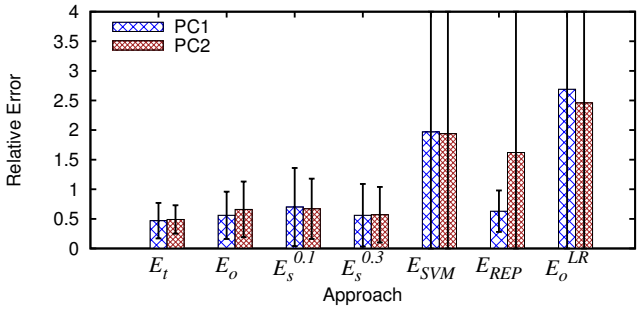
C. Prediction Results

We evaluated the accuracy of prediction in terms of the *mean relative error* (MRE), a metric used in [4]. MRE is defined as

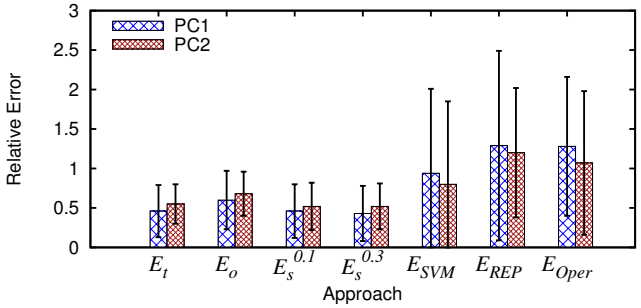
$$\frac{1}{M}\sum_{i=1}^{M}\frac{|T_i^{pred} - T_i^{act}|}{T_i^{act}},$$

where M is the number of testing queries, T_i^{pred} and T_i^{act} are the predicted and actual execution time of the testing query i, respectively.

We compare the prediction accuracy of our approach with several state-of-the-art machine-learning-based solutions: *Plan-level modeling with SVM* [4], *Plan-level modeling with REP trees* [31], and *Operator-level modeling with Multivariate Linear Regression (MLR)* [4]. We use the same set of features as described in [4]. We focus on the settings of the so-called *dynamic workload* in [4]. The idea of plan-level modeling



(a) Without E_{Oper}, 21 templates



(b) With E_{Oper}, 11 templates

Fig. 2. Uniform TPC-H 1GB database

was also tried in [12], and the authors chose to use Kernel Canonical Correlation Analysis (KCCA) [5] instead of SVM as the machine-learning approach. We do not compare our techniques with theirs, for it has been shown in [4] that both the plan-level and operator-level modeling approach of [4] are superior to the KCCA-based approach for dynamic workloads.

To generate a dynamic workload, we conducted the following "leave-one-template-out" experiment as in [4]. Among the N TPC-H query templates, we chose one template to generate the queries whose execution time is to be predicted, and the other *N-1* templates were used to generate the training queries used by the machine learning methods to train their predictive models.

Figure 2 shows the results on the uniform (i.e., $z = 0$) 1GB TPC-H database. As presented in [4], operator-level modeling requires that the testing queries do not contain operators not used by the training queries. Since some TPC-H templates include specific operators not found in the other templates (e.g., *hash-semi-join*), we excluded these templates from our experiments. The same argument applies to TPC-H templates containing PostgreSQL-specific structures (i.e., `INITPLAN` and `SUBQUERY`). The authors of [4] also excluded these queries for the same reason. However, we note here that this is a problem due to the particular choice of the workload and database system, not due to the operator-level modeling itself. If TPC-H were a more varied workload, we would not have this restriction. For example, if it had multiple queries that used the hash-semi-join operator, we could have incorporated queries with that operator in our experiments. This leaves 11 TPC-H templates participating in the dynamic workload

experiment when operator-level modeling is leveraged (see Figure 2(b)).

In Figure 2, the x-axis represents the approaches we tested in the experiments, and the y-axis shows the average error and the standard deviation (shown with the error bars) over the TPC-H templates. Here, E_t is the prediction error of our approach when the *true* cardinalities are used (the true cardinalities are measured in an artificial "pre-running" of the query — we present this number to provide insight into what could be achievable if we were able to get perfect cost estimates). E_o is the prediction error of our approach when the cardinalities from the optimizer are used without refinement. E_s^f is the prediction error of our approach when the cardinalities are estimated via sampling, where f is the sampling ratio (e.g., $f = 0.1$ means we take a 10% sample from each underlying table). In our experiments, we tested sampling ratios $f = 0.05, 0.1, 0.2, 0.3, 0.4$. Due to space limitations, we only present the results of $f = 0.1$ and $f = 0.3$. E_{SVM}, E_{REP}, and E_{Oper} are the prediction errors of the three machine-learning based approaches, i.e., plan-level modeling with SVM, plan-level modeling with REP, and operator-level modeling, respectively. Finally, as a baseline, E_o^{LR} is the prediction error of mapping the original cost estimates from the optimizer to the execution time via linear regression, as was done in previous work.

We have several observations. First, in the case of uniform data, the cost models with properly tuned c's already work well (the E_o in Figure 2 is close to E_t). Sampling does not help much in improving the prediction accuracy. This is reasonable, because the assumptions like uniformity and independence leveraged by the optimizer for cardinality estimation usually hold in this case.

Second, the performance of machine-learning based approaches is not consistent. For some queries, their predictions are good. However, for the other queries, their predictions are far away from the true values. This can be observed by noticing the big error bars in the figures. As an example, the E_{SVM} in Figure 2(a) varies between 0.03 for Q7 and 12.16 for Q17 (see [1]). One possible reason for this is: most machine learning methods assume that the testing queries should be *similar* to the queries used in training the model. More specifically, the feature vectors of the testing queries should be close to those feature vectors of the training queries, in terms of the distance in the feature space. Unfortunately, this assumption is not valid for dynamic workloads.

To see this, we further apply Principal Component Analysis (PCA) [17] on the query features (47 features in total) and project the queries onto the subspace spanned by the three most dominating principal components. It is well known that these dominating components reveal the *major* directions in the feature space. While it is true that more principal components are better, we restrict ourselves to 3-dimensional space for the purpose of visualization.

Figure 3 shows the queries in the projected space, where each combination of color and shape represents one query template. From the figure we can see that the templates can

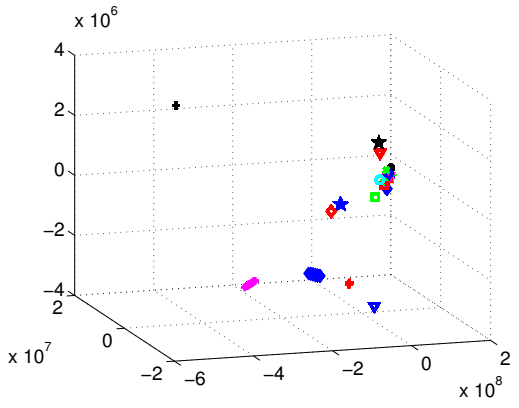


Fig. 3. Queries projected on the 3 dominating principal components

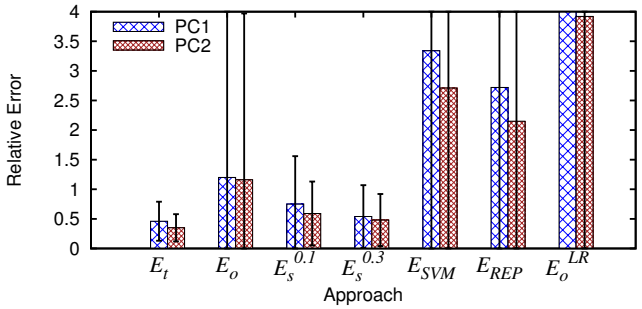
be grouped into several clusters. About half of the templates fall into the rightmost cluster, and each of the remaining templates usually forms a singleton cluster. The distances between clusters are quite big. Note that PCA will not increase the distances between feature vectors after the projection, which means the distances between feature vectors in the original 47-dimensional space can only be the same or even bigger. This suggests that there is little similarity among the TPC-H templates within different clusters. Therefore, if we test the queries from a template within a singleton cluster, by using the model trained with the other templates, then there is little hope for us to observe good predictions.

Third, machine-learning approaches are sensitive to the set of queries used in training. In [1], we show that the prediction errors for some queries fluctuate dramatically when different sets of training queries are used[3]. For instance, E_{REP} for Q8 is 0.44 when 20 templates are used in training (as in Figure 2(a)), but it will increase to 2.87 when only 10 templates are used (as in Figure 2(b)). Picking a set of proper training queries hence is critical in practice when using machine-learning approaches. However, it seems quite difficult in the environment when workload is not known in advance.

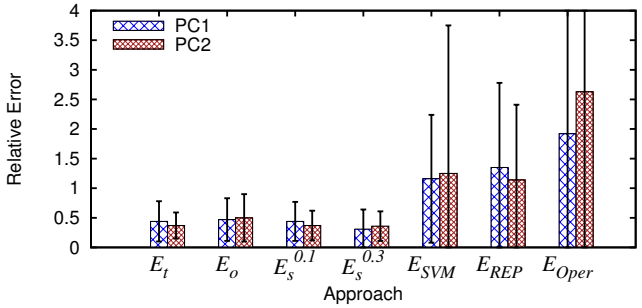
Figure 4 further presents the results on the skewed TPC-H 1GB database. As expected, when data becomes skewed, the cardinality estimates from the optimizer become inaccurate, and hence the predictive power is weakened. However, by leveraging the sampling-based cardinality correction, the prediction accuracy is improved. Moreover, more samples usually mean better prediction accuracy, as long as the overhead on sampling is acceptable (see Section V-D). We note that the sampling overhead can be up to 20% on this data set. As we will see, this can be viewed as a problem that arises on small data sets, as the overhead due to sampling for the 10GB data set is much lower. On the other hand, the performance of machine-learning based approaches becomes even worse. This is perhaps partially because of the worse distortion of the assumption that training and testing queries should be similar.

Similar results on the TPC-H 10GB database are observed

[3]Recall that, to be fair, we only use 10 templates in training when comparing with operator-level modeling (as in Figure 2(b)).

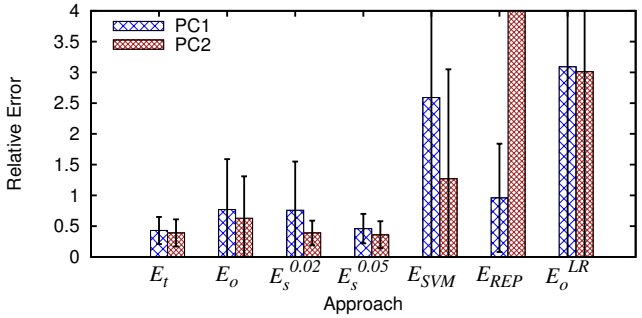


(a) Without E_{Oper}, 21 templates

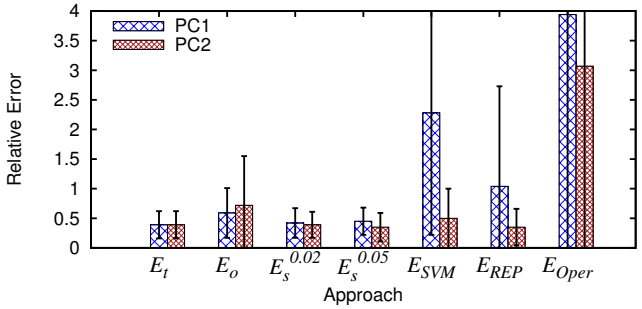


(b) With E_{Oper}, 11 templates

Fig. 4. Skewed TPC-H 1GB database



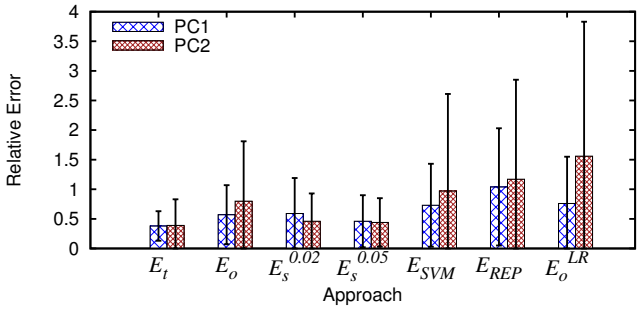
(a) Without E_{Oper}, 21 templates



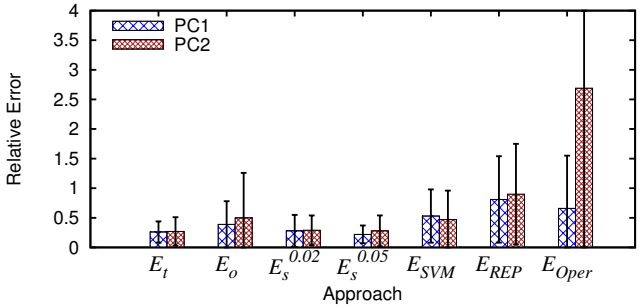
(b) With E_{Oper}, 11 templates

Fig. 5. Uniform TPC-H 10GB database

in our experiments, as shown in Figure 5 and Figure 6. Here, to make the overall experiment time controllable, as done in [4], we kill the query if it runs longer than an hour. This leaves us with 18 templates participating in the evaluation. We tested sampling ratios $f = 0.01, 0.02, 0.05, 0.1$, and present



(a) Without E_{Oper}, 21 templates



(b) With E_{Oper}, 11 templates

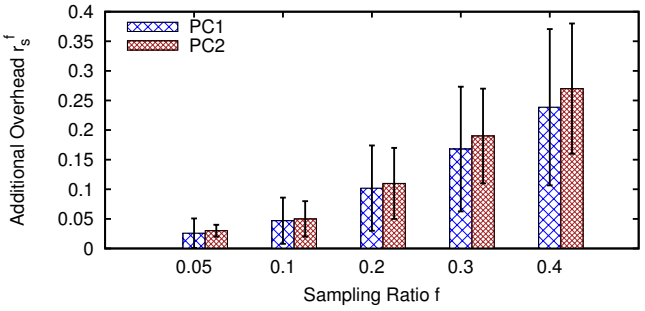
Fig. 6. Skewed TPC-H 10GB database

the results of $f = 0.02$ and $f = 0.05$, for space constraints. Note that, while the database size scales up by a factor of 10, the required absolute number of samples to achieve predictions close to the ideal case (compare $E_s^{0.05}$ and E_t in the figures) remains almost the same. We need $0.05 \times 10GB = 0.5GB$ samples here, while we need $0.3 \times 1GB = 0.3GB$ samples in the case of 1GB database. Therefore, the additional overhead of taking samples becomes ignorable when the database is larger (see Section V-D).

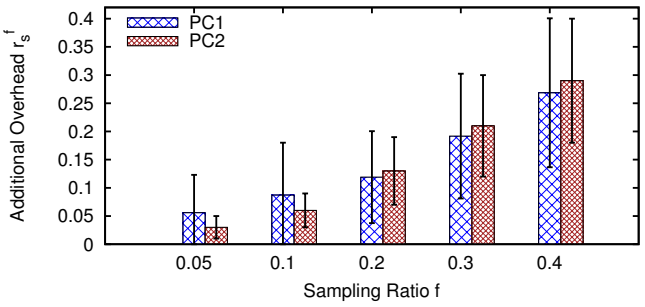
D. Overhead of Sampling

Figure 7 shows the additional runtime overhead due to sampling for the 1GB TPC-H database. Here r_s^f is defined as $r_s^f = T_s/T$, where T_s and T are the time to run the queries over the sample tables and original tables, respectively, and f is the sampling ratio as before. For each sampling ratio, we report the average r_s^f as well as the standard deviation (shown with the error bars) over the participating TPC-H templates.

We can see that for the sampling ratio $f = 0.3$, which allows us to achieve close prediction accuracy to what if the true cardinalities were used on both the uniform and skewed data, the average additional runtime overhead is around 20% of the actual execution time of the query. Note that for query optimization, 20% is prohibitively high. For example, it means that we can only consider $1/20\% = 5$ plans during optimization before the estimation cost dominates the query execution time, since sampling should be invoked for every query plan considered. But for our purposes, where we are trying to estimate the running time of a single query plan, this amount of overhead may be acceptable. Perhaps more

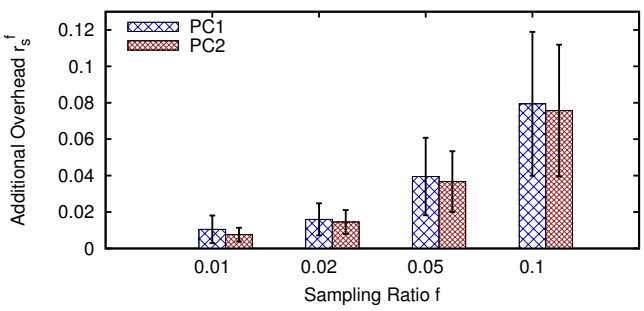


(a) On uniform data

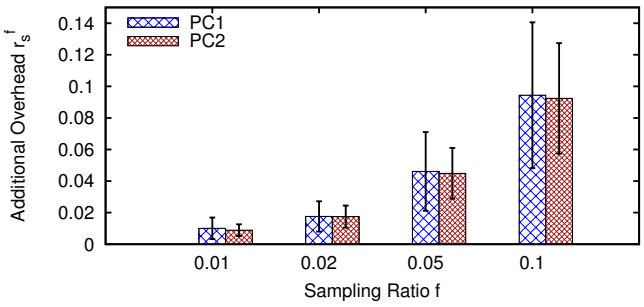


(b) On skewed data

Fig. 7. Additional overhead of sampling on TPC-H 1GB database



(a) On uniform data



(b) On skewed data

Fig. 8. Additional overhead of sampling on TPC-H 10GB database

importantly, this overhead drops dramatically when we move to the 10GB data set.

Figure 8 further presents the results over 10GB TPC-H database. It confirms that the additional overhead introduced by sampling is even smaller, compared with the overhead of running the original query. For the case where good prediction can be achieved (i.e., $f = 0.05$, see Figure 5 and Figure 6), the additional overhead is below 4% on average. This demonstrates the practicality of incorporating sampling for the purpose of query time prediction.

VI. Related Work

Query optimizers have built-in cost models that provide cardinality/cost estimates for a given query. There is a lot of previous work on this topic, including methods based on sampling (e.g., [18], [22]), methods based on histograms (e.g., [19]), methods based on machine learning (e.g., [13], [29]), and methods based on using execution feedback (e.g., [8], [27]). However, the purpose of these estimates is to help the optimizer pick a relatively good plan, not to predict the *actual* execution time of the query. Therefore, these estimates need not to be very accurate as long as the optimizer can leverage them to distinguish good plans from bad ones. As shown in [4], [12], without proper calibration, directly leveraging these cost estimates cannot provide good predictions of execution time. Nonetheless, it would be very interesting future work to see the effectiveness by incorporating some methods other than sampling into our current framework for refining cardinality estimates. For example, recent work [29] presented an efficient approach based on graphical models, which was reported to have an order of magnitude better selectivity estimates.

Previous work has explored the issue of calibrating optimizer parameters, for different purposes such as query optimization in heterogeneous DBMS [10], DB resource virtualization [26], and storage type selection [32]. To the best of our knowledge, we are not aware of any work that tries to predict the execution time of SQL queries based on calibrating the cost models of query optimizers.

Another related research direction is *query progress indicators* [7], [20], [21], [23]. The task of a progress indicator is to dynamically monitor the percentage of work that has been done so far for the query. The key difference from query execution time prediction is that progress indicators usually rely on *runtime* statistics obtained *during* the actual execution of the query, which are not available if the prediction is restricted to be made before query execution. A query running time predictor could be useful in providing the very first estimate for a progress indicator (one used before the query starts executing).

Quite surprisingly, the problem of predicting *actual* execution time of a query has been specifically addressed only recently [12]. Existing work [3], [4], [11], [12], [28] usually employs predictive frameworks based on statistical machine learning techniques.

In [12], each query is represented as a set of features containing an instance count and cardinality sum for each possible operator. Kernel Canonical Correlation Analysis (KCCA) [5] modeling techniques are then used to map the queries from their feature space onto their performance space. One main limitation of this approach is that its prediction is based on taking the average of the k (usually 3) nearest neighbors in the training set, which means that the prediction can never exceed

the longest execution time observed during training stage. Hence, when the query to be predicted takes significantly longer time than all the training queries observed, the model is incapable of giving reasonable predictions.

In [4], a similar idea of using features extracted from the entire plan to represent a query is leveraged, and the authors propose to use SVM instead of KCCA. However, the SVM approach still suffers from the same generalization problem. To alleviate this, the authors further apply this idea at the operator-level. But from the reported experimental results (both in [4] and Section V of this paper), it seems that operator-level modeling is still quite vulnerable to workload changes. Our approach in this paper avoids this generalization problem, for it does not rely on any particular training queries.

In [11], the authors study the problem of predicting the execution time when the query is concurrently running with the other queries, with a linear multivariate regression model to capture the interaction between queries. Similar problems in admission control and query scheduling are also studied in [28] and [3], respectively. One key limitation of this line of work is that they all assume a *closed-world* workload scenario, where all possible queries are needed to be known in ahead, which is hardly to be the case in practice. Although this paper does not address the prediction problem in the presence of concurrent query execution, it is interesting to see how to extend the techniques here to provide alternative solutions to this challenge.

VII. Conclusion

In this paper, we studied the problem of leveraging optimizer's cost models to predict query execution time. We show that, after proper calibration, the current cost models used by query optimizers can be more effective for predicting query execution time than reported by previous work.

Of course, it is possible that a new machine learning technique, perhaps with improved feature selection, will outperform the techniques presented here. On the other hand, further improvements are also possible in optimizer-based running time prediction. Perhaps the most interesting aspect of this work is the basic question it raises: should query running time prediction treat the DBMS as a black box (the machine learning approach), or should we exploit the fact that we actually know exactly what is going on inside the box (the optimizer based approach)? We regard this paper as an argument that the latter approach shows promise, and expect that exploring the capabilities of the two very different approaches will be fertile ground for future research.

Acknowledgements

We thank the anonymous reviewers for their valuable comments. The work described in this paper is part of the CloudDB project [16] at NEC Laboratories America.

References

[1] http://pages.cs.wisc.edu/~wentaowu/papers/prediction-full.pdf.

[2] Skewed tpc-h data generator. ftp://ftp.research.microsoft.com/users/viveknar/TPCDSkew/.

[3] M. Ahmad, A. Aboulnaga, S. Babu, and K. Munagala. Interaction-aware scheduling of report-generation workloads. *The VLDB Journal*, 20:589–615, 2011.

[4] M. Akdere, U. Çetintemel, M. Riondato, E. Upfal, and S. Zdonik. Learning-based query performance modeling and prediction. In *ICDE*, 2012.

[5] F. R. Bach and M. I. Jordan. Kernel independent component analysis. *Journal of Machine Learning Research*, 3:1–48, 2002.

[6] M. Charikar, S. Chaudhuri, R. Motwani, and V. R. Narasayya. Towards estimation error guarantees for distinct values. In *PODS*, pages 268–279, 2000.

[7] S. Chaudhuri, V. R. Narasayya, and R. Ramamurthy. Estimating progress of execution for SQL queries. In *SIGMOD*, 2004.

[8] S. Chaudhuri, V. R. Narasayya, and R. Ramamurthy. Diagnosing estimation errors in page counts using execution feedback. In *ICDE*, pages 1013–1022, 2008.

[9] Y. Chi, H. J. Moon, and H. Hacıgümüş. iCBS: Incremental costbased scheduling under piecewise linear slas. *PVLDB*, 4(9):563–574, 2011.

[10] W. Du, R. Krishnamurthy, and M.-C. Shan. Query optimization in a heterogeneous dbms. In *VLDB*, pages 277–291, 1992.

[11] J. Duggan, U. Çetintemel, O. Papaemmanouil, and E. Upfal. Performance prediction for concurrent database workloads. In *SIGMOD*, 2011.

[12] A. Ganapathi, H. A. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. I. Jordan, and D. A. Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *ICDE*, 2009.

[13] L. Getoor, B. Taskar, and D. Koller. Selectivity estimation using probabilistic models. In *SIGMOD*, 2001.

[14] S. Guirguis, M. A. Sharaf, P. K. Chrysanthis, A. Labrinidis, and K. Pruhs. Adaptive scheduling of web transactions. In *ICDE*, 2009.

[15] 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.

[16] H. Hacıgümüş, J. Tatemura, W.-P. Hsiung, H. J. Moon, O. Po, A. Sawires, Y. Chi, and H. Jafarpour. CloudDB: One size fits all revived. In *SERVICES*, 2010.

[17] T. Hastie, R. Tibshirani, and J. H. Friedman. *The Elements of Statistical Learning*. Springer, 2003.

[18] W.-C. Hou and G. Ozsoyoglu. Statistical estimators for aggregate relational algebra queries. *ACM Trans. Database Syst.*, 16:600–654, 1991.

[19] Y. E. Ioannidis. The history of histograms (abridged). In *VLDB*, pages 19–30, 2003.

[20] A. C. König, B. Ding, S. Chaudhuri, and V. R. Narasayya. A statistical approach towards robust progress estimation. *PVLDB*, 5(4):382–393, 2011.

[21] J. Li, R. V. Nehme, and J. F. Naughton. GSLPI: A cost-based query progress indicator. In *ICDE*, pages 678–689, 2012.

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

[23] G. Luo, J. F. Naughton, C. J. Ellmann, and M. Watzke. Toward a progress indicator for database queries. In *SIGMOD*, 2004.

[24] C. Mishra and N. Koudas. The design of a query monitoring system. *ACM Trans. Database Syst.*, 34(1), 2009.

[25] R. Ramamurthy and D. J. DeWitt. Buffer-pool aware query optimization. In *CIDR*, pages 250–261, 2005.

[26] A. A. Soror, U. F. Minhas, A. Aboulnaga, K. Salem, P. Kokosielis, and S. Kamath. Automatic virtual machine configuration for database workloads. In *SIGMOD*, 2008.

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

[28] S. Tozer, T. Brecht, and A. Aboulnaga. Q-Cop: Avoiding bad query mixes to minimize client timeouts under heavy loads. In *ICDE*, 2010.

[29] K. Tzoumas, A. Deshpande, and C. S. Jensen. Lightweight graphical models for selectivity estimation without independence assumptions. *PVLDB*, 4(11):852–863, 2011.

[30] T. J. Wasserman, P. Martin, D. B. Skillicorn, and H. Rizvi. Developing a characterization of business intelligence workloads for sizing new database systems. In *DOLAP*, 2004.

[31] P. Xiong, Y. Chi, S. Zhu, J. Tatemura, C. Pu, and H. Hacıgümüş. ActiveSLA: a profit-oriented admission control framework for database-as-a-service providers. In *SOCC*, 2011.

[32] N. Zhang, J. Tatemura, J. M. Patel, and H. Hacıgümüş. Towards cost-effective storage provisioning for DBMSs. *PVLDB*, 5(4):274–285, 2011.