Uncertainty Aware Query Execution Time Prediction

Wentao Wu[†] Xi Wu[†] Hakan Hacıgümüş[‡] Jeffrey F. Naughton[†]
[†]Department of Computer Sciences, University of Wisconsin-Madison
[‡]NEC Laboratories America
{wentaowu, xiwu, naughton}@cs.wisc.edu, hakan@nec-labs.com

ABSTRACT

Predicting query execution time is a fundamental issue underlying many database management tasks. Existing predictors rely on information such as cardinality estimates and system performance constants that are difficult to know exactly. As a result, accurate prediction still remains elusive for many queries. However, existing predictors provide a single, point estimate of the true execution time, but fail to characterize the uncertainty in the prediction. In this paper, we take a first step towards providing uncertainty information along with query execution time predictions. We use the query optimizer's cost model to represent the query execution time as a function of the selectivities of operators in the query plan as well as the constants that describe the cost of CPU and I/O operations in the system. By treating these quantities as random variables rather than constants, we show that with low overhead we can infer the distribution of likely prediction errors. We further show that the estimated prediction errors by our proposed techniques are strongly correlated with the actual prediction errors.

1. INTRODUCTION

The problem of predicting query execution time has received a great deal of recent research attention (e.g., [4, 5, 16, 17, 38, 39]). Knowledge about query execution time is essential to many important database management issues, including query optimization, admission control [40], query scheduling [11], and system sizing [36]. Existing predictors rely on information such as cardinality estimates and system performance constants that are difficult to know exactly. As a result, accurate prediction remains elusive for many queries. However, existing predictors provide a single, point estimate of the true execution time, but fail to characterize the *uncertainty* in the prediction.

It is a general principle that if there is uncertainty in the estimate of a quantity, systems or individuals using the estimate can benefit from information about this uncertainty. (As a simple but ubiquitous example, opinion polls cannot be reliably interpreted without considering the uncertainty bounds on their results.) In view of this, it is somewhat surprising that something as foundational as query running time estimation typically does not provide any information about the uncertainty embedded in the estimates.

There is already some early work indicating that providing this uncertainty information could be useful. For example, in approximate query answering [22, 24], approximate query results are accompanied by error bars to indicate the confidence in the estimates. It stands to reason that other user-facing running time estimation tasks, for example, query progress indicators [10, 27], could also benefit from similar mechanisms regarding uncertainty. Other examples include robust query processing and optimization techniques (e.g., [7, 18, 29, 35]) and distribution-based query schedulers [11]. We suspect that if uncertainty information were widely available many more applications would emerge.

In this paper, we take a first step towards providing uncertainty information along with query execution time predictions. In particular, rather than just reporting a point estimate, we provide a distribution of likely running times. There is a subtlety in semantics involved here — the issue is not "if we run this query 100 times what do we think the distribution of running times will be?" Rather, we are reporting "what are the likelihoods that the actual running time of this query would fall into certain confidence intervals?" As a concrete example, the distribution conveys information such as "I believe, with probability 70%, the running time of this query should be between 10s and 20s."

Building on top of our previous work [39], we use query optimizers' cost models to represent the query execution time as a function of selectivities of operators in the query plan as well as basic system performance parameters such as the unit cost of a single CPU or I/O operation. However, our approach here is different from that in [39] — we treat these quantities as random variables rather than fixed constants. We then use sampling based approaches to estimate the distributions of these random variables. Based on that, we further develop analytic techniques to infer the distribution of likely running times.

In more detail, for specificity consider the cost model used by the query optimizer of PostgreSQL:

EXAMPLE 1 (POSTGRESQL'S COST MODEL). *PostgreSQL estimates the execution runtime overhead t_O of an operator O (e.g., scan, sort, join, etc.) as follows:*

$$t_O = n_s \cdot c_s + n_r \cdot c_r + n_t \cdot c_t + n_i \cdot c_i + n_o \cdot c_o. \quad (1)$$

Here the c's are *cost units* described in Table 1. Accordingly, the n's are then the number of pages sequentially scanned, the number of pages randomly accessed, and so on, during the execution of O. The total estimated overhead t_q of a query q is simply the sum of the costs of the individual operators in its query plan. Moreover, as illustrated in [39], the n's are actually functions of the input/output cardinalities (or equivalently, selectivities) of the operators. As a

 Articles from this volume were invited to present their results at the 40th International Conference on Very Large Data Bases, September 1st - 5th 2014, Hangzhou, China.
Proceedings of the VLDB Endowment, Vol. 7, No. 14

c	Description
c_s	The I/O cost to *sequentially* access a page
c_r	The I/O cost to *randomly* access a page
c_t	The CPU cost to process a *tuple*
c_i	The CPU cost to process a tuple via *index* access
c_o	The CPU cost to perform an *operation* (e.g., hash)

Table 1: Cost units in PostgreSQL's cost model

result, we can further represent t_q as a function of the cost units $\mathbf{c}$ and the selectivities $\mathbf{X}$, namely,

$$t_q = \sum_{O \in Plan(q)} t_O = g(\mathbf{c}, \mathbf{X}). \quad (2)$$

Perfect predictions therefore rely on three assumptions: (i) the c's are accurate; (ii) the X's are accurate; and (iii) g is itself accurate. Unfortunately, none of these holds in practice. First, the c's are inherently random. For example, the value of c_r may vary for different disk pages accessed by a query, depending on where the pages are located on disk. Second, accurate selectivity estimation is often challenging, though significant progress has been made. Third, the equations and functions modeling query execution make approximations and simplifications so they could make errors. For instance, Equation (1) does not consider the possible interleaving of CPU and I/O operations during runtime.

To quantify the uncertainty in the prediction, we therefore need to consider potential errors in all three parts of the running time estimation formula. It turns out that the errors in the c's, the X's, and g are inherently different. The errors in the c's result from fluctuations in the system state and/or variances in the way the system performs for different parts of different queries. (That is, for example, the cost of a random I/O may differ substantially from operator to operator and from query to query.) We therefore model the c's as random variables and extend our previous calibration framework [39] to obtain their distributions. The errors in the X's arise from selectivity estimation errors. We therefore also model these as random variables and consider sampling-based approaches to estimate their variance. The errors in g, however, result from simplifications or errors made by the designer of the cost model and are out of the scope of this work. We show in our experiments that even imperfect cost model functions are useful for estimating uncertainty in predictions.

Based on the idea of treating the c's and the X's as random variables rather than constants, the predicted execution time t_q is then also a random variable so that we can estimate its distribution. A couple of challenges arise immediately. First, unlike the case of providing a point estimate of t_q, knowing that t_q is "some" function of the c's and the X's is insufficient if we want to infer the distribution of t_q — we need to know the *explicit* form of g. By Equation (2), g relies on cost functions that map the X's to the n's. As a result, for concreteness we have to choose some specific cost model. Here, for simplicity and generality, we leverage the notion of *logical* cost functions [15] rather than the cost functions of any specific optimizer. The observation is that the costs of an operator can be specified according to its logical execution. For instance, the number of CPU operations of the in-memory *sort* operator could be specified as $n_o = aN \log N$, where N is the input cardinality. Second, while we can show that the distribution of t_q is asymptotically normal based on our current ways of modeling the c's and the X's, determining the parameters of the normal distribution (i.e., the mean and variance) is difficult for non-trivial queries with deep query trees. The challenge arises from correlations between selectivity estimates derived by using shared samples. We present a detailed analysis of the correlations and develop techniques to either directly compute or provide upper bounds for the covariances with respect to the presence of correlations. Finally, providing estimates to distributions of likely running times is desirable only if it can be achieved with low overhead. We show that it is the case for our proposed techniques — the overhead is almost the same as that of the predictor in [39] which only provides point estimates.

Since our approach makes a number of approximations when computing the distribution of running time estimates, an important question is how accurate the estimated distribution is. An intuitively appealing experiment is the following: run the query multiple times, measure the distribution of its running times, and see if this matches the estimated distribution. But this is not a reasonable approach due to the subtlety we mentioned earlier. The estimated distribution we calculate is not the expected distribution of the actual query running time, it is the distribution of running times our estimator expects due to uncertainties in its estimation process. To see this another way, note that cardinality estimation error is a major source of running time estimation error. But when the query is actually run, it does not appear at all — the query execution of course observes the true cardinalities, which are identical every time it is run.

Speaking informally, what our predicted running time distribution captures is the "self-awareness" of our estimator. Suppose that embedded in the estimate is a dependence on what our estimator knows is a very inaccurate estimate. Then the estimator knows that while it gives a specific point estimate for the running time (the mean of a distribution), it is likely that the true running time will be far away from the estimate, and it captures this by indicating a distribution with a large variance.

So our task in evaluating our approach is to answer the following question: how closely does the variance of our estimated distribution of running times correspond to the observed errors in our estimates (when compared with true running times)? To answer this question, we estimate the running times for and run a large number of different queries and test the agreement between the observed errors and the predicted distribution of running times, where "agreement" means that larger variations correspond to more inaccurate estimates.

In more detail, we report two metrics over a large number of queries: (M1) the correlation between the standard deviations of the estimated distributions and the actual prediction errors; and (M2) the proximity between the inferred and observed distributions of prediction errors. We show that (R1) the correlation is strong; and (R2) the two distributions are close. Intuitively, (R1) is *qualitative*; it suggests that one can judge if the prediction errors will be small or large based on the standard deviations of the estimated distributions. (R2) is more *quantitative*; it further suggests that the likelihoods of prediction errors are specified by the distributions as well. We therefore conclude that the estimated distributions do a reasonable job as indicators of prediction errors.

We start by presenting terminology and notation used throughout the paper in Section 2. We then present the details of how to estimate the distributions of the c's and the X's (Section 3), the explicit form of g (Section 4), and the distribution of t_q (Section 5). We further present experimental evaluation results in Section 6, discuss related work in Section 7, and conclude the paper in Section 8.

2. PRELIMINARIES

In most current DBMS implementations, the operators are either unary or binary. Therefore, we can model a query plan with a rooted *binary tree*. Consider an operator O in the query plan. We use O_l and O_r to represent its *left* and *right* child operator, and use

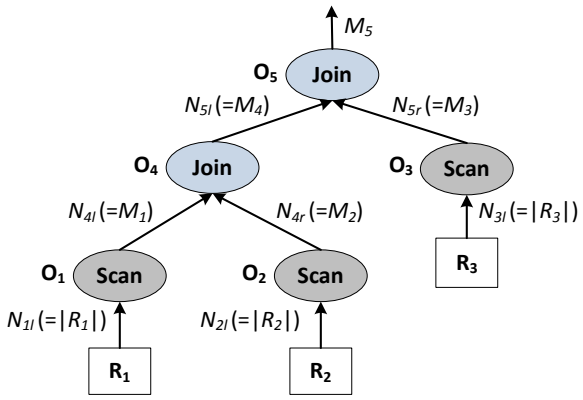


Figure 1: Example query plan

Notation	Description
O	An operator in the query plan
O_l (O_r)	The left (right) child operator of O
N_l (N_r)	The left (right) input cardinality of O
M	The output cardinality of O
$\mathcal{R}$	The leaf tables of O
X	The selectivity of O
$\mathcal{T}$	The subtree rooted at O
$Desc(O)$	The descendant operators of O in $\mathcal{T}$

Table 2: Terminology and notation

N_l and N_r to denote its left and right input cardinality. If O is unary, then O_r does not exist and thus $N_r = 0$. We further use M to denote O's output cardinality.

Let $\mathcal{T}$ be the subtree rooted at the operator O, and let $\mathcal{R}$ be the (multi)set of relations accessed by the leaf nodes of $\mathcal{T}$. Note that the leaf nodes in a query plan must be *scan* operators that access the underlying tables.[1] We call $\mathcal{R}$ the *leaf tables* of O. Let $|\mathcal{R}| = \prod_{R\in\mathcal{R}} |R|$. We define the *selectivity* X of O to be:

$$X = \frac{M}{|\mathcal{R}|} = \frac{M}{\prod_{R\in\mathcal{R}} |R|}. \tag{3}$$

EXAMPLE 2 (SELECTIVITY). *Consider the query plan in Figure 1. O_1, O_2, and O_3 are scan operators that access three underlying tables R_1, R_2, and R_3, and O_4 and O_5 are join operators. The selectivity of O_1, for instance, is $X_1 = \frac{M_1}{|R_1|}$, whereas the selectivity of O_4 is $X_4 = \frac{M_4}{|R_1|\cdot|R_2|}$.*

We summarize the above notation in Table 2 for convenience of reference. Since the n's in Equation (1) are functions of input/output cardinalities of the operators (we discuss different types of cost functions in Section 4.1), it is clear that the n's are also functions of the selectivities (i.e., the X's) defined here. Based on Equation (2), t_q is therefore a function of the c's and the X's. We next discuss how to measure the uncertainties in these parameters.

3. INPUT DISTRIBUTIONS

To learn the distribution of t_q, we first need to know the distributions of the c's and the X's. We do this by extending the framework in our previous work [39].

[1]We use "*relation*" and "*table*" interchangeably in this paper since our discussion does not depend on the *set/bag* semantics.

3.1 Distributions of the c's

In [39], we designed dedicated calibration queries for each c. Consider the following example:

EXAMPLE 3 (CALIBRATION QUERY). *Suppose that we want to know the value of c_t, namely, the CPU cost of processing one tuple. We can use the calibration query* `SELECT * FROM R`, *where* `R` *is some table whose size is known and is loaded into memory. Since this query only involves c_t, its execution time τ can be expressed as $\tau = |\mathtt{R}| \cdot c_t$. We can then run the query, record τ, and compute c_t from this equation.*

Note that we can use different `R`'s here, and different `R`'s may give us different c_t's. We can think of these observed values as i.i.d. samples from the distribution of c_t, and in [39] we used the sample mean as our estimate of c_t. To quantify the uncertainty in c_t, it would make more sense to treat c_t as a random variable rather than a constant. We assume that the distribution of c_t is normal (i.e., Gaussian), for intuitively the CPU speed is likely to be stable and centered around its mean value. Now let $c_t \sim \mathcal{N}(\mu_t, \sigma_t^2)$. It is then a common practice to use the mean and variance of the observed c_t's as estimates for μ_t and σ_t^2.

In general, we can apply similar arguments to all the five cost units. Due to space limitations, readers are referred to [39] for more details on the calibration procedure. In [39] we only calculated the mean, not the variance, but the extension to the variance is straightforward.

3.2 Distributions of the X's

The uncertainties in the X's are quite different from those in the c's. The uncertainties in the c's are due to unavoidable fluctuations in hardware execution speeds. In other words, the c's are inherently random. However, the X's are actually fixed numbers — if we run the query we should always obtain the same ground truths for the X's. The uncertainties in the X's really come from the fact that so far we do not have a perfect selectivity estimator. How to quantify the uncertainties in the X's therefore depends on the nature of the selectivity estimator used. Here we extend the sampling-based approach used in [39], which was first proposed by Haas et al. [21]. It provides a mathematically rigorous way to quantify potential errors in selectivity estimates. It remains interesting future work to investigate the possibility of extending other alternative estimators such as those based on histograms.

3.2.1 A Sampling-Based Selectivity Estimator

Suppose that we have a database consisting of K relations R_1, ..., R_K, where R_k is partitioned into m_k blocks each with size N_k, namely, $|R_k| = m_k N_k$. Without loss of generality, let q be a selection-join query over R_1, ..., R_K, and let $B(k, j)$ be the j-th block of relation k ($1 \le j \le m_k$, and $1 \le k \le K$). Define

$$\mathbf{B}(L_{1,i_1}, ..., L_{K,i_K}) = B(1, L_{1,i_1}) \times \cdots \times B(K, L_{K,i_K}),$$

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. After n steps, we can obtain n^K such samples (notice that these samples are not independent), and the estimator is defined as

$$\rho_n = \frac{1}{n^K} \sum_{i_1=1}^{n} \cdots \sum_{i_K=1}^{n} \rho_{\mathbf{B}(L_{1,i_1},\cdots,L_{K,i_K})}. \tag{4}$$

Here ρ_n is the estimated selectivity of q (after n sampling steps), and $\rho_{\mathbf{B}}$ is the observed selectivity of q over the sample $\mathbf{B}$. This

estimator is shown to be both unbiased and strongly consistent for the actual selectivity ρ of q [21, 39].[2]

By applying the Central Limit Theorem, we can show that

$$\frac{\sqrt{n}}{\sigma}(\rho_n - \rho) \xrightarrow{d} N(0,1).$$

That is, the distribution of ρ_n is approximately normal after a large number of sampling steps [21]: $\rho_n \sim \mathcal{N}(\rho, \sigma_n^2)$, where $\sigma_n^2 = \sigma^2/n$ and $\sigma^2 = \lim_{n\to\infty} n \operatorname{Var}[\rho_n]$.

However, here σ_n^2 is unknown since σ^2 is unknown. In [21], the authors further proposed the following estimator for σ^2:

$$S_n^2 = \sum_{k=1}^{K}\left(\frac{1}{n-1}\sum_{j=1}^{n}(Q_{k,j,n}/n^{K-1} - \rho_n)^2\right), \qquad (5)$$

for $n \geq 2$ (we set $S_1^2 = 0$). Here

$$Q_{k,j,n} = \sum_{(i_1,\ldots,i_K)\in\Omega_k^{(n)}(j)} \rho_{\mathbf{B}(L_{1,i_1},\ldots,L_{K,i_K})}, \qquad (6)$$

where $\Omega_k^{(n)}(j) = \{(i_1,\ldots,i_K) \in \{1,\ldots,n\}^K : i_k = j\}$. It can be shown that $\lim_{n\to\infty} S_n^2 = \sigma^2$ a.s. As a result, it is reasonable to approximate σ^2 with S_n^2 when n is large. So $\sigma_n^2 \approx S_n^2/n$.

3.2.2 *Efficient Computation of S_n^2*

Efficiency is crucial for a predictor to be practically useful. We have discussed efficient implementation of ρ_n in [39]. Taking samples at runtime might not be acceptable since it will result in too many random disk I/O's. Therefore, we instead take samples offline and store them as materialized views (i.e., sample tables). In the following presentation, we use R^s to denote the sample table of a relation R. In [39], we further showed that, given a selection-join query, we can estimate the selectivities of all the selections and joins by running the original query plan over the sample tables once. The trick is that, since the block size is not specified when partitioning the relations, it could be arbitrary. We can then let a block be a single tuple so that the cross-product of sample blocks is reduced to the cross-product of sample tuples.

EXAMPLE 4 (IMPLEMENTATION OF ρ_n). *Let us consider the query plan in Figure 1 again. Based on the tuple-level partitioning scheme, by Equation (4) we can simply estimate X_4 and X_5 as*

$$\widehat{X}_4 = \frac{|R_1^s \bowtie R_2^s|}{|R_1^s| \cdot |R_2^s|} \quad \textit{and} \quad \widehat{X}_5 = \frac{|R_1^s \bowtie R_2^s \bowtie R_3^s|}{|R_1^s| \cdot |R_2^s| \cdot |R_3^s|}.$$

Also note that we can compute the two numerators by running the query plan over the sample relations R_1^s, R_2^s, and R_3^s once. That is, to compute $R_1^s \bowtie R_2^s \bowtie R_3^s$, we reuse the join results from $R_1^s \bowtie R_2^s$ that has been computed when estimating X_4.

We now extend the above framework to further compute S_n^2. For this sake we need to know how to compute the $Q_{k,j,n}$'s in Equation (5). Let us consider the cases when an operator represents a selection (i.e., a scan), a two-way join, or a multi-way join query.

[2]Strong consistency is also called *almost sure convergence* in probability theory (denoted as "a.s."). It means that the more samples we take, the closer ρ_n is to ρ.

Selection. In this case, $K = 1$ and by Equation (6) $Q_{k,j,n}$ is reduced to $Q_{1,j,n} = \rho_B(L_{1,j})$. Therefore, S_n^2 can be simplified as

$$S_n^2 = \frac{1}{n-1}\sum_{j=1}^{n}(\rho_B(L_{1,j}) - \rho_n)^2.$$

Since a block here is just a tuple, $\rho_B(L_{1,j}) = 0$ or $\rho_B(L_{1,j}) = 1$. We thus have

$$\begin{aligned} S_n^2 &= \frac{1}{n-1}\Big(\sum_{\rho_B(L_{1,j})=0}\rho_n^2 + \sum_{\rho_B(L_{1,j})=1}(1-\rho_n)^2\Big) \\ &= \frac{1}{n-1}\big((n-M)\rho_n^2 + M(1-\rho_n)^2\big), \end{aligned}$$

where M is the number of output tuples from the selection. When n is large, $n \approx n-1$, so we have

$$S_n^2 \approx (1-\frac{M}{n})\rho_n^2 + \frac{M}{n}(1-\rho_n)^2 = \rho_n(1-\rho_n),$$

by noticing that $\rho_n = \frac{M}{n}$. Hence S_n^2 is directly computable for a scan operator once we know its estimated selectivity ρ_n.

Two-way Join. Consider a join $R_1 \bowtie R_2$. In this case, $Q_{k,j,n}$ ($k = 1, 2$) can be reduced to

$$Q_{1,j,n} = \sum_{i_2=1}^{n}\rho_{\mathbf{B}}(L_{1,j}, L_{2,i_2}) \text{ and } Q_{2,j,n} = \sum_{i_1=1}^{n}\rho_{\mathbf{B}}(L_{1,i_1}, L_{2,j}).$$

Again, since a block here is just a tuple, $\rho_{\mathbf{B}}$ is either 0 or 1. It is then equivalent to computing the following two quantities:

- $Q_{1,j,n} = |\{t_{1j}\} \bowtie R_2^s|$, where t_{1j} is the jth tuple of R_1^s;
- $Q_{2,j,n} = |R_1^s \bowtie \{t_{2j}\}|$, where t_{2j} is the jth tuple of R_2^s.

That is, to compute $Q_{k,j,n}$ ($k = 1, 2$), conceptually we need to join each sample tuple of one relation with all the sample tuples of the other relation. However, directly performing this is quite expensive, for we need to do $2n$ joins here.

We seek a more efficient solution. Recall that we need to join R_1^s and R_2^s to compute ρ_n. Let $R^s = R_1^s \bowtie R_2^s$. Consider any $t \in R^s$. t must satisfy $t = t_{1i} \bowtie t_{2j}$, where $t_{1i} \in R_1^s$ and $t_{2j} \in R_2^s$. Then t contributes 1 to $Q_{1,i,n}$ and 1 to $Q_{2,j,n}$. On the other hand, any t in $R_1^s \times R_2^s$ but not in R^s will contribute nothing to the Q's. Based on this observation, we only need to scan the tuples in R^s and increment the corresponding Q's. The remaining problem is how to know the indexes i and j as in $t = t_{1i} \bowtie t_{2j}$. For this purpose, we assign an *identifier* to each tuple in the sample tables when taking the samples. This is akin to the idea in data provenance research where tuples are annotated to help tracking the lineages of the query results [19].

Multi-way Joins. The approach of processing two-way joins can be easily generalized to handle multi-way joins. Now we have

$$Q_{k,j,n} = |R_1^s \bowtie \cdots \bowtie \{t_{kj}\} \bowtie \cdots \bowtie R_K^s|.$$

As a result, if we let $R^s = R_1^s \bowtie \cdots \bowtie R_K^s$, then any $t \in R^s$ satisfies $t = t_{1i_1} \bowtie \cdots \bowtie t_{Ki_K}$. $t \in R_1^s \times \cdots \times R_K^s$ will contribute 1 to each $Q_{k,i_k,n}$ ($1 \leq k \leq K$) if and only if $t \in R^s$. Therefore, as before, we can just simply scan R^s and increment the corresponding Q's when processing each tuple.

Putting It Together. Algorithm 1 summarizes the procedure of computing ρ_n and S_n^2 for a single operator O. It is straightforward to incorporate it into the previous framework where the selectivities of the operators are refined in a bottom-up fashion (see [39] and [1]). We discuss some implementation details in the following.

Algorithm 1: Computation of ρ_n and S_n^2

Input: O, an operator; $\mathcal{R}^s = \{R_1^s, ..., R_K^s\}$, the sample tables; Agg, if some $O' \in Desc(O)$ is an aggregate
Output: ρ_n, estimated selectivity of O; S_n^2, sample variance

1 $R^s \leftarrow RunOperator(O, \mathcal{R}^s)$;
2 **if** Agg **then**
3 $\quad M \leftarrow CardinalityByOptimizer(O)$;
4 $\quad \rho_n \leftarrow \frac{M}{\prod_{k=1}^{K} |R_k|}$;
5 $\quad S_n^2 \leftarrow 0$;
6 **else if** O *is a scan* **then**
7 $\quad \rho_n \leftarrow \frac{|R^s|}{|R_1^s|}$;
8 $\quad S_n^2 \leftarrow \rho_n(1-\rho_n)$;
9 **else if** O *is a join* **then**
10 $\quad \rho_n \leftarrow \frac{|R^s|}{\prod_{k=1}^{K} |R_k^s|}$;
11 $\quad$ **foreach** $t = t_{1i_1} \bowtie \cdots \bowtie t_{Ki_K} \in R^s$ **do**
12 $\quad\quad Q_{k,i_k,n} \leftarrow Q_{k,i_k,n} + 1$, for $1 \le k \le K$;
13 $\quad$ **end**
14 $\quad S_n^2 \leftarrow \sum_{k=1}^{K} \left(\frac{1}{n-1} \sum_{j=1}^{n} (Q_{k,j,n}/n^{K-1} - \rho_n)^2 \right)$;
15 **else**
16 $\quad \rho_n \leftarrow \hat{\mu}_l$, $S_n^2 \leftarrow \hat{\sigma}_l^2$; // Let $X_l \sim \mathcal{N}(\hat{\mu}_l, \hat{\sigma}_l^2)$.
17 **end**
18 **return** ρ_n and S_n^2;

First, the selectivity estimator cannot work for operators such as *aggregates*. Our current strategy is to use the original cardinality estimates from the optimizer to compute ρ_n, and we simply set S_n^2 to be 0 for these operators (lines 3 to 5). This may cause inaccuracy in the prediction as well as our estimate of its uncertainty, if the optimizer does a poor job in estimating the cardinalities. However, we find that it works reasonably well in our experiments. Nonetheless, we are working to incorporate sampling-based estimators for aggregates (e.g., the GEE estimator [8]) into our current framework.

Second, to compute the $Q_{k,i_k,n}$'s, we maintain a hash map H_k for each k with i_k's the keys and $Q_{k,i_k,n}$'s the values. The size of H_k is upper bounded by $|R_k^s|$ and usually is much smaller.

Third, for simplicity of exposition, in Algorithm 1 we first compute the whole R^s and then scan it. In practice we actually do not need to do this. Typical join operators, such as *merge join*, *hash join*, and *nested-loop join*, usually compute join results on the fly. Once a join tuple is computed, we can immediately postprocess it by increasing the corresponding $Q_{k,i_k,n}$'s. Therefore, we can avoid the additional memory overhead of caching intermediate join results, which might be large even if the sample tables are small.

4. COST FUNCTIONS

By Equation (2), to infer the distribution of t_q for a query q, we also need to know the explicit form of g. According to Equation (1), g relies on the cost functions of operators that map the selectivities to the n's. As mentioned in the introduction, we use logical cost functions in our work. While different DBMS may differ in their implementations of a particular operator, e.g., nested-loop join, they follow the same execution logic and therefore have the same logical cost function. In the following, we first present a detailed study of representative cost functions. We then formulate the computation of cost functions as an optimization problem that seeks the best fit for the unknown coefficients, and we use standard quadratic programming techniques to solve this problem.

4.1 Types of Functions

We consider the following types of cost functions in this paper:

(C1) $f = a_0$: The cost function is a constant. For instance, since a sequential scan has no random disk reads, $n_r = 0$.

(C2) $f = a_0 M + a_1$: The cost function is linear with respect to the *output* cardinality. For example, the number of random reads of an index-based table scan falls into this category, which is proportional to the number of qualified tuples that pass the selection predicate.

(C3) $f = a_0 N_l + a_1$: The cost function is linear with respect to the *input* cardinality. This happens for unary operators that process each input tuple once. For example, *materialization* is such an operator that creates a buffer to cache the intermediate results.

(C4) $f = a_0 N_l^2 + a_1 N_l + a_2$: The cost function is nonlinear with respect to the *input* cardinality. For instance, the number of CPU operations (i.e., c_o) performed by a *sort* operator is proportional to $N_l \log N_l$. While different nonlinear unary operators may have specific cost functions, we choose to only use quadratic polynomials based on the following observations:

- It is quite general to approximate the nonlinear cost functions used by current relational operators. First, as long as a function is smooth (i.e., it has continuous derivatives up to some desired order), it can be approximated by using the well-known Taylor series, which is basically a polynomial of the input variable. Second, for efficiency reasons, the overhead of an operator usually does not go beyond quadratic of its input cardinality — we are not aware of any operator implementation whose time complexity is $\omega(N^2)$. Similar observations have been made in [13].
- Compared with functions such as logarithmic ones, polynomials are mathematically much easier to manipulate. Since we need to further infer the distribution of the predicted query execution time based on the cost functions, this greatly simplifies the derivations.

(C5) $f = a_0 N_l + a_1 N_r + a_2$: This cost function is linear with respect to the *input* cardinalities when the operator is binary. An interesting observation here is that the cost functions in the case of binary operators are not necessarily nonlinear. For example, the number of I/O's involved in a hash join is only proportional to the number of input tuples.

(C6) $f = a_0 N_l N_r + a_1 N_l + a_2 N_r + a_3$: The cost function here also involves the product of the left and right input cardinalities of a binary operator. This happens typically in a nested-loop join, which iterates over the inner (i.e., the right) input table multiple times with respect to the number of rows in the outer (i.e., the left) input table.

It is straightforward to translate these cost functions in terms of selectivities. Specifically, we have $N_l = |\mathcal{R}_l| X_l$, $N_r = |\mathcal{R}_r| X_r$, and $M = |\mathcal{R}| X$. The above six cost functions can be rewritten as

(C1') $f = b_0$, where $b_0 = a_0$.

(C2') $f = b_0 X + b_1$, where $b_0 = a_0|\mathcal{R}|$ and $b_1 = a_1$.

(C3') $f = b_0 X_l + b_1$, where $b_0 = a_0|\mathcal{R}_l|$ and $b_1 = a_1$.

(C4') $f = b_0 X_l^2 + b_1 X_l + b_2$, where $b_0 = a_0|\mathcal{R}_l|^2$, $b_1 = a_1|\mathcal{R}_l|$, and $b_2 = a_2$.

(C5') $f = b_0 X_l + b_1 X_r + b_2$, where $b_0 = a_0|\mathcal{R}_l|$, $b_1 = a_1|\mathcal{R}_r|$, and $b_2 = a_2$.

(C6') $f = b_0 X_l X_r + b_1 X_l + b_2 X_r + b_3$, where $b_0 = a_0|\mathcal{R}_l| \cdot |\mathcal{R}_r|$, $b_1 = a_1|\mathcal{R}_l|$, $b_2 = a_2|\mathcal{R}_r|$, and $b_3 = a_3$.

4.2 Computation of Cost Functions

To compute the cost functions, we use an approach that is similar to the one proposed in [13]. Regarding the types of cost functions we considered, the only unknowns given the selectivity estimates are the coefficients in the functions (i.e., the b's). Moreover, notice that f is a *linear* function of the b's once the selectivities are given. We can then collect a number of f values by feeding in the cost model with different X's and find the best fit for the b's.

As an example, consider (C4'). Suppose that we invoke the cost model m times and obtain m points:

$$\{(X_{l1}, f_1), ..., (X_{lm}, f_m)\}.$$

Let $\mathbf{y} = (f_1, ..., f_m)$, $\mathbf{b} = (b_0, b_1, b_2)$, and

$$\mathbf{A} = \begin{pmatrix} X_{l1}^2 & X_{l1} & 1 \\ \vdots & \vdots & \vdots \\ X_{lm}^2 & X_{lm} & 1 \end{pmatrix}.$$

The optimization problem we are concerned with is:

$$\begin{aligned} \underset{\mathbf{b}}{\text{minimize}} \quad & \|\mathbf{Ab} - \mathbf{y}\| \\ \text{subject to} \quad & b_i \geq 0,\ i = 0, 1. \end{aligned}$$

Note that we require b_0 and b_1 be nonnegative since they have the natural semantics in the cost functions as the amount of work with respect to the corresponding terms. For example, $b_1 X_l = a_1 N_l$ is the amount of work that is proportional to the input cardinality. To solve this quadratic programming problem, we use the `qpsolve` function of Scilab [34]. Other equivalent solvers could also be used.

The remaining problem is how to pick these (X, f)'s. In theory, one could arbitrarily pick the X's from $[0, 1]$ to obtain the corresponding f's as long as we have more points than unknowns. Although more points usually mean we can have better fittings, in practice we cannot afford too many points due to the efficiency requirements when making the prediction. On the other hand, given that the X's here follow normal distributions and the variances are usually small when the sample size is large, the likely selectivity estimates are usually concentrated in a much shorter interval than $[0, 1]$. Intuitively, we should take more points within this interval, for we can then have a more accurate view of the shape of the cost function restricted to this interval. Therefore, in our current implementation, we adopt the following strategy.

Let $X \sim \mathcal{N}(\mu, \sigma^2)$. Consider the interval $\mathcal{I} = [\mu - 3\sigma, \mu + 3\sigma]$. It is well known that $\Pr(X \in \mathcal{I}) \approx 0.997$, which means the probability that X falls out of $\mathcal{I}$ is less than 0.3%. We then proceed by partitioning $\mathcal{I}$ into W subintervals of equal width, and pick the $W + 1$ boundary X's to invoke the cost model. Generalizing this idea to binary cost functions is straightforward. Suppose $X_l \sim \mathcal{N}(\mu_l, \sigma_l^2)$ and $X_r \sim \mathcal{N}(\mu_r, \sigma_r^2)$. Let $\mathcal{I}_l = [\mu_l - 3\sigma_l, \mu_l + 3\sigma_l]$ and $\mathcal{I}_r = [\mu_r - 3\sigma_r, \mu_r + 3\sigma_r]$. We then partition $\mathcal{I}_l \times \mathcal{I}_r$ into a $W \times W$ grid and obtain $(W + 1) \times (W + 1)$ points (X_l, X_r) to invoke the cost model.

5. DISTRIBUTION OF RUNNING TIMES

We have discussed how to estimate the distributions of input parameters (i.e., the c's and the X's) and how to estimate the cost functions of each operator. In this section, we discuss how to combine these two to further infer the distribution of t_q for a query q.

Since $t_q = g(\mathbf{c}, \mathbf{X})$, the distribution of t_q relies on the *joint* distribution of $(\mathbf{c}, \mathbf{X})$.[3] We therefore first present a detailed analysis of the correlations between the c's and the X's. Based on that, we then show that the distribution of t_q is asymptotically normal and thus reduce the problem to estimating the two parameters of normal distributions, i.e., the mean and variance of t_q. We further address the nontrivial problem of computing $\mathrm{Var}[t_q]$ due to correlations between selectivity estimates.

5.1 Correlations of Input Variables

In our current setting, it is reasonable to assume that the c's and the X's are independent. In the following, we analyze the correlations within the c's and the X's.

5.1.1 Correlations Between Cost Units

Since the randomness within the c's comes from the variations in hardware execution speeds, we have no way to observe the true values of the c's and thus it is impossible to obtain the exact joint distribution of the c's. Nonetheless, it might be reasonable to assume the independence of the c's. First, since the CPU and I/O cost units measure the speeds of different hardware devices, their values do not depend on each other. Second, within each group (i.e., CPU or I/O cost units), we used independent calibration queries for each individual cost unit.

ASSUMPTION 1. *The c's are independent of each other.*

5.1.2 Correlations Between Selectivity Estimates

The X's are clearly not independent, because the same samples are used to estimate the selectivities of different operators. We next study the correlations between the X's in detail.

Let O and O' be two operators, and $\mathcal{R}$ and $\mathcal{R}'$ be the corresponding leaf tables. Consider the two corresponding selectivity estimates ρ_n and ρ'_n as defined by Equation (4). Since the samples from each table are drawn independently, we first have:

LEMMA 1. *If $\mathcal{R} \cap \mathcal{R}' = \emptyset$, then $\rho_n \perp \rho'_n$.*[4]

For binary operators, it follows from Lemma 1 immediately that:

LEMMA 2. *Let O be binary. If $\mathcal{R}_l \cap \mathcal{R}_r = \emptyset$, then $X_l \perp X_r$.*

That is, X_l and X_r will only be correlated if $\mathcal{R}_l$ and $\mathcal{R}_r$ share *common* relations. However, in practice, we can maintain more than one sample table for each relation. When the database is large, this is affordable since the number of samples is very small compared to the database size [39]. Since the samples from each relation are drawn independently, X_l and X_r are still independent if we use a different sample table for each appearance of a shared relation. We thus assume $X_l \perp X_r$ in the rest of the paper.

[3]Note that the distributions of the c's and X's that we obtained in Section 3 are *marginal* rather than joint.

[4]We use $Y \perp Z$ to denote that Y and Z are independent.

More generally, X and X' are independent as long as neither $O \in Desc(O')$ nor $O' \in Desc(O)$. However, the above discussion cannot be applied if $O \in Desc(O')$ (or vice versa). This is because we pass the join results from downstream joins to upstream joins when estimating the selectivities (recall Example 4). So $\mathcal{R}$ and $\mathcal{R}'$ are naturally not disjoint. In fact, $\mathcal{R} \subseteq \mathcal{R}'$. To make ρ_n and ρ'_n independent, we need to replace each of the sample tables used in computing ρ'_n with another sample table from the same relation, which basically is the same as run the query plan again on a different set of sample tables. The number of runs is then in proportion to the number of selective operators (i.e., selections and joins) in the query plan, and the runtime overhead might be prohibitive in practice. We summarize this observation as follows:

LEMMA 3. *Given that multiple sample tables of the same relation can be used, ρ_n and ρ'_n are correlated if and only if either $O \in Desc(O')$ or vice versa.*

5.2 Asymptotic Distributions

Now for specificity suppose that the query plan of q contains m operators O_1, ..., O_m. Since t_q is the sum of the predicted execution time spent on each operator, it can be expressed as $t_q = \sum_{k=1}^{m} t_k$, where t_k is the predicted execution time of O_k and is itself a random variable.

We next show that t_k is asymptotically normal, and then by using very similar arguments, we can show that t_q is asymptotically normal as well. Since t_k can be further expressed in terms of Equation (1), to learn its distribution we need to know the distributions of cost functions that map the selectivities to the n's. We therefore start by discussing the distributions of the typical cost functions as presented in Section 4.1.

5.2.1 Asymptotic Distributions of Cost Functions

In the following discussion, we assume that $X \sim \mathcal{N}(\mu, \sigma^2)$, $X_l \sim \mathcal{N}(\mu_l, \sigma_l^2)$, and $X_r \sim \mathcal{N}(\mu_r, \sigma_r^2)$. The distributions of the six types of cost functions previously discussed are as follows:

(C1') $f = b_0$: $f \sim \mathcal{N}(b_0, 0)$.

(C2') $f = b_0 X + b_1$: $f \sim \mathcal{N}(b_0\mu + b_1, b_0^2\sigma^2)$.

(C3') $f = b_0 X_l + b_1$: $f \sim \mathcal{N}(b_0\mu_l + b_1, b_0^2\sigma_l^2)$.

(C4') $f = b_0 X_l^2 + b_1 X_l + b_2$: In this case $\Pr(f)$ is not normal. Although it is possible to derive the *exact* distribution of f based on the distribution of X_l, the derivation would be very messy. Instead, we consider $f^{\mathcal{N}} \sim \mathcal{N}(\mathrm{E}[f], \mathrm{Var}[f])$ and use this to approximate $\Pr(f)$. We present the formula of $\mathrm{Var}[f]$ in Lemma 4. Obviously, $f^{\mathcal{N}}$ and f have the same expected value and variance. Moreover, we can actually show that $f^{\mathcal{N}}$ and f (and therefore their corresponding distributions) are very close to each other when the number of samples is large (see Theorem 1).

(C5') $f = b_0 X_l + b_1 X_r + b_2$: Since $X_l \perp X_r$ by Lemma 2, $f \sim \mathcal{N}(b_0\mu_l + b_1\mu_r + b_2, b_0^2\sigma_l^2 + b_1^2\sigma_r^2)$.

(C6') $f = b_0 X_l X_r + b_1 X_l + b_2 X_r + b_3$: Again, $\Pr(f)$ is not normal. Since $X_l \perp X_r$, $X_l X_r$ follows the so called *normal product distribution* [6], whose exact form is again complicated. We thus use the same strategy as in (C4') (see [1]).

LEMMA 4. *If $X_l \sim \mathcal{N}(\mu_l, \sigma_l^2)$ and $f = b_0 X_l^2 + b_1 X_l + b_2$, then*

$$\mathrm{Var}[f] = \sigma_l^2[(b_1 + 2b_0\mu_l)^2 + 2b_0^2\sigma_l^2].$$

THEOREM 1. *Suppose that $X_l \sim \mathcal{N}(\mu_l, \sigma_l^2)$ and $f = b_0 X_l^2 + b_1 X_l + b_2$. Let $f^{\mathcal{N}} \sim \mathcal{N}(\mathrm{E}[f], \mathrm{Var}[f])$, where $\mathrm{Var}[f]$ is shown in Lemma 4. Then $f^{\mathcal{N}} \xrightarrow{p} f$.*[5]

5.2.2 Asymptotic Distribution of t_k

Based on the previous analysis, the cost functions (or equivalently, the n's in Equation (1)) are asymptotically normal. Since the c's are normal and independent of the X's (and hence the n's as well), by Equation (1) again t_k is asymptotically the sum of products of two independent normal random variables. Specifically, let $\mathcal{C} = \{c_s, c_r, c_t, c_i, c_o\}$, and for $c \in \mathcal{C}$, let f_{kc} be the cost function indexed by c. Defining $t_{kc} = f_{kc}^{\mathcal{N}} c$, we have

$$t_k \approx \sum_{c \in \mathcal{C}} t_{kc} = \sum_{c \in \mathcal{C}} f_{kc}^{\mathcal{N}} c,$$

Again, each t_{kc} is not normal. But we can apply techniques similar to that in Theorem 1 here by using the normal random variable

$$t_{kc}^{\mathcal{N}} \sim \mathcal{N}(\mathrm{E}[f_{kc}^{\mathcal{N}} c], \mathrm{Var}[f_{kc}^{\mathcal{N}} c]) = \mathcal{N}(\mathrm{E}[f_{kc} c], \mathrm{Var}[f_{kc} c])$$

as an approximation of t_{kc}. Defining $Z = \mathrm{E}[f_{kc}]c$, we have

THEOREM 2. *$t_{kc} \xrightarrow{d} Z$, and $t_{kc}^{\mathcal{N}} \xrightarrow{d} Z$.*

Theorem 2 implies that t_{kc} and $t_{kc}^{\mathcal{N}}$ tend to follow the same distribution as the sample size grows. Since c is normal, Z is normal as well. Furthermore, the independence of the c's also implies the independence of the Z's. So t_k is approximately the sum of the independent normal random variables $t_{kc}^{\mathcal{N}}$, which means t_k is itself approximately normal when the sample size is large.

5.2.3 Asymptotic Distribution of t_q

Finally, let us consider the distribution of t_q. Since t_q is merely the sum of the t_k's, we have exactly the same situation as when we analyze each t_k. Specifically, we can express t_q as

$$t_q = \sum_{k=1}^{m} t_k \approx \sum_{c \in \mathcal{C}} g_c c,$$

where $g_c = \sum_{k=1}^{m} f_{kc}^{\mathcal{N}}$ is the sum of the cost functions of the operators with respect to the particular c. However, since the $f_{kc}^{\mathcal{N}}$'s are not independent, g_c is not normal. We can again use the normal random variable

$$g_c^{\mathcal{N}} \sim \mathcal{N}(\mathrm{E}[g_c], \mathrm{Var}[g_c])$$

as an approximation of g_c. We can show that $g_c^{\mathcal{N}} \xrightarrow{p} g_c$ (see [1]). With exactly the same argument used in Section 5.2.2 we can then see that t_q is approximately normal when the sample size is large.

5.2.4 Discussion

The analysis that t_q is asymptotically normal relies on three facts: (1) the selectivity estimates are unbiased and strongly consistent; (2) the cost model is additive; and (3) the cost units are independently normally distributed. While the first fact is a property of the sampling-based selectivity estimator and thus always holds, the latter two are specific merits of the cost model of PostgreSQL, though we believe that cost models of other database systems share more or less similar features. Therefore, we need new techniques when either (2) or (3) does not hold. For instance, if the cost model is still additive and the c's are independent but cannot be modeled as normal variables, then by the analysis in Section 5.2.3 we can still see that t_q is asymptotically a linear combination of the c's and thus

[5] $f^{\mathcal{N}} \xrightarrow{p} f$ means $f^{\mathcal{N}}$ converges in probability to f.

the distribution of t_q can be expressed in terms of the *convolution* of the distributions of the c's. We may then find this distribution by using generating functions or characteristic functions [33]. We leave the investigation of other types of cost models as future work.

5.3 Computing Distribution Parameters

As discussed, we can approximate the distribution of t_q with a normal distribution $\mathcal{N}(\mathrm{E}[t_q], \mathrm{Var}[t_q])$. We are then left with the problem of estimating the two parameters $\mathrm{E}[t_q]$ and $\mathrm{Var}[t_q]$. While $\mathrm{E}[t_q]$ is trivial to compute — it is merely the original prediction from our predictor, estimating $\mathrm{Var}[t_q]$ is a challenging problem due to the correlations presented in selectivity estimates.

In more detail, so far we have observed the additive nature of t_q, that is, $t_q = \sum_{k=1}^{m} t_k$ and $t_k = \sum_{c \in \mathcal{C}} t_{kc}$ (Section 5.2.2). Recall the fact that for sum of random variables $Y = \sum_{1 \le i \le m} Y_i$,

$$\mathrm{Var}[Y] = \sum\nolimits_{1 \le i,j \le m} \mathrm{Cov}(Y_i, Y_j).$$

Applying this to t_q, our task is then to compute each $\mathrm{Cov}(t_i, t_j)$. Note that $\mathrm{Cov}(t_i, t_i) = \mathrm{Var}[t_i]$ which is easy to compute, so it is left to compute $\mathrm{Cov}(t_i, t_j)$ for $i \neq j$. By linearity of covariance,

$$\mathrm{Cov}(t_i, t_j) = \mathrm{Cov}\left(\sum_{c \in \mathcal{C}} t_{ic}, \sum_{c \in \mathcal{C}} t_{jc}\right) = \sum_{c,c' \in \mathcal{C}} \mathrm{Cov}(t_{ic}, t_{jc'}).$$

In the following, we first specify the cases where direct computation of $\mathrm{Cov}(t_{ic}, t_{jc'})$ can be done. We then develop upper bounds for those covariances that cannot be directly computed.

5.3.1 Direct Computation of Covariances

Any $\mathrm{Cov}(t_{ic}, t_{jc'})$ can fall into the following two cases:

- $i = j$, then it is the covariance between different cost functions from the same operator.
- $i \neq j$, then it is the covariance between cost functions from different operators.

Consider the case $i = j$ first. If the operator is unary, regarding the cost functions we are concerned with, we only need to consider $\mathrm{Cov}(X, X)$, $\mathrm{Cov}(X, X^2)$, and $\mathrm{Cov}(X^2, X^2)$, where $X \sim \mathcal{N}(\mu, \sigma^2)$. Since X is normal, the non-central moments of X can be expressed in terms of μ and σ^2. Hence it is straightforward to compute these covariances [37]. If the operator is binary, then we need to consider $\mathrm{Cov}(X_l, X_l)$, $\mathrm{Cov}(X_r, X_r)$, $\mathrm{Cov}(X_l, X_r)$, $\mathrm{Cov}(X_l X_r, X_l)$, $\mathrm{Cov}(X_l X_r, X_r)$, and $\mathrm{Cov}(X_l X_r, X_l X_r)$. By Lemma 2, $X_l \perp X_r$. So we are able to directly compute these covariances as well.

When $i \neq j$, while the types of covariances that we need to consider are similar as before, it is more complicated since the selectivities are no longer independent. Without loss of generality, we consider two operators O and O' such that $O \in Desc(O')$. By Lemma 3, this is the only case where the covariances might not be zero. Based on the cost functions considered in this paper, we need to consider the covariances $\mathrm{Cov}(Z, Z')$, where $Z \in \{X_l, X_l^2, X_r, X_l X_r\}$ and $Z' \in \{X_l', (X_l')^2, X_r', X_l' X_r'\}$. Some of them can be directly computed by applying Lemma 3, while the others can only be bounded as discussed in the next section.

EXAMPLE 5 (COVARIANCES BETWEEN SELECTIVITIES). *To illustrate, consider the two join operators O_4 and O_5 in Figure 1. Assume that the cost functions of O_4 and O_5 are all linear, i.e., they are of type (C5'). Based on Lemma 2,* $\mathrm{Cov}(X_1, X_2) = 0$ *and* $\mathrm{Cov}(X_4, X_3) = 0$. *Also, based on Lemma 3,* $\mathrm{Cov}(X_1, X_3) = 0$ *and* $\mathrm{Cov}(X_2, X_3) = 0$. *However, we are not able to compute* $\mathrm{Cov}(X_1, X_4)$ *and* $\mathrm{Cov}(X_2, X_4)$. *Instead, we provide upper bounds for them.*

5.3.2 Upper Bounds of Covariances

Based on the fact that the covariance between two random variables is bounded by the geometric mean of their variances [33], we can establish an upper bound for Z and Z' in the previous section:

$$|\mathrm{Cov}(Z, Z')| \le \sqrt{\mathrm{Var}[Z]\,\mathrm{Var}[Z']}.$$

Note that the variances are directly computable based on the independence assumptions (Lemma 2 and 3).

By analyzing the correlation of the samples used in selectivity estimation, we can develop tighter bounds. Due to space limitations, we defer the details to the full version of this paper [1]. The key observation here is that the correlations are caused by the samples from the shared relations. Consider two operators O and O' such that $O \in Desc(O')$. Suppose that $|\mathcal{R} \cap \mathcal{R}'| = m$ ($m \ge 1$), namely, O and O' share m common leaf tables. Let the estimators for O and O' be ρ_n and ρ_n', where n is the number of sample steps. We define $S_\rho^2(m, n)$ to be the variance of samples restricted to the m common relations. This is actually a generalization of $\mathrm{Var}[\rho_n]$. To see this, let $\mathcal{R}' = \mathcal{R}$. Then $\rho_n = \rho_n'$ and hence

$$\mathrm{Var}[\rho_n] = \mathrm{Cov}(\rho_n, \rho_n) = \mathrm{Cov}(\rho_n, \rho_n') = S_\rho^2(K, n),$$

where $K = |\mathcal{R}|$. We can show that $S_\rho^2(m, n)$ is a monotonically increasing function of m [1]. As a result, $S_\rho^2(m, n) \le \mathrm{Var}[\rho_n]$ given that $m \le K$. Hence, we have the following refined upper bound for $\mathrm{Cov}(\rho_n, \rho_n')$:

$$|\mathrm{Cov}(\rho_n, \rho_n')| \le \sqrt{S_\rho^2(m, n) S_{\rho'}^2(m, n)} \le \sqrt{\mathrm{Var}[\rho_n]\,\mathrm{Var}[\rho_n']}.$$

To compute $S_\rho^2(m, n)$, we use an estimator akin to the estimator $\sigma_n^2 = S_n^2/n$ that we used to estimate $\mathrm{Var}[\rho_n]$. Specifically, define

$$S_{n,m}^2 = \sum_{r=1}^{m} \left(\frac{1}{n-1} \sum_{j=1}^{n} (Q_{r,j,n}/n^{m-1} - \rho_n)^2 \right),$$

for $n \ge 2$ (we set $S_{1,m}^2 = 0$). Very similarly, we can show that $\lim_{n \to \infty} S_{n,m}^2 = n S_\rho^2(m, n)$. As a result, it is reasonable to approximate $S_\rho^2(m, n)$ with $S_\rho^2(m, n) \approx S_{n,m}^2/n$. Moreover, by comparing the expressions of $S_{n,m}^2$ and S_n^2 (ref. Equation (5)), we can see that $S_n^2 = S_{n,K}^2$. Therefore it is straightforward to adapt the implementation framework in Section 3.2.2 to compute $S_{n,m}^2$.

6. EXPERIMENTAL EVALUATION

We present experimental evaluation results in this section. There are two key respects that could impact the utility of a predictor: its prediction accuracy and runtime overhead. However, for the particular purpose of this paper, we do not care much about the *absolute* accuracy of the prediction. Rather, we care if the distribution of likely running times reflects the uncertainty in the prediction. Specifically, we measure if the estimated prediction errors are correlated with the actual errors. To measure the accuracy of the predicted distribution, we also compare the estimated likelihoods that the actual running times will fall into certain confidence intervals with the actual likelihoods. On the other hand, we measure the runtime overhead of the sampling-based approach in terms of its relative overhead with respect to the original query running time without sampling. We start by presenting the experimental settings and the benchmark queries we used.

6.1 Experimental Settings

We implemented our proposed framework in PostgreSQL 9.0.4. We ran PostgreSQL under Linux 3.2.0-26, and we evaluated our approaches with both the TPC-H 1GB and 10 GB databases. Since

the original TPC-H database generator uses uniform distributions, to test the effectiveness of the approach under different data distributions, we used a skewed TPC-H database generator [2]. It produces TPC-H databases with a Zipf distribution and uses a parameter z to control the degree of skewness. $z = 0$ generates a uniform distribution, and the data becomes more skewed as z increases. We created skewed databases using $z = 1$. All experiments were conducted on two machines with the following configurations:

- PC1: Dual Intel 1.86 GHz CPU and 4GB of memory;
- PC2: 8-core 2.40GHz Intel CPU and 16GB of memory.

6.2 Benchmark Queries

We created three benchmarks **MICRO**, **SELJOIN**, and **TPCH**:

- **MICRO** consists of pure selection queries (i.e., scans) and two-way join queries. It is a micro-benchmark with the purpose of exploring the strength and weakness of our proposed approach at different points in the selectivity space. We generated the queries with the similar ideas used in the Picasso database query optimizer visualizer [32]. Since the queries have either one (for scans) or two predicates (for joins), the selectivity space is either one or two dimensional. We generated SQL queries that were evenly across the selectivity space, by using the statistics information (e.g., histograms) stored in the database catalogs to compute the selectivities.
- **SELJOIN** consists of selection-join queries with multi-way joins. We generated the queries in the following way. We analyzed each TPC-H query template, and identified the "maximal" sub-query without aggregates. We then randomly generated instance queries from these *reduced* templates. The purpose is to test the particular type of queries to which our proposed approach is tailored — the selection-join queries.
- **TPCH** consists of instance queries from the TPC-H templates. These queries also contain aggregates, and our current strategy is simply ignoring the uncertainty there (recall Section 3.2.2). The purpose of this benchmark is to see how this simple work-around works in practice. We used 14 TPC-H templates: 1, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 14, 18, and 19. We did not use the other templates since their query plans contain structures that cannot be handled by our current framework (e.g., sub-query plans or views).

We ran each query 5 times and took the average as the actual running time of a query. We cleared both the filesystem cache and the database buffer pool between each run of each query.

6.3 Usefulness of Predicted Distributions

Since our goal is to quantify the uncertainty in the prediction and our output is a distribution of likely running times, the question is then how we can know that we have something useful. A reasonable metric here could be the correlation between the standard deviation of the predicted (normal) distribution and the actual prediction error. Intuitively, the standard deviation indicates the confidence of the prediction. A larger standard deviation indicates lower confidence and hence larger potential prediction error. With this in mind, if our approach is effective, we would expect to see positive correlations between the standard deviations and the real prediction errors when a large number of queries are tested.

A common metric used to measure the correlation between two random variables is the Pearson correlation coefficient r_p. Suppose that we have n queries q_1, ..., q_n. Let σ_i be the standard deviation of the distribution predicted for q_i, μ_i and t_i be the predicted (mean) and actual running time of q_i, and $e_i = |\mu_i - t_i|$ be the prediction error. r_p is then defined as

$$r_p = \frac{\sum_{i=1}^{n}(\sigma_i - \bar{\sigma})(e_i - \bar{e})}{\sqrt{\sum_{i=1}^{n}(\sigma_i - \bar{\sigma})^2}\sqrt{\sum_{i=1}^{n}(e_i - \bar{e})^2}}, \quad (7)$$

where $\bar{\sigma} = \frac{1}{n}\sum_{i=1}^{n}\sigma_i$ and $\bar{e} = \frac{1}{n}\sum_{i=1}^{n}e_i$.

Basically, r_p measures the *linear* correlation between the σ's and the e's. The closer r_p is to 1, the better the correlation is. However, there are two issues here. First, even if the σ's and the e's are positively correlated, the correlation may not be linear. Second, r_p is not robust and its value can be misleading if outliers are present [14]. Therefore, we also measure the correlations by using another well known metric called the Spearman's rank correlation coefficient r_s [30]. The formula of r_s is the same as Equation (7) except for that the σ's and e's are replaced with their *ranks* in the ascending order of the values. For instance, given three σ's $\sigma_1 = 4$, $\sigma_2 = 7$, and $\sigma_3 = 5$, their ranks are 1, 3, and 2 respectively. Intuitively, r_s indicates the linear correlation between the ranks of the values, which is more robust than r_p since the mapping from the values to their ranks can be thought of as some *normalization* procedure that reduces the impact of outliers. In fact, r_s assesses how well the correlation can be characterized by using a *monotonic* function and $r_s = 1$ means the correlation is perfect.

In Figure 2, we report the r_s's (and the corresponding r_p's) for the benchmark queries over different hardware and database settings. Due to space limitations, we refer the readers to [1] for the complete results. Here, sampling ratio (SR) stands for the fraction of the sample size with respect to the database size. For instance, SR = 0.01 means that 1% of the data is taken as samples. We have several observations.

First, for most of the cases we tested, both r_s and r_p are above 0.7 (in fact above 0.9), which implies strong positive (linear) correlation between the standard deviations of the predicted distributions and the actual prediction errors.[6] Second, in [39] we showed that as expected, prediction errors can be reduced by using larger number of samples. Interestingly, it is not necessarily the case that more samples improves the correlation between the predicted and actual errors. This is because taking more samples simultaneously reduces the errors in selectivity estimates and the uncertainty in the predicted running times. So it might improve the estimate but not the correlation with the true errors. Third, reporting both r_s and r_p is necessary since they sometimes disagree with each other. For instance, consider the following two cases in Figure 2(a) and 2(b):

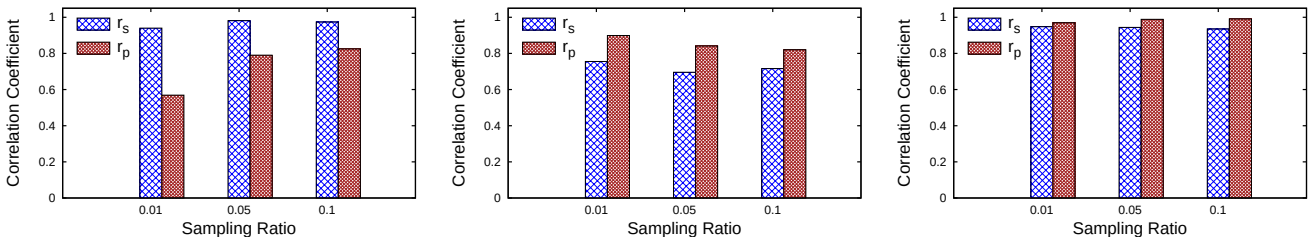
(1) On PC2, the **MICRO** queries over the uniform TPC-H 1GB database give r_s = 0.9400 but r_p = 0.5691 when SR = 0.01;

(2) On PC1, the **SELJOIN** queries over the uniform TPC-H 1GB database give r_s = 0.6958 but r_p = 0.8414 when SR = 0.05.

In Figure 3(a) and 3(c), we present the scatter plots of these two cases. Figure 3(b) further shows the scatter plot after the rightmost point is removed from Figure 3(a). We find that now r_s = 0.9386 but r_p = 0.8868. So r_p is much more sensitive to outliers in the population. Since in our context there is no good criterion to remove outliers, r_s is thus more trustworthy. On the other hand, although the r_p of (2) is better than that of (1), by comparing Figure 3(b) with Figure 3(c) we would instead conclude that the correlation of (2) seems to be worse. This is again implied by the worse r_s of (2).

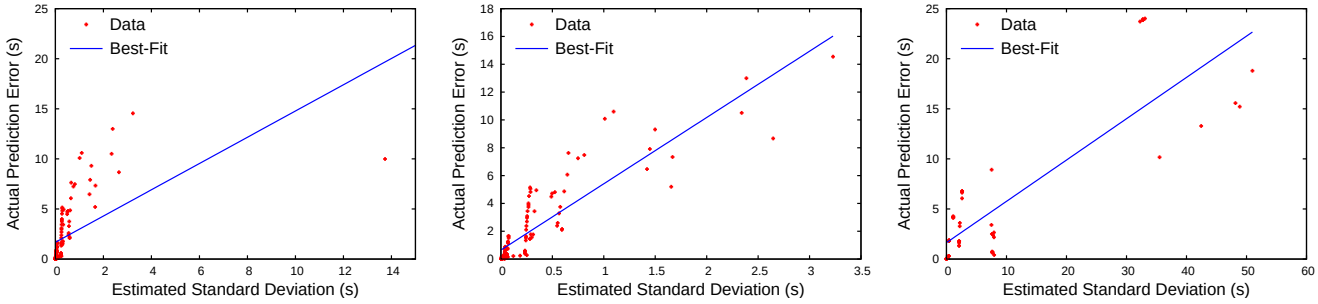
Nonetheless, the strong positive correlations between the estimated standard deviations and the actual prediction errors may not

[6]It is generally believed that two variables are strongly correlated if their correlation coefficient is above 0.7.



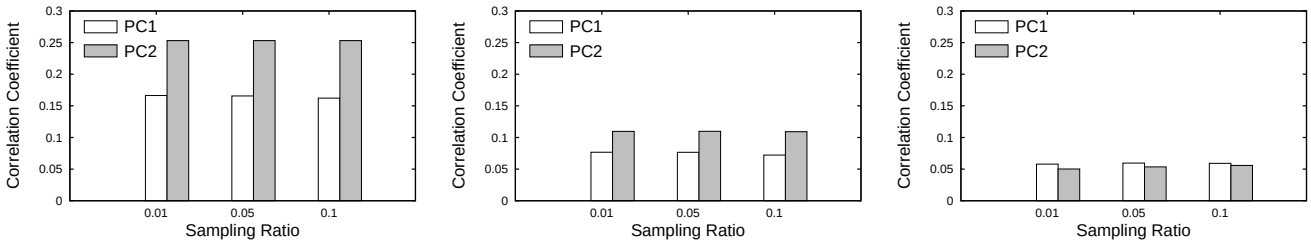
(a) **MICRO**, Uniform 1GB, PC2 (b) **SELJOIN**, Uniform 1GB, PC1 (c) **TPCH**, Skewed 10GB, PC1

Figure 2: r_s and r_p of the benchmark queries over different hardware and database settings



(a) Case (1) (b) Case (1) after one outlier is removed (c) Case (2)

Figure 3: Robustness of r_s and r_p with respect to outliers



(a) **MICRO** (b) **SELJOIN** (c) **TPCH**

Figure 4: $\overline{D}_n$ of the benchmark queries over uniform TPC-H 10GB databases

be sufficient to conclude that the distributions of likely running times are useful. For our purpose of informing the consumer of the running time estimates of the potential prediction errors, it might be worth to further consider what information regarding the errors the predicted distributions really carry. Formally, consider the n queries q_1, ..., q_n as before. Since the estimated distributions are normal, with the previous notation the distribution for the likely running times T_i of q_i is $T_i \sim \mathcal{N}(\mu_i, \sigma_i^2)$. As a result, assuming $\alpha > 0$, without loss of generality the estimated prediction error $E_i = |T_i - \mu_i|$ follows the distribution

$$\Pr(E_i \le \alpha\sigma_i) = \Pr(-\alpha \le \frac{T_i - \mu_i}{\sigma_i} \le \alpha) = 2\Phi(\alpha) - 1,$$

where Φ is the cumulative distribution function of the standard normal distribution $\mathcal{N}(0, 1)$. Therefore, if we define the statistic $E_i' = \frac{E_i}{\sigma_i} = |\frac{T_i - \mu_i}{\sigma_i}|$, then $\Pr(E_i' \le \alpha) = \Pr(E_i \le \alpha\sigma_i)$. Note that $\Pr(E_i' \le \alpha)$ is determined by α but not i. We thus simply use $\Pr(\alpha)$ to denote $\Pr(E_i' \le \alpha)$. On the other hand, we can estimate the actual likelihood of $E_i' \le \alpha$ by using

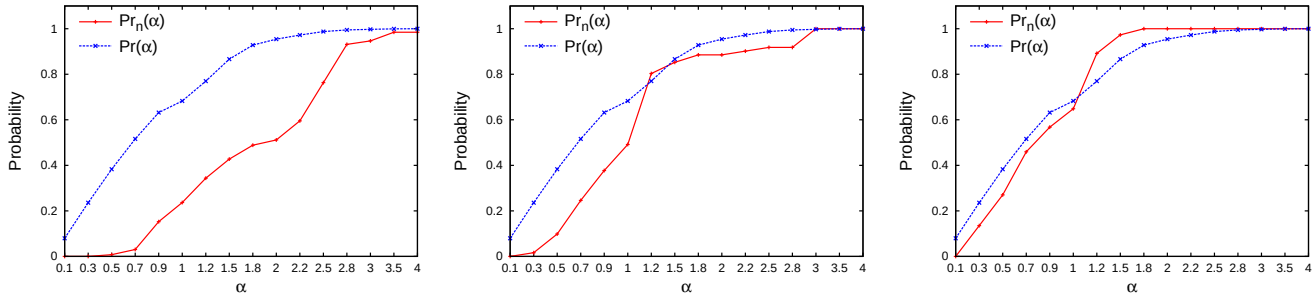
$$\Pr_n(\alpha) = \frac{1}{n}\sum_{i=1}^{n} I(e_i' \le \alpha), \text{ where } e_i' = \frac{e_i}{\sigma_i} = |\frac{t_i - \mu_i}{\sigma_i}|.$$

Here I is the indicator function. To measure the proximity of $\Pr_n(\alpha)$ and $\Pr(\alpha)$, we define

$$D_n(\alpha) = |\Pr_n(\alpha) - \Pr(\alpha)|.$$

Clearly, a smaller $D_n(\alpha)$ means $\Pr(\alpha)$ is closer to $\Pr_n(\alpha)$, which implies better quality of the distributions. We further generated α's from the interval $(0, 6)$ which is sufficiently wide for normal distributions and computed the average of the $D_n(\alpha)$'s (denoted as $\overline{D}_n$). Figure 4 reports the results for the benchmark queries over uniform TPC-H 10GB databases (see [1] for the complete results).

We observe that in most cases the $\overline{D}_n$'s are below 0.3 with the majority below 0.2, which suggests that the estimated $\Pr(\alpha)$'s are reasonably close to the observed $\Pr_n(\alpha)$'s. To shed some light on what is going on here, in Figure 5 we further plot the $\Pr(\alpha)$ and $\Pr_n(\alpha)$ for the (1) **MICRO**, (2) **SELJOIN**, and (3) **TPCH** queries over the uniform TPC-H 10GB database on PC2 when SR = 0.05, which give $\overline{D}_n$ = 0.2532, 0.1098, and 0.0535 respectively (see Figure 4). We can see that we overestimated the $\Pr(\alpha)$'s for small α's. In other words, we underestimated the prediction errors by presenting smaller than actual variances in the distributions. Moreover, we find that overestimate is more significant for the **MICRO** queries (Figure 5(a)). One possible reason is that since these queries are



(a) Case (1), $\overline{D}_n = 0.2532$ (b) Case (2), $\overline{D}_n = 0.1098$ (c) Case (3), $\overline{D}_n = 0.0535$

Figure 5: The proximity of $\Pr_n(\alpha)$ and $\Pr(\alpha)$ with respect to different $\overline{D}_n$'s

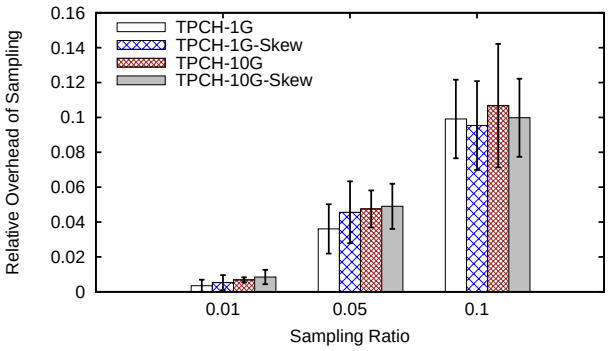


Figure 6: Relative overhead of TPCH queries on PC1

really simple the predictor tends to be over-confident by underestimating the variances even more. When handling **SELJOIN** and **TPCH** queries, the confidence of the predictor drops and underestimate tends to be alleviated (Figure 5(b) and 5(c)).

6.4 Runtime Overhead of Sampling

We also measured the relative overhead of running the queries over the sample tables compared with that of running them over the original tables. Figure 6 presents the results of the **TPCH** queries on PC1. Since the other results are very similar, the readers are referred to [1] for the complete details. We observe that the relative overhead is comparable to that reported in [39]. For instance, for the TPC-H 10GB database, the relative overhead is around 0.04 to 0.06 when the sampling ratio is 0.05. Note that, here we computed the estimated selectivities as well as their variances by only increasing the relative overhead a little. Also note that, here we measured the relative overhead based on disk-resident samples. The relative overhead can be dramatically reduced by using the common practice of caching the samples in memory [31].

6.5 Applications

We discuss some potential applications that could take advantage of the distributional information of query running times. The list of applications here is by no means exhaustive, and it is our hope that our study in this paper could stimulate further research in this direction and more applications could emerge in the future.

6.5.1 *Query Optimization*

Although significant progress has been made in the past several decades, query optimization remains challenging for many queries due to the difficulty in accurately estimating query running times. Rather than betting on the optimality of the plan generated based on (perhaps erroneous) point estimates for parameters such as selectivities and cost units, it makes sense to also consider the uncertainties of these parameters. In fact, there has been some theoretical work investigating optimization based on least *expected* cost (LEC) based upon distributions of the parameters of the cost model [12]. However, that work did not address the problem of how to obtain the distributions. It would be interesting future work to see the effectiveness of LEC plans by incorporating our techniques into query optimizers.

6.5.2 *Query Progress Monitoring*

State-of-the-art query progress indicators [10, 27] provide estimates of the percentage of the work that has been completed by a query at regular intervals during the query's execution. However, it has been shown that in the worst case no progress indictor can outperform a naive indicator simply saying the progress is between 0% and 100% [9]. Hence, information about uncertainty in the estimate of progress is desirable. Our work provides a natural building block that could be used to develop an uncertainty-aware query progress indicator: the progress indicator could call our predictor to make a prediction for the remaining query running time as well as its uncertainty.

6.5.3 *Database as a Service*

The problem of predicting query running time is revitalized by the recent move towards providing database as a service (DaaS). Many important decision-making procedures, including admission control [40], query scheduling [11], and system sizing [36], rely on estimation of query running time. Distributional information enables more robust decision procedures in contrast to point estimates. Recent work [11] has shown the benefits in query scheduling by leveraging distributional information. Similar ideas have also been raised in [40] for admission control. Again, these work did not address the fundamental issue of obtaining the distributions without running the queries. It would be interesting to see the effectiveness of our proposed techniques in these DaaS applications.

7. RELATED WORK

The problem of predicting query execution time has been extensively studied quite recently [4, 5, 16, 17, 25, 38, 39]. However, none of this work considers the problem of measuring the degree of uncertainty present in predictions. We have reused some techniques developed in [39] for computing the means of selectivities and cost units when viewed as random variables. However, [39] focused on point estimates rather than distributional information, and hence these techniques were insufficient. We have substantially extended [39] by developing new techniques for computing

variances (and hence distributions) of selectivity and cost-unit estimates (Section 3), cost functions (Section 4), and, based on that, distributions of likely running times (Section 5).

The idea of using samples to estimate selectivity goes back more than two decades ago (e.g., [7, 8, 20, 21, 23, 26]). While we focused on estimators for selection and join queries [21], some estimators that estimate the number of distinct values might be further used to refine selectivity estimates of aggregate queries [8, 20]. However, not only do we need an estimate of selectivity, we need an estimated distribution as well. So far, we are not aware of any previous study towards this direction for aggregate queries. Regarding the problem of estimating selectivity distributions for selection and join queries, there are options other than the one used in this paper. For example, Babcock and Chaudhuri [7] proposed a framework to learn the posterior distributions of the selectivities based on *join synopses* [3]. Unfortunately, this solution is restricted to SPJ expressions with foreign-key joins, due to the overhead of computing and maintaining join synopses over a large database.

The framework proposed in this paper also relies on accurate approximation of the cost models used by the optimizer. Du et al. [15] first proposed the idea of using logical cost functions in the context of heterogenous database systems. Similar ideas were later on used in developing generic cost models for main memory based database systems [28] and identifying robust plans in the plan diagram generated by the optimizer [13]. Our idea of further using optimization techniques to find the best coefficients in the logical cost functions is motivated by the approach used in [13].

8. CONCLUSION

In this paper, we take a first step towards the problem of measuring the uncertainty within query execution time prediction. We quantify prediction uncertainty using the distribution of likely running times. Our experimental results show that the standard deviations of the distributions estimated by our proposed approaches are strongly correlated with the actual prediction errors.

The idea of leveraging cost models to quantify prediction uncertainty need not be restricted to single standalone queries. As shown in [38], Equation (2) can also be used to provide point estimates for multiple concurrently-running queries. The key observation is that the selectivities of the operators in a query are independent of whether or not it is running with other queries. Hence it is promising to consider applying the techniques proposed in this paper to multi-query workloads by viewing the interference between queries as changing the distribution of the c's. We regard this as a compelling area for future work.

9. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their valuable comments. This work was supported in part by a gift from Google.

10. REFERENCES

[1] http://arxiv.org/abs/1408.6589.

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

[3] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join synopses for approximate query answering. In *SIGMOD*, 1999.

[4] M. Ahmad, S. Duan, A. Aboulnaga, and S. Babu. Predicting completion times of batch query workloads using interaction-aware models and simulation. In *EDBT*, pages 449–460, 2011.

[5] M. Akdere, U. Çetintemel, M. Riondato, E. Upfal, and S. B. Zdonik. Learning-based query performance modeling and prediction. In *ICDE*, pages 390–401, 2012.

[6] L. A. Aroian. The probability function of the product of two normally distributed variables. *Ann. Math. Statist*, 18(2):265–271, 1947.

[7] B. Babcock and S. Chaudhuri. Towards a robust query optimizer: A principled and practical approach. In *SIGMOD*, 2005.

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

[9] S. Chaudhuri, R. Kaushik, and R. Ramamurthy. When can we trust progress estimators for sql queries? In *SIGMOD*, 2005.

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

[11] Y. Chi, H. Hacigümüs, W.-P. Hsiung, and J. F. Naughton. Distribution-based query scheduling. *PVLDB*, 6(9):673–684, 2013.

[12] F. C. Chu, J. Y. Halpern, and P. Seshadri. Least expected cost query optimization: An exercise in utility. In *PODS*, pages 138–147, 1999.

[13] H. D., P. N. Darera, and J. R. Haritsa. Identifying robust plans through plan diagram reduction. *PVLDB*, 1(1):1124–1140, 2008.

[14] S. J. Devlin, R. Gnanadesikan, and J. R. Kettenring. Robust estimation and outlier detection with correlation coefficients. *Biometrika*, 62(3):pp. 531–545, 1975.

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

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

[17] 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.

[18] G. Graefe. Robust query processing. In *ICDE*, page 1361, 2011.

[19] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, pages 31–40, 2007.

[20] P. J. Haas, J. F. Naughton, S. Seshadri, and L. Stokes. Sampling-based estimation of the number of distinct values of an attribute. In *VLDB*, pages 311–322, 1995.

[21] 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.

[22] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD Conference*, pages 171–182, 1997.

[23] W.-C. Hou, G. Özsoyoglu, and B. K. Taneja. Statistical estimators for relational algebra expressions. In *PODS*, pages 276–287, 1988.

[24] C. M. Jermaine, S. Arumugam, A. Pol, and A. Dobra. Scalable approximate query processing with the dbo engine. In *SIGMOD Conference*, pages 725–736, 2007.

[25] J. Li, A. C. König, V. R. Narasayya, and S. Chaudhuri. Robust estimation of resource consumption for sql queries using statistical techniques. *PVLDB*, 5(11):1555–1566, 2012.

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

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

[28] S. Manegold, P. A. Boncz, and M. L. Kersten. Generic database cost models for hierarchical memory systems. In *VLDB*, 2002.

[29] V. Markl, V. Raman, D. E. Simmen, G. M. Lohman, and H. Pirahesh. Robust query processing through progressive optimization. In *SIGMOD Conference*, pages 659–670, 2004.

[30] J. L. Myers and A. D. Well. *Research Design and Statistical Analysis*. Lawrence Erlbaum, 2 edition, 2003.

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

[32] N. Reddy and J. R. Haritsa. Analyzing plan diagrams of database query optimizers. In *VLDB*, pages 1228–1240, 2005.

[33] S. Ross. *A First Course in Probability*. Prentice Hall, 8 edition, 2009.

[34] Scilab Enterprises. *Scilab: Free and Open Source software for numerical computation*. Scilab Enterprises, Orsay, France, 2012.

[35] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. Predictable performance for unpredictable workloads. *PVLDB*, 2(1):706–717, 2009.

[36] 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.

[37] A. Winkelbauer. Moments and absolute moments of the normal distribution. *arXiv preprint arXiv:1209.4340*, 2012.

[38] W. Wu, Y. Chi, H. Hacigümüs, and J. F. Naughton. Towards predicting query execution time for concurrent and dynamic database workloads. *PVLDB*, 6(10):925–936, 2013.

[39] W. Wu, Y. Chi, S. Zhu, J. Tatemura, H. Hacigümüs, and J. F. Naughton. Predicting query execution time: Are optimizer cost models really unusable? In *ICDE*, pages 1081–1092, 2013.

[40] 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.