Towards Predicting Query Execution Time for Concurrent and Dynamic Database Workloads

Wentao Wu[†][*]   Yun Chi[‡]   Hakan Hacıgümüş[‡]   Jeffrey F. Naughton[†]

[†]Department of Computer Sciences, University of Wisconsin-Madison
[‡]NEC Laboratories America

{wentaowu,naughton}@cs.wisc.edu, {ychi,hakan}@nec-labs.com

ABSTRACT

Predicting query execution time is crucial for many database management tasks including admission control, query scheduling, and progress monitoring. While a number of recent papers have explored this problem, the bulk of the existing work either considers prediction for a single query, or prediction for a *static* workload of concurrent queries, where by "static" we mean that the queries to be run are fixed and known. In this paper, we consider the more general problem of *dynamic* concurrent workloads. Unlike most previous work on query execution time prediction, our proposed framework is based on analytic modeling rather than machine learning. We first use the optimizer's cost model to estimate the I/O and CPU requirements for each pipeline of each query in isolation, and then use a combination queueing model and buffer pool model that merges the I/O and CPU requests from concurrent queries to predict running times. We compare the proposed approach with a machine-learning based approach that is a variant of previous work. Our experiments show that our analytic-model based approach can lead to competitive and often better prediction accuracy than its machine-learning based counterpart.

1. INTRODUCTION

The ability to predict query execution time is useful for a number of database management tasks, including *admission control* [25], *query scheduling* [10], *progress monitoring* [15], and *system sizing* [26]. The current trend of offering database as a service (DaaS) makes this capacity even more attractive, since a DaaS provider needs to honor service level agreements (SLAs) to avoid loss of revenue and reputation. Recently, there has been substantial work on query execution time prediction [2, 3, 6, 8, 13, 28]. Much of this work focuses on predicting the execution time for a single standalone query [3, 8, 13, 28], while only a fraction of this work considers the more challenging problem of predicting the execution time for multiple concurrently-running queries [2, 6].

Predicting execution time for concurrent queries is arguably more important than prediction for standalone queries, because database

systems usually allow multiple queries to execute concurrently. The existing work on concurrent query prediction [2, 6], however, assumes that the workload is *static*, namely, the queries participating in the workload are known beforehand. While some workloads certainly conform to this assumption (e.g., the *report-generation* workloads described in [1]), others do not. Real-world database workloads can be *dynamic*, in that the set of queries that will be submitted to the system cannot be known a priori.

This paper takes a first step towards predicting query execution times for database workloads that are both concurrent and dynamic. Unlike the currently dominant paradigm of machine-learning based approaches, which treat the underlying system as a black box, our approach relies on analytic models that explicitly characterize the system's query evaluation mechanisms. We first use the optimizer's cost model to estimate the I/O and CPU requirements for each query in isolation, and then use a combination queueing model and buffer pool model that merges the I/O and CPU requests from concurrent queries to predict their running times.

Specifically, for a single query, the optimizer uses the query plan, cardinality estimates, and cost equations for the operators in the plan to generate counts for various types of I/O and CPU operations. It then converts these counts to time by using system-specific parameters that capture the time each operation takes. In more detail, for specificity consider the cost model used by the PostgreSQL query optimizer:

EXAMPLE 1 (POSTGRESQL'S COST MODELS). *PostgreSQL uses a simple cost model for each operator O such that its execution cost (i.e., time) C_O can be expressed as:*

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

Here c_s, c_r, c_t, c_i, and c_o are *cost units* 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.

n_s, n_r, n_t, n_i, and n_o 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 cost of a query plan is simply the sum of the costs of the individual operators in the plan.
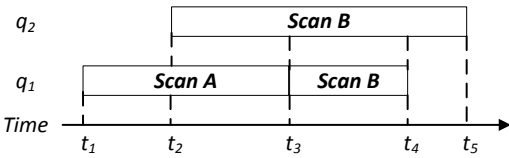
Figure 1: Interactions between queries

For multiple concurrently-running queries, one could try to build a generalization of the optimizer's cost model that explicitly takes into account the execution of other queries. For example, it could make guesses as to what might be in the buffer pool; or what fraction of the CPU this query will get at each point in execution; or which sequential I/O's will be converted to random I/O's due to interference, and so forth. But this seems very difficult if not impossible. First, the equations capturing such a complex system will be messy. Second, it requires very detailed knowledge about the exact mix of queries that will be run and how the queries will overlap (in particular, which parts of each query execution will overlap with which parts of other query's execution). This detailed knowledge is not even available in a dynamic system.

Therefore, instead of building sophisticated extensions to the optimizer's cost model, we retain the single query optimizer estimate, but stop at the point where it estimates the counts of the operations required (rather than going all the way to time). We then use a combination queueing model/buffer pool model to estimate how long each concurrently running query will take.

More specifically, we model the underlying database system with a queueing network, which treats hardware components such as disks and CPU cores as *service centers*, and views the queries as *customers* that visit these service centers. The n's of a query are then the numbers of visits it pays to the service centers, and the c's are the times spent on serving one single visit. In queueing theory terminology, the c's are the *residence times per visit* of a customer, and can be computed with the well-known mean value analysis technique [20, 23]. However, the queueing network cannot account for the cache effect of the buffer pool, which might be important for concurrent queries. Therefore, we further incorporate a model to predict buffer pool hit rate [16], based on the "clock" algorithm used by PostgreSQL.

However, queries are not uniform throughout, they change behavior as they go through different phases of their execution. Consider a query q that is concurrently running with other queries. For different operators of q, the cost units (i.e., the c's) might have different values, depending on the particular operators running inside q's neighbors. Consider the following example:

EXAMPLE 2 (QUERY INTERACTIONS). *Figure 1 shows two queries q_1 and q_2 that are concurrently running. q_1 starts at time t_1 and has 2 scan operators, with the first one scanning the table A and the second one scanning the table B. q_2 starts at time t_2 and has only 1 scan operator that scans the table B. The I/O cost units (i.e., c_s and c_r) of q_1 between t_2 and t_3 are expected to be greater than that between t_1 and t_2, due to the contention with q_2 on disk service after q_2 joins. At time t_3, the first scan of q_1 finishes, and the second one starts. The I/O cost units of q_1 (and q_2) are then expected to decrease, since the contention on disk would be less intensive for two scans over the same table B than when one is over A while the other is over B, due to potential buffer pool sharing. At time t_4, q_1 finishes and q_2 becomes the only query running in the system. The I/O cost units of q_2 are thus again expected to decrease.*

Therefore, to the queuing/buffer model, a query looks like a series of phases, each with different CPU and I/O demands. Hence, rather than applying the models at the query level, we choose to apply them to each execution phase. The remaining issue is then how to define the "phase" here. One natural choice could be to define a phase to be an individual operator. However, a number of practical difficulties arise. A serious problem is that database queries are often implemented using an *iterator* model [9]. When evaluating a query, the operators are usually grouped into pipelines, and the execution of operators inside a pipeline are *interleaved* rather than sequential. For this reason, we instead define a phase to be a pipeline. This fixes the issue of "interleaved phases" if a phase were defined as an operator. By doing this, however, we implicitly assumed the requests of a query are relatively constant during a pipeline and may only change at pipeline boundaries. In other words, we use the same c's for different operators of a pipeline. Of course, this sacrifices some accuracy compared with modeling at the operator level, and hence is a tradeoff between complexity and accuracy. Nonetheless, modeling interactions at the pipeline level is still a good compromise between doing it at the operator level and doing it at the query level.

Nonetheless, this still leaves the problem of predicting the future. Throughout the above discussion, we have implicitly assumed that no knowledge is available about queries that will arrive in the future, and our task is to estimate the running times of all concurrently running queries at any point in time. If a new query arrives, the estimates for all other running queries will change to accommodate that query. Of course, if information about future workloads were available we could use that, but this is out of the scope of this paper.

We have compared our analytic-model based approach with a machine-learning based approach that is a variant of the approach used in [1, 2]. Our experimental evaluation over the TPC-H benchmark shows that, the analytic-model based approach can lead to comparable and often better prediction accuracy than the machine-learning based approach. Compared with machine-learning based approaches, the benefit of using analytic models is three-fold:

- *Universality*: The analytic-model based approach does not rely on any training data set and thus can work reasonably well even for ad-hoc database workloads.

- *Intelligibility*: The analytic-model based approach provides a white-box perspective that explicitly interprets the working mechanism of the database system and thus is easier to understand.

- *Lightweight*: By getting rid of the expensive training phase, the analytic-model based approach is more efficient in terms of the setup time required to deploy the system.

We thus regard the use of analytic models in query time prediction for concurrent workloads as another contribution of this paper.

The rest of the paper is organized as follows. We first present our predictive framework and give some analysis in Section 2. We then describe the two approaches that combine cost estimates for concurrently-running pipelines in Section 3, where Section 3.1 describes the machine-learning based approach, and Section 3.2 describes the analytic-model based approach, respectively. We present experimental results in Section 4, summarize related work in Section 5, and conclude the paper in Section 6.

2. THE FRAMEWORK

We present the details of our predictive framework in this section. We first formally define the prediction problem we are con-
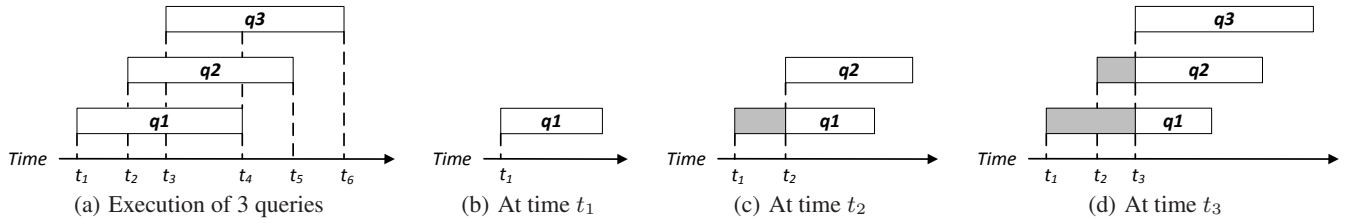
(a) Execution of 3 queries   (b) At time t_1 (c) At time t_2 (d) At time t_3

Figure 2: Illustration of the prediction problem for multiple concurrently-running queries

cerned with in this paper, then describe our solution and provide some analysis.

2.1 Problem Definition

We use an example to illustrate the prediction problem. As shown in Figure 2(a), suppose that we have three queries q_1, q_2, and q_3 that are concurrently running, which arrive at time t_1, t_2 and t_3, respectively. Accordingly, we have three prediction problems in total. At t_1, we need to predict the execution time for q_1 (Figure 2(b)). Perfect prediction here would require the information of the upcoming q_2 and q_3, which is unfortunately not available at t_1. So the best prediction for q_1 at t_1 has to be based on assuming that there will be no query coming before q_1 finishes. At t_2, q_2 joins and we need to make a prediction for both q_1 and q_2 (Figure 2(c)). For q_1, we actually predict its *remaining* execution time, since it has been running for some time (the gray part). Perfect predictions would again require the knowledge that q_3 will arrive, which is unavailable at t_2. As a result, the best prediction at t_2 needs the assumption that no query will come before q_1 and q_2 end. The same argument can be further applied to the prediction for q_1, q_2, and q_3 at t_3 (Figure 2(d)). We therefore define our prediction problem as:

DEFINITION 1 (PROBLEM DEFINITION). *Let Q be a mix of n queries $Q = \{q_1, ..., q_n\}$ that are concurrently running, and assume that no new query will come before Q finishes. Let s_0 be the start time of these n queries, and let f_i be the finish time for the query q_i. Define $T_i = f_i - s_0$ to be the execution time of q_i. The problem we are concerned with in this paper is to build a predictive model $\mathcal{M}$ for $\{T_i\}_{i=1}^n$.*

For instance, the prediction problem in Figure 2(d) is generated by setting $Q = \{q_1, q_2, q_3\}$ and $s_0 = t_3$ in the above definition.

2.2 Query Decomposition

To execute a given SQL query, the query optimizer will choose an *execution plan* for it. A plan is a tree such that each node of the tree is a physical operator, such as *sequential scan*, *sort*, or *hash join*. Figure 3 presents an example query and the execution plan returned by the PostgreSQL query optimizer.

A physical operator can be either *blocking* or *nonblocking*. An operator is blocking if it cannot produce any output tuple without reading all of its input. For instance, the operator *sort* is a blocking operator. In Figure 3, blocking operators are highlighted.

Based on the notion of blocking/nonblocking operators, the execution of the query can then be divided into multiple *pipeline*s. As in previous work [5, 14], we define pipelines inductively, starting from the leaf operators of the plan. Whenever we encounter a blocking operator, the current pipeline ends, and a new pipeline starts if any operators are remaining after we remove the current pipeline from the plan. Therefore, a pipeline always ends with a blocking operator (or the root operator). Figure 3 also shows the 5 pipelines P_1 to P_5 of the example execution plan.

By organizing concurrently running operators into pipelines, the original plan can also be viewed as a tree of pipelines, as illustrated

Tables:
Students (<u>sid</u>, sname)
Enroll (<u>sid, cid</u>)
Courses (<u>cid</u>, cname)

SELECT S.sid, S.sname
FROM Students S, Enroll E, Courses C
WHERE S.sid = E.sid **AND** E.cid = C.cid
AND S.sid > 1 **AND** S.sid < 10
AND C.cid < 5 **AND** S.sname <> 'Mike'

(a) Database and query
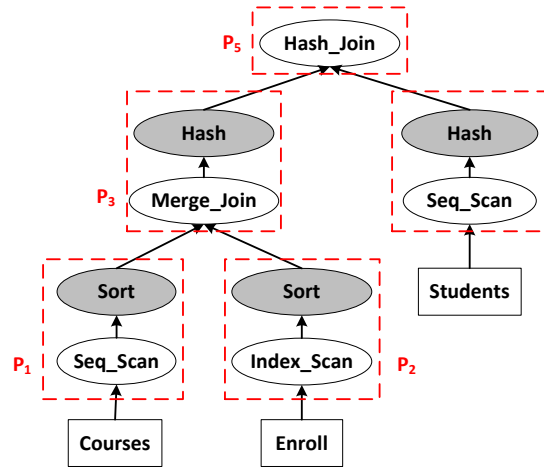
(b) Execution plan

Figure 3: Example query and its execution plan.

in Figure 3. We assume that at any time, only one pipeline of the plan is running in the database system, which is a common way in current database implementations. The execution plan thus defines a partial order over the pipelines. For instance, in the example plan, the execution of P_1 and P_2 must precede P_3, while the order between P_1 and P_2 is arbitrary. Similarly, the execution of P_3 and P_4 must precede P_5. The execution order of the pipelines can usually be obtained by analyzing the information contained in the plan. For example, in our implementation with PostgreSQL, we order the pipelines based on estimating their start times by using the optimizer's running time estimates. We then decompose the plan into a sequence of pipelines, with respect to their execution order. For the example plan, suppose that the optimizer specifies that P_1 precedes P_2 and P_3 precedes P_4. Then the plan can be decomposed into the sequence of pipelines: $P_1 P_2 P_3 P_4 P_5$.

2.3 Progressive Predictor

For the given mix of queries q_1, ..., q_n, after decomposing their execution plans into sequences of pipelines, the mix of queries can be viewed as multiple stages of mixes of pipelines. We illustrate this with the following example:

EXAMPLE 3 (MIX OF PIPELINES). *As presented in Figure 4, suppose that we have a mix of 3 queries q_1, q_2, and q_3. After decomposition of their plans, q_1 is represented as a sequence of 3*
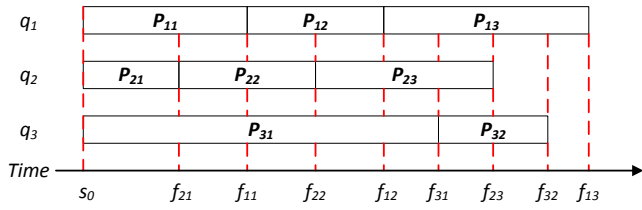
Figure 4: Progressive predictor

pipelines $P_{11}P_{12}P_{13}$, q_2 is represented as a sequence of 3 pipelines $P_{21}P_{22}P_{23}$, and q_3 is represented as a sequence of 2 pipelines $P_{31}P_{32}$. We use P_{ij} to denote the jth pipeline of the ith query, and use f_{ij} to denote the time when P_{ij} finishes. It is easy to see that whenever a pipeline finishes, we will have a new mix of pipelines. For the example query mix in Figure 4, we will thus have 8 mixes of pipelines in total, delimited by the red dash lines that indicate the finish timestamps for the pipelines.

If we could know the f_{ij}'s, then it would be straightforward to compute the execution time of the P_{ij}'s and hence the q_i. Suppose that we have some model $\mathcal{M}_{ppl}$ to predict the execution time of a pipeline by assuming that its neighbor pipelines do not change. We can then progressively determine the next finishing pipeline and therefore its finish time. For example, in Figure 4, we first call $\mathcal{M}_{ppl}$ for the mix of pipelines $\{P_{11}, P_{21}, P_{31}\}$. Based on the prediction from $\mathcal{M}_{ppl}$, we can learn that P_{21} is the next finishing pipeline and we have a new mix of pipelines $\{P_{11}, P_{22}, P_{31}\}$ at time f_{21}. We then call $\mathcal{M}_{ppl}$ for this new mix again. Note that here we also need to adjust the prediction for P_{11} and P_{31}, since they have been running for some time. We then learn that P_{11} is the next finishing pipeline for this mix and it finishes at time f_{11}. We proceed in this way until all the pipelines finish. The details of this idea are presented in Algorithm 1.

Each pipeline P_{ij} is associated with two timestamps: s_{ij}, the (predicted) start timestamp of P_{ij}; and f_{ij}, the (predicted) finish timestamp of P_{ij}. The (predicted) execution time of P_{ij} is thus $T_{ij} = f_{ij} - s_{ij}$. We also maintain the *remaining ratio* ρ_{ij}^r for P_{ij}, which is the percentage of P_{ij} that has not been executed yet. Algorithm 1 works as follows. For each query q_i, we first call the query optimizer to generate its execution plan $Plan_i$, and then decompose $Plan_i$ into a sequence of pipelines $\mathcal{P}_i$, as illustrated in Section 2.2 (line 1 to 4). The first pipeline P_{i1} in each $\mathcal{P}_i$ is added into the current mix $CurrentMix$. Its start timestamp s_{i1} is set to be 0, and its remaining ratio ρ_{i1}^r is set to be 1.0 (line 6 to 10).

Algorithm 1 then proceeds stage by stage. It makes a prediction of the initial mix of pipelines by calling the given model $\mathcal{M}_{ppl}$ (line 13). As long as the current mix is not empty, it will determine the pipeline P_{ij} with the shortest (predicted) execution time t_{min}. The current (virtual) timestamp $CurrentTS$ is forwarded by adding t_{min}. The finish time f_{ij} of P_{ij} is then set accordingly, and P_{ij} is removed from the current mix (line 15 to 20). For each remaining pipeline P_{ik} in the current mix, we update its remaining ratio ρ_{ik}^r by multiplying it by $\frac{t_{ik}^r}{t_{ik}}$, where t_{ik} is the predicted time of P_{ik} (at the beginning time of the current mix of pipelines), and t_{ik}^r is the remaining (predicted) time of P_{ik} when P_{ij} finishes and exits the current mix. $t_{ik}^r = t_{ik} - t_{min}$ by definition (line 21 to 24). Intuitively, $\frac{t_{ik}^r}{t_{ik}}$ is the *relative* remaining ratio of P_{ik} at the end of the current mix. If $\mathcal{P}_i$ contains more pipelines after P_{ij} finishes, we add the next one $P_{i(j+1)}$ into the current mix, set $s_{i(j+1)}$ to be the current timestamp, and set $\rho_{i(j+1)}^r$ to be 1.0 since the pipeline is just about to start (line 25 to 29). Note that now the current mix changes, due to removing P_{ij} and perhaps adding in $P_{i(j+1)}$. We

---

Algorithm 1: Progressive Predictor

Input: $Q = \{q_1, ..., q_n\}$, a mix of n SQL queries; $\mathcal{M}_{ppl}$: a model to predict the execution times for a mix of pipelines

Output: $\{T_i\}_{i=1}^n$, where T_i is the predicted execution time of the query q_i

1 **for** $1 \leq i \leq n$ **do**
2     $Plan_i \leftarrow GetPlan(q_i)$;
3     $\mathcal{P}_i \leftarrow DecomposePlan(Plan_i)$;
4 **end**
5
6 $CurrentMix \leftarrow \emptyset$;
7 **for** $1 \leq i \leq n$ **do**
8   Add $P_{i1} \in \mathcal{P}_i$ into $CurrentMix$;
9     $s_{i1} \leftarrow 0$; $\rho_{i1}^r \leftarrow 1.0$;
10 **end**
11
12 $CurrentTS \leftarrow 0$;
13 $MakePrediction(CurrentMix, \mathcal{M}_{ppl})$;
14 **while** $CurrentMix \neq \emptyset$ **do**
15     $t_{min} \leftarrow MinPredictedTime(CurrentMix)$;
16     $CurrentTS \leftarrow CurrentTS + t_{min}$;
17     $P_{ij} \leftarrow ShortestPipeline(CurrentMix)$;
18     $f_{ij} \leftarrow CurrentTS$;
19     $T_{ij} \leftarrow f_{ij} - s_{ij}$;
20   Remove P_{ij} from $CurrentMix$;
21   **foreach** $P_{ik} \in CurrentMix$ **do**
22        $t_{ik}^r \leftarrow t_{ik} - t_{min}$; // t_{ik} is $\mathcal{M}_{ppl}$'s prediction for P_{ik}
23        $\rho_{ik}^r \leftarrow \rho_{ik}^r \cdot \frac{t_{ik}^r}{t_{ik}}$;
24   **end**
25   **if** $HasMorePipelines(\mathcal{P}_i)$ **then**
26   Add $P_{i(j+1)}$ into $CurrentMix$;
27        $s_{i(j+1)} \leftarrow CurrentTS$;
28        $\rho_{i(j+1)}^r \leftarrow 1.0$;
29   **end**
30     $MakePrediction(CurrentMix, \mathcal{M}_{ppl})$;
31   **foreach** $P_{ik} \in CurrentMix$ **do**
32        $t_{ik} \leftarrow \rho_{ik}^r \cdot t_{ik}$;
33   **end**
34 **end**
35
36 **for** $1 \leq i \leq n$ **do**
37     $T_i \leftarrow 0$;
38   **foreach** *pipeline* P_{ij} *in* q_i **do**
39        $T_i \leftarrow T_i + T_{ij}$;
40   **end**
41 **end**
42 **return** $\{T_i\}_{i=1}^n$;

---

thus call $\mathcal{M}_{ppl}$ again for this new mix (line 30). However, we need to adjust the prediction t_{ik} for each pipeline, by multiplying it with its remaining ratio ρ_{ik}^r (line 31 to 33). The iteration then repeats by determining the next finishing pipeline.

We call this procedure the *progressive predictor*. The remaining problem is to develop the predictive model $\mathcal{M}_{ppl}$ for a mix of pipelines. We discuss our approaches in Section 3.

2.4 Analysis

We give some analysis to Algorithm 1, in terms of its efficiency and prediction errors as the number of mixes increases.

2.4.1 Efficiency

Whenever $\mathcal{M}_{ppl}$ is called, we must have one pipeline in the current mix that finishes and exits the mix. So the number of times that $\mathcal{M}_{ppl}$ is called cannot exceed the total number of pipelines in the given query mix. Thus we have

LEMMA 1. $\mathcal{M}_{ppl}$ *is called at most* $\sum_{i=1}^{n} |\mathcal{P}_i|$ *times, where* $\mathcal{P}_i$ *is the set of pipelines contained in the query* q_i.

It is possible that several pipelines may finish at the same (predicted) time. In this case, we remove all of them from the current mix, and add each of their successors (if any) into the current mix. We omit this detail in Algorithm 1 for simplicity of exposition. Note that if this happens, the number of times calling $\mathcal{M}_{ppl}$ is fewer than $\sum_{i=1}^{n} |\mathcal{P}_i|$.

2.4.2 Prediction Errors

Let the mixes of pipelines in the query mix be $M_1, ..., M_n$. For the mix M_i, let T_i and T_i' be the *actual* and *predicted* time for M_i. The prediction error D_i is defined as $D_i = \frac{T_i' - T_i}{T_i}$. So $T_i' = T_i(1 + D_i)$. If $D_i > 0$, then $T_i' > T_i$ and it is an overestimation, while if $D_i < 0$, then $T_i' < T_i$ and it is an underestimation. We can view $D_1, ..., D_n$ as i.i.d. random variables with mean μ and variance σ^2. Let D be the overall prediction error. We have

$$D = \frac{T' - T}{T} = \frac{\sum_{i=1}^{n}(T_i' - T_i)}{T} = \frac{\sum_{i=1}^{n} T_i D_i}{T},$$

where $T = \sum_{i=1}^{n} T_i$ and $T' = \sum_{i=1}^{n} T_i'$, and thus:

LEMMA 2. $E[D] = \mu$, *and* $Var[D] = \frac{\sum_{i=1}^{n} T_i^2}{\left(\sum_{i=1}^{n} T_i\right)^2} \sigma^2$.

Since $\left(\sum_{i=1}^{n} T_i\right)^2 = \sum_{i=1}^{n} T_i^2 + 2\sum_{1 \leq i < j \leq n} T_i T_j$, we have $\left(\sum_{i=1}^{n} T_i\right)^2 \geq \sum_{i=1}^{n} T_i^2$ and hence $Var[D] \leq \sigma^2$, according to Lemma 2. This means that the expected overall accuracy is no worse than the expected accuracy of $\mathcal{M}_{ppl}$ over a single mix of pipelines. Intuitively, it is because $\mathcal{M}_{ppl}$ may both overestimate and underestimate some mixes of pipelines, the errors of which are canceled with each other when the overall prediction is computed by summing up the predictions over individual pipeline mixes. So the key to improving the accuracy of the progressive predictor is to improve the accuracy of $\mathcal{M}_{ppl}$.

3. PREDICTIVE MODELS

In this section, we present the predictive model $\mathcal{M}_{ppl}$ for a mix of pipelines. $\mathcal{M}_{ppl}$ is based on the cost models used by query optimizers, which basically applies Equation (1) to each pipeline. As discussed in the introduction, the key challenge is to compute the c's in Equation (1) when the pipelines are concurrently running. In the following, we present two alternative approaches. One is a new approach based on previously proposed machine-learning techniques, and the other is a new approach based on analytic models reminiscent of those used by query optimizers. As in previous work [2, 6], we target analytic workloads and assume that queries are primarily I/O-bound.

3.1 A Machine-Learning Based Approach

The c's are related to the CPU and I/O interactions between pipelines. These two kinds of interactions are different. CPU interactions are usually *negative*, namely, the pipelines are competing with each other to share CPU cycles. On the other hand, I/O interactions can be either *positive* or *negative* [1] (see Example 2 as well). Therefore, we propose separating the modeling of CPU and I/O interactions.

For CPU interactions, we derive a simple model for the CPU-related cost units c_t, c_i, and c_o. For I/O interactions, we extend the idea from [1], using machine-learning techniques to build regression models for the I/O-related cost units c_s and c_r. Due to space limitations, in the following, we focus on illustrating the basic ideas. More details can be found in [27].

3.1.1 Modeling CPU Interactions

We use c_{cpu} to represent c_t, c_i, or c_o. Suppose that we have m CPU cores and n pipelines. Let the time to process one CPU request be τ for a standalone pipeline. If $m \geq n$, then each pipeline can have its own dedicated CPU core, so the CPU time per request for each pipeline is still τ, namely, $c_{cpu} = \tau$. If $m < n$, then we have more pipelines than CPU cores. In this case, we assume that the CPU sharing among pipelines is fair, and the CPU time per request for each pipeline is therefore $c_{cpu} = \frac{n}{m}\tau$.

3.1.2 Modeling I/O Interactions

Previous work [1] proposed an experiment-driven approach based on machine learning. The idea in that work is the following. Assuming that we know all possible queries (or query types/templates whose instances have very similar execution times) beforehand, we can then run a number of sample mixes of these queries, record their execution time as ground truth, and train a regression model with the data collected. This idea cannot be directly applied to the dynamic workloads we consider in this paper, since it requires prior knowledge of all query templates to be run.

Accordingly, we extend this idea to apply to mixes of pipelines rather than mixes of queries. As a first approximation, we assume the only I/O's performed by a query are due to scans. Later, we relax this assumption. We have the following observation:

OBSERVATION 1. *For a specific database system implementation, the number of possible* scan *operators is fixed.*

For instance, PostgreSQL implements three scan operators: *sequential scan* (SS), *index scan* (IS), and *bitmap index scan* (BIS).

We define a *scan type* to be a specific scan operator over a specific table. It is easy to see that:

OBSERVATION 2. *For a specific database system implementation and a specific database schema, the number of possible scan types is fixed.*

For example, since the TPC-H benchmark database contains 8 tables, and PostgreSQL has 3 scan operators (i.e, SS, IS, and BIS), the number of scan types in this case is 24.

Using these observations, we can apply the previous machine-learning based approach for scan types instead of query templates. Specifically, in the training stage, we collect sample mixes of scans and build regression models. For each mix of pipelines, we first identify the scans within each pipeline, and then reduce the problem to mixes of scans so that the regression models can be leveraged.

Discussion. We assumed for simplicity that the I/O's of a query were only from scans. We now return to this issue. In practice, the I/O's from certain operators (e.g., hash join) due to spilling intermediate results to disk are often not negligible. We have observed in our experiments that completely eliminating these additional I/O's from the model can harm the prediction accuracy by 10% to 30%. Therefore, we choose to incorporate these I/O's into the current model as much as possible. Specifically, we treat the additional

I/O's as if they were scans over the underlying tables. For example, PostgreSQL uses the hybrid hash join algorithm. If the partitions produced in the building phase cannot fit in memory, they will be written to disk and read back in the probing phase. This causes additional I/O's. Now suppose that $R \bowtie S$ is a hash join between the table R and S. We model these additional I/O's as additional sequential scans over R and S, respectively.

3.2 An Analytic-Model Based Approach

The machine-learning based approach suffers the problem of infinite number of unknown queries. Specifically, the sample space of training data now moves from mixes of queries to mixes of (instance) scans. Note that, although the number of scan types is finite, each scan type can have infinitely many instances. So the number of mixes of instance scans is still infinite. It could be imagined (and also verified in our experimental evaluation) that if the queries contain scans not observed during training, then the prediction is unlikely to be good.

In this section, we present a different approach based on analytic models. Specifically, we model the underlying database system with a queueing network. The c's in Equation (1) are equivalent to the *resident times per visit* of the pipelines within the network, and can be computed with standard queueing-network evaluation techniques. Since the queueing network is incapable of characterizing the cache effect of the buffer pool, we further incorporate an analytic model to predict the buffer pool hit rate.

3.2.1 The Queueing Network

As shown in Figure 5, the queueing network consists of two service centers, one for the disks, and the other for the CPU cores. This is a *closed* model with a *batch* workload (i.e., a *terminal* workload with a think time of zero) [12]. The customers of this queueing system are the pipelines in the mix. In queueing theory terminology, the execution time of a pipeline is its *residence time* in the queueing network.

If both service centers only contain a single server, then it is straightforward to apply the standard mean value analysis (MVA) technique [20] to solve the model. In practice, we usually use the approximate version of MVA for computational efficiency. The results obtained via exact and approximate MVA are close to each other [12]. However, if some service center has multiple servers, the standard technique cannot be directly used, and we instead use the extended approach presented in [23].

The queueing system shown in Figure 5 can be described by the following set of equations:

$$R_{k,m} = \tau_k + Y_k \tau_k \sum_{j \neq m} Q_{k,j}, \qquad (2)$$

$$Q_{k,j} = \frac{V_{k,j} R_{k,j}}{\sum_{i=1}^{K} V_{i,j} R_{i,j}}, \qquad (3)$$

$$Y_k = \frac{1}{C_k} \rho^{4.464(C_k^{0.676}-1)}, \qquad (4)$$

$$\rho_k = \frac{\tau_k}{C_k} \sum_{j=1}^{M} \frac{V_{k,j}}{\sum_{i=1}^{K} V_{i,j} R_{i,j}}, \qquad (5)$$

where $k \in \{cpu, disk\}$, and $1 \leq m \leq M$ (M is the number of customers). Table 1 illustrates the notations used in the above equations. Our goal is to compute the residence time $R_{k,m}$ per visit for each customer m at each service center k.

The input parameters of the equations are the τ_k's and $V_{k,m}$'s. τ_k is the mean service time per visit to the service center k. For example, τ_{disk} is the average time for the disk to perform an I/O
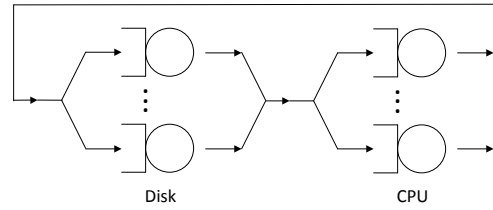

Figure 5: A queueing network

Notation	Description
C_k	# of servers in (service) center k
τ_k	Mean service time per visit to center k
Y_k	Correction factor of center k
ρ_k	Utility of center k
$V_{k,m}$	Mean # of visits by customer m to center k
$Q_{k,m}$	Mean queue length by customer m at center k
$R_{k,m}$	Mean residence time per visit by customer m to center k

Table 1: Notations used in the queueing model

operation. The τ_k's should be the same as the cost units used for estimating the execution time of a single standalone query. For PostgreSQL, however, we have 5 cost units but we only need 2 τ_k's. We address this issue by picking a *base* cost unit and transform all the other cost units into equivalent amounts of base cost units, with respect to their relative ratios. For example, for the specific machine used in our experiments (see Table 3 in Section 4), we know that $c_r = 11.3c_s$, which means the time of 1 random I/O is equivalent to 11.3 sequential I/O's. In our experiments, we pick $\tau_{disk} = c_r$ and $\tau_{cpu} = c_t$ as the base I/O and CPU cost unit (the other choices are also OK). Then the number of I/O and CPU visits $V_{k,m}$ of a pipeline are $(n_r + n_s \cdot \frac{c_s}{c_r})$ and $(n_t + n_i \cdot \frac{c_i}{c_t} + n_o \cdot \frac{c_o}{c_t})$. The n's of a pipeline are computed based on the n's of each operator in the pipeline. Specifically, suppose that a pipeline contains l operators $O_1, ..., O_l$. Let n_j (n_j can be any of the n_s, n_r, etc) be the optimizer's estimate for the operator O_j. The corresponding quantity for the pipeline is then $\sum_{j=1}^{l} n_j$.

If there is only one server in the service center k (i.e., $C_k = 1$), then $Y_k = 1$ by Equation (4). Equation (2) is then reduced to the case of standard MVA, which basically says that the residence time $R_{k,m}$ is sum of the service time τ_k and the queueing time $\tau_k \sum_{j \neq m} Q_{k,j}$. The expression of the queueing time is intuitively the sum of the queueing time of the customers other than the customer m, each of which in turn is the product of the queue length for each class (i.e., $Q_{k,j}$) and their service time (i.e., τ_k).

When there are multiple servers in the service center, intuitively the queueing time would be less than if there were only one server. The *correction factor* Y_k is introduced for this purpose. The formula of Y_k in Equation (4) was derived in [23], and was shown to be good in their simulation results.

By substituting Equation (3) to (5) into Equation (2), we can obtain a system of nonlinear equations where the only unknowns are the $R_{k,m}$'s. We use the `fsolve` function of Scilab [21] to solve this system. Any other equivalent solver can be used as well.

3.2.2 The Buffer Pool Model

The weakness of the queueing network introduced above is that it does not consider the effect of the buffer pool. Actually, since the buffer pool plays the role of eliminating I/O's, it cannot be viewed as a service center and therefore cannot be modeled within the queueing network. We hence need a special-purpose model here to predict the buffer pool hit rate. Of course, different buffer pool replacement policies need different models. We adapt the analytic model introduced in [16] for the "clock" algorithm that is used

Notation	Description
n_0	Mean # of buffer pages with count 0
m	Overall buffer pool miss rate
S_p	# of pages in partition p
r_p	Probability of accessing partition p
I_p	Maximum value of the counter of partition p
N_p	Mean # of buffer pool pages from partition p
h_p	Buffer pool hit rate of partition p

Table 2: Notations used in the buffer pool model

in PostgreSQL. If a system uses a different algorithm (e.g., LRU, LRU-k, etc), a different model should be used.

The clock algorithm works as follows. The pages in the buffer pool are organized in a circular queue. Each buffer page has a counter that is set to its maximum value when the page is brought into the buffer pool. On a buffer miss, if the requested page is not in the buffer pool and there is no free page in the buffer, a current buffer page must be selected for replacement. The clock pointer scans the pages to look for a victim. If a page has count 0, then this page is chosen for replacement. If a page has a count larger than 0, then the count is decreased by 1 and the search proceeds. On a buffer hit, the counter of the page is reset to its maximum value.

The analytic approach in [16] models this procedure by using a Markov chain. Suppose that we have P partitions in the system (we will discuss the notion of partition later). Let h_p be the buffer pool hit rate for the partition p, where $1 \leq p \leq P$. h_p can be obtained by solving the following system of equations:

$$\sum_{p=1}^{P} S_p \left(1 - \frac{1}{(1 + \frac{n_0}{m} \frac{r_p}{S_p})^{I_p+1}}\right) - B = 0, \qquad (6)$$

$$N_p = S_p \left(1 - \frac{1}{(1 + \frac{n_0}{m} \frac{r_p}{S_p})^{I_p+1}}\right), \qquad (7)$$

$$h_p = \frac{N_p}{S_p}. \qquad (8)$$

The notations used in the above equations are illustrated in Table 2.

By Equation (7) and (8),

$$m_p = 1 - h_p = \left[(1 + \frac{n_0}{m} \frac{r_p}{S_p})^{I_p+1}\right]^{-1}$$

represents the buffer *miss* rate of the partition p. Note that n_0 can be thought of as the number of buffer misses that can be handled in one clock cycle. As a result, $\frac{n_0}{m}$ is the number of buffer accesses (including both buffer hits and misses) in one clock cycle. Hence $\frac{n_0}{m} \frac{r_p}{S_p}$ is the expected number of accesses to a page in the partition p. Intuitively, the higher this number is, the more likely the page is in the buffer pool and hence the smaller m_p could be. The expression of m_p thus captures this intuition.

It is easy to see that we can determine the quantity $\frac{n_0}{m}$ from Equation (6), since it is the only unknown there. We can then figure out N_p and hence h_p by examining Equation (7) and Equation (8), respectively. To solve $\frac{n_0}{m}$ from Equation (6), define

$$F(t) = \sum_{p=1}^{P} S_p \left(1 - \frac{1}{(1 + t \cdot \frac{r_p}{S_p})^{I_p+1}}\right) - B.$$

We have $F(0) = -B < 0$, and $F(+\infty) = \lim_{t \to +\infty} F(t) = \left(\sum_{p=1}^{P} S_p\right) - B > 0$, since we except the size of the database $\sum_{p=1}^{P} S_p$ is bigger than the size of the buffer pool B (in pages). Since $F(t)$ is strictly increasing as t increases, we know that there is some $t_0 \in [0, +\infty)$ such that $F(t_0) = 0$. We can then use a simple but very efficient bisection method to find t_0 [16]. Here, B,

$\{S_p\}_{p=1}^{P}$, and $\{I_p\}_{p=1}^{P}$ are measurable system parameters. $\{r_p\}_{p=1}^{P}$ can be computed based on $\{S_p\}_{p=1}^{P}$ and the number of I/O accesses to each partition, which can be obtained from the query plans.

The remaining issue is how to partition the database. The partitioning should not be arbitrary because the analytic model is derived under the assumption that the access to database pages within a partition is uniform. An accurate partitioning thus requires information about access frequency of each page in the database, which depends on the particular workload to the system. For the TPC-H workload we used in our experiments, since the query templates are designed in some way that a randomly generated query instance is equally likely to touch each page,[1] we simplified the partitioning procedure by treating each TPC-H table as a partition. In a real deployed system, we can further refine the partitioning by monitoring the access patterns of the workload [16].

3.2.3 Put It Together

The complete predictive approach based on the analytic models is summarized in Algorithm 2. We first call the analytic model $\mathcal{M}_{buf}$ to make a prediction for the buffer pool hit rate h_p of each partition p (line 1). Since only buffer pool misses will cause actual disk I/O's, we discount the disk visits $V_{disk,i,p}$ of each partition p accessed by the pipeline i with the buffer pool miss rate $(1 - h_p)$. The disk visits $V_{disk,i}$ of the pipeline i is the sum of its visits to each partition (line 2 to 7). We then call the queueing model $\mathcal{M}_{queue}$ to make a prediction for the residence time per visit of the pipeline i in the service center k, where $k \in \{cpu, disk\}$ (line 8). The predicted execution time T_i for the pipeline i is simply $T_i = V_{cpu,i} R_{cpu,i} + V_{disk,i} R_{disk,i}$ (line 10).

Algorithm 2: $\mathcal{M}_{ppl}$ based on analytic models

Input: $\{P_1, ..., P_n\}$, a mix of n pipelines; $\mathcal{M}_{queue}$: the queueing model; $\mathcal{M}_{buf}$: the buffer pool model
Output: $\{T_i\}_{i=1}^{n}$, where T_i is the predicted execution time of the pipeline P_i

1 $\{h_p\}_{p=1}^{P} \leftarrow PredictHitRate(\mathcal{M}_{buf})$;
2 **for** $1 \leq i \leq n$ **do**
3 $\quad V_{disk,i} \leftarrow 0$;
4 $\quad$ **foreach** *partition p accessed by P_i* **do**
5 $\quad\quad V_{disk,i} \leftarrow V_{disk,i} + V_{disk,i,p}(1 - h_p)$;
6 $\quad$ **end**
7 **end**
8 $\{R_{k,i}\}_{i=1}^{n} \leftarrow PredictResTime(\mathcal{M}_{queue}, \{V_{k,i}\}_{i=1}^{n})$;
9 **for** $1 \leq i \leq n$ **do**
10 $\quad T_i \leftarrow V_{cpu,i} R_{cpu,i} + V_{disk,i} R_{disk,i}$;
11 **end**
12 **return** $\{T_i\}_{i=1}^{n}$;

It might be worth noting that, the queueing model here is equivalent to the optimizer's cost model when there is only one single pipeline. To see this, notice that the $\sum_{j \neq m} Q_{k,j}$ in the second summand of Equation (2) vanishes if there is only one customer. Therefore, we simply have $R_{k,m} = \tau_k$ in this case. Due to the use of base cost units, no information is lost when multiplying $V_{k,m}$ by

---

[1]Specifically, the TPC-H benchmark database is uniformly generated. The TPC-H queries usually use pure sequential scans or index scans with range predicates to access the tables. If it is a sequential scan, then clearly the access to the table pages is uniform. If it is an index scan, the range predicate is uniformly generated so that each page in the table is equally likely to be touched.

τ_k. Specifically, for example, suppose that $k = disk$. We have

$$V_{disk,m} \cdot \tau_{disk} = (n_r + n_s \cdot \frac{c_s}{c_r}) \cdot c_r = n_r \cdot c_r + n_s \cdot c_s,$$

which is the same as the optimizer's estimate. Since the progressive predictor degenerates to summing up the predicted time of each individual pipeline if there is only one query, the predicted execution time of the query is therefore the same as what if the optimizer's cost model is used. In this regard, for single-query execution time prediction, the analytic-model based approach here can also be viewed as a new predictor based on the optimizer's cost model, with the addition of the buffer pool model.

4. EVALUATION

In this section, we present our experimental evaluation of the proposed approaches. We measure the prediction accuracy in terms of *mean relative error* (MRE), a metric used in [2, 6]. MRE is defined as

$$\frac{1}{N} \sum_{i=1}^{N} \frac{|T_i^{pred} - T_i^{act}|}{T_i^{act}}.$$

Here N is the number of testing queries, T_i^{pred} and T_i^{act} are the predicted and actual execution times for query i. We measured the overhead of the prediction approaches as well.

4.1 Experimental Setup

We evaluated our approaches with the TPC-H 10GB benchmark database. In our experiments, we varied the multiprogramming level (MPL), i.e., the number of queries that were concurrently running, from 2 to 5. All the experiments were conducted on a machine with dual Intel 1.86 GHz CPU and 4GB of memory. We ran PostgreSQL 9.0.4 under Linux 3.2.0-26.

4.1.1 Workloads

We used the following two TPC-H-based workloads and three micro-benchmarking workloads in our experiments:

I. TPC-H workloads

• **TPC-H1**: This is a workload created with 9 TPC-H query templates that are of light to moderate weight queries. Specifically, the templates we used are TPC-H queries 1, 3, 5, 6, 10, 12, 13, 14, and 19. We choose light to moderate queries because they allow us to explore higher MPL's without overloading the system [6]. For each MPL, we then generated mixes of TPC-H queries via Latin Hypercube Sampling (LHS) [2, 6]. LHS creates a hypercube with the same dimensionality as the given MPL. Each dimension is divided into T equally probable intervals marked with 1, 2, ..., T, where T is the number of templates. The interval i represents instances of the template i. LHS then selects T sample mixes such that every value in every dimension appears in exact one mix. Intuitively, LHS has better coverage of the space of mixes than uniformly random sampling, given that the same number of samples are selected. The purpose of TPC-H1 is to compare different approaches over uniformly generated query mixes.

• **TPC-H2**: This workload is generated in the same way as TPC-H1. In addition to the 9 templates in TPC-H1, we added 3 more expensive TPC-H templates 7, 8, and 9. The purpose is to test the approaches under a more *diverse* workload, in terms of the distribution of query execution times. Figure 6 compares the variance in query execution times of TPC-H1 and TPC-H2, by presenting

the mean and standard deviation (shown as error bars) of the execution times of queries in each TPC-H template. As we can see, the execution times of some queries (e.g., Q3 and Q5) are much longer in TPC-H2 than in TPC-H1, perhaps due to the more severe interactions with the three newly-added, long-running templates.
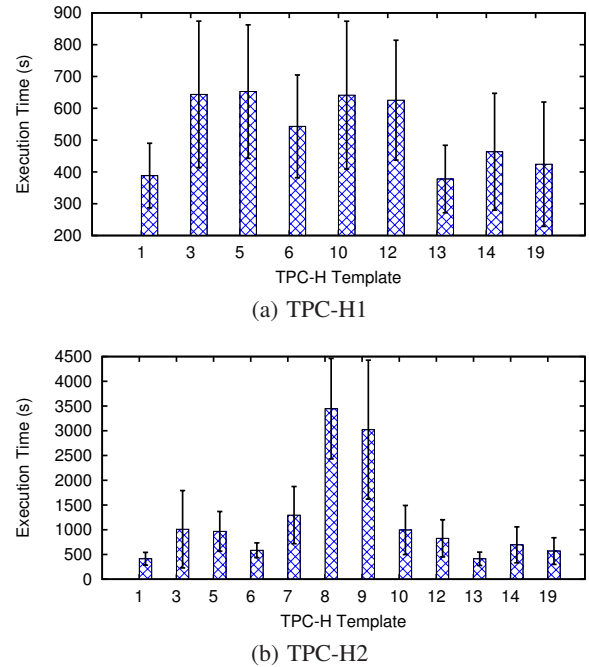

(a) TPC-H1

(b) TPC-H2

Figure 6: Variance in query execution times

II. Micro-benchmarking workloads

• **MB1**: This is a workload with 36 mixes of queries, 9 for each MPL from 2 to 5. A mix for MPL m contains m queries of the following form:

```
SELECT * FROM lineitem
WHERE l_partkey > a and l_partkey < b.
```

Here *l_partkey* is an attribute of the *lineitem* table with an unclustered index. The values of *l_partkey* is between 0 and 2,000,000. We vary a and b to produce index scans with data sharing ratio 0, 0.1, ..., 0.8. For example, when MPL is 2, if the data sharing ratio is 0, the first scan is generated with $a = 0$ and $b = 1,000,000$, and the second scan is generated with $a = 1,000,000$ and $b = 2,000,000$; if the data sharing ratio is 0.2, then the first scan is generated with $a = 0$ and $b = 1,111,111$, while the second scan is generated with $a = 888,888$ and $b = 2,000,000$. The purpose of MB1 is to compare different approaches over query mixes with different data sharing ratios.

• **MB2**: This is a workload with mixes that mingle both sequential and index scans. We focus on the two biggest tables *lineitem* and *orders*. For each table, we include 1 sequential scan and 5 index scans, and there is no data sharing between the index scans. For each MPL from 2 to 5, we generate query mixes by enumerating all possible combinations of scans. For example, when MPL is 2, we can have 10 different mixes, such as 2 sequential scans over *lineitem*, 1 sequential scan over *lineitem* and 1 sequential scan over *orders*, and so on. Whenever an index scan is required, we randomly pick one from the five candidates. The purpose of MB2 is to

compare different approaches over query mixes with different proportion of sequential and random accesses.

● **MB3**: This is a workload similar to MB2, for which we replace the scans in MB2 with TPC-H queries. We do this by classifying the TPC-H templates based on their scans over the *lineitem* and *orders* table. For example, the TPC-H $Q1$ contains a sequential scan over *lineitem*, and $Q13$ contains a sequential scan over *orders*. When generating a query mix, we first randomly pick a TPC-H template containing the required scan, and then randomly pick a TPC-H query instance from that template. The purpose of MB3 is to repeat the experiments on MB2 with less artificial, more realistic query mixes.

4.1.2 Calibrating PostgreSQL's Cost Models

Optimizer Parameter	Calibrated μ (ms)	Default
seq_page_cost (c_s)	8.52e-2	1.0
rand_page_cost (c_r)	9.61e-1	4.0
cpu_tuple_cost (c_t)	2.04e-4	0.01
cpu_index_tuple_cost (c_i)	1.07e-4	0.005
cpu_operator_cost (c_o)	1.41e-4	0.0025

Table 3: Actual values of PostgreSQL optimizer parameters

Both the machine-learning and analytic-model based approaches need the c's and n's from the query plan as input. However, the crude values of these quantities might be incorrect and hence are not ready for use. The default values of the c's are usually arbitrarily set by the optimizer developers based on their own experience, and therefore are often not correct for a specific hardware configuration. Meanwhile, the n's are closely related to *cardinality estimation*, which are also likely to be erroneous, if the assumptions used by the optimizer such as uniformity and independence do not hold on the real data distribution. We can use the framework proposed in our recent work [28] to calibrate the c's and n's. For the c's, a set of calibration queries are used to profile the underlying database system. Table 3 presents the calibrated values for the 5 cost units on the machine used in our experiments. For the n's, a sampling-based approach is used to refine the cardinality estimates. In the following experiments, we set the sampling ratio to be 0.05 (i.e., the sample size was 5% of the database size). We note here that the prediction accuracy of our proposed approaches observed on the tested workloads is quite close to that observed using the *true* cardinalities to compute the n's (see [27] for a comparative study).

4.1.3 Settings for Machine Learning

As mentioned before, the TPC-H benchmark database consists of 8 tables, 6 of which have indexes. Also, there are 3 kinds of scan operators implemented by PostgreSQL, namely, sequential scan (SS), index scan (IS), and bitmap index scan (BIS). Therefore, we have 8 SS scan types, one for each table, and 6 IS scan types, one for each table with some index. Since BIS's are rare, we focus on the two biggest tables *lineitem* and *orders* for which we observed the occurrences of BIS's in the query plans. By including these 2 BIS scan types, we have 16 scan types in total. We then use Latin Hypercube Sampling (LHS) to generate sample mixes of scan types. For a given sample mix, we further randomly generate an instance scan for each scan type in the mix. Since we have 16 scan types, each run of LHS can generate 16 sample mixes. While we can run LHS many times, executing these mixes to collect training data is costly. Hence, for each MPL, we run LHS 10 times.

We used the features in Table 4 to represent an scan instance s_i in a mix $\{s_1, ..., s_n\}$, where tbl_i is the table accessed by s_i, and

ID	Description
F1	# of sequential I/O's of s_i
F2	# of random I/O's of s_i
F3	# of scans in $N(s_i)$ that are over tbl_i
F4	# of sequential I/O's from scans in $N(s_i)$ that are over tbl_i
F5	# of random I/O's from scans in $N(s_i)$ that are over tbl_i
F6	# of scans in $N(s_i)$ that are *not* over tbl_i
F7	# of sequential I/O's from scans in $N(s_i)$ that are *not* over tbl_i
F8	# of random I/O's from scans in $N(s_i)$ that are *not* over tbl_i

Table 4: Features of s_i

$N(s_i)$ is the set of neighbor scans of s_i in the mix. We tested representatives of both linear models and nonlinear models. For linear models, we used multivariate linear regression (MLR), and for nonlinear models, we used REP regression trees (REP) [18]. We also tested the well-known boosting technique that combines predictions from multiple models, which is generally believed to be better than a single model. Specifically, we used additive regression [7] here, with shallow REP trees as base learners. All of these models can be obtained from the WEKA software package [11]. In our study, we found that REP trees outperformed both linear regression and additive regression, in terms of prediction accuracy. Therefore, in this paper we present the results of the machine-learning based approach by using the REP trees. Readers are referred to [27] for more results.

4.1.4 Settings for Analytic Models

The queueing model needs the calibrated c's (in Table 3) and n's as input. In addition, the buffer pool model also requires a dozen parameters. Table 5 lists the values of these parameters for the system and database configurations used in our experiments.

Parameter	Description	Value
B	# of buffer pool pages	439,463
I_p	Max counter value (for all p)	5
$S_{lineitem}$	# of pages in *lineitem*	1,065,410
S_{orders}	# of pages in *orders*	253,278
$S_{partsupp}$	# of pages in *partsupp*	170,916
S_{part}	# of pages in *part*	40,627
$S_{customer}$	# of pages in *customer*	35,284
$S_{supplier}$	# of pages in *supplier*	2,180
S_{nation}	# of pages in *nation*	1
S_{region}	# of pages in *region*	1

Table 5: Values of buffer pool model parameters

4.2 Prediction Accuracy

We evaluated the accuracy of our approaches with the five workloads described in Section 4.1.1. To see the effectiveness of our approaches, in our evaluation we also included a simple baseline approach:

Baseline: For each query in the mix, predict its execution time as if it were the only query running in the database system, by using the method described in [28]. Then multiply it with the MPL (i.e., the number of queries in the mix) as the prediction for the query. Intuitively, this approach ignores the impact of interactions from different neighbors of the query. It will produce the same prediction for the query as long as the MPL is not changed.

4.2.1 Results on TPC-H Workloads

Figure 7 and 8 present the prediction errors over the two TPC-H-based workloads TPC-H1 and TPC-H2. On TPC-H1, the accuracy of the analytic-model based and the machine-learning based
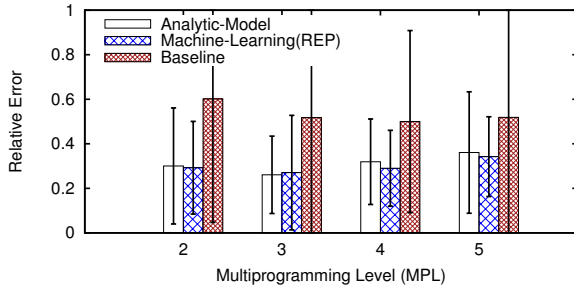
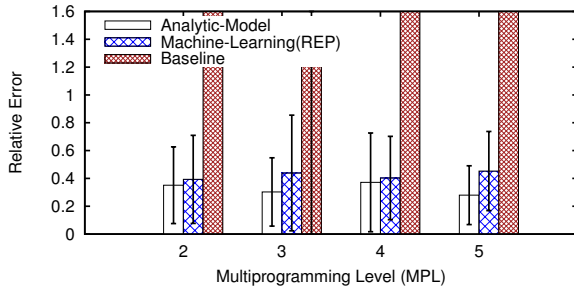**Figure 7: Prediction error on TPC-H1 for different approaches**

Figure 8: Prediction error on TPC-H2 for different approaches

Figure 9: Prediction error on MB1 for different approaches

Figure 10: Prediction error on MB2 for different approaches

approach are close, both outperforming the baseline approach by reducing the error by 15% to 30% (Figure 7). This performance improvement may not be very impressive in view of the simplicity of the baseline approach. However, we note here that this is because of the way the workload is generated rather than a problem in our approaches. The workload TPC-H1 turns out to be relatively easy to predict (the errors of all approaches are relatively small). When we move to the more diverse workload TPC-H2, the prediction accuracy of the baseline approach deteriorates dramatically (Figure 8), while its two competitors retain prediction accuracy close to that observed on TPC-H1. Nonetheless, it is important to include the baseline to show that sometimes it does surprisingly well, and makes it challenging to improve substantially over that baseline. We also observe that, on TPC-H2, the analytic-model based approach slightly outperforms the machine-learning based approach, improving the prediction accuracy by about 10%.

4.2.2 Results on Micro-Benchmarking Workloads

Since the TPC-H workloads were generated via LHS, they cover only a small fraction of the space of possible query mixes. As a result, many particular kinds of query interactions might not be captured. We therefore evaluated the proposed approaches over the three micro-benchmarking workloads as well, which were more diverse than the TPC-H workloads in terms of query interactions. Figure 9 to 11 present the results.

On MB1, the prediction errors of the machine-learning based and the baseline approach are very large, while the errors of the analytic-model based approach remain small (Figure 9). The baseline approach fails perhaps because it does not take the data sharing between queries into consideration. We observed consistent overestimation made by the baseline approach, while the analytic-model based approach correctly detected the data sharing and hence leveraged it in buffer pool hit rate prediction. The machine-learning based approach is even worse than the baseline approach. This is because we train the model with mixes of scans generated via LHS, which are quite different from the mixes of scans in MB1. MB1 fo-
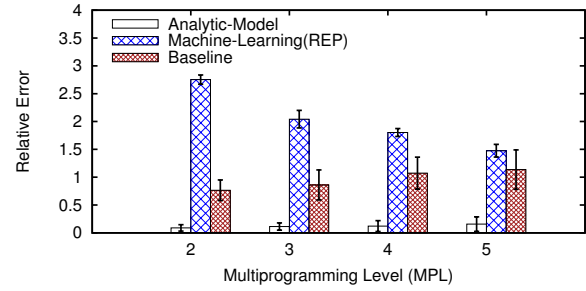
cuses on heavy index scans over a particular table. In typical LHS runs, very few samples can be obtained from such a specific region since the goal of LHS is to uniformly cover the whole huge space of query mixes.
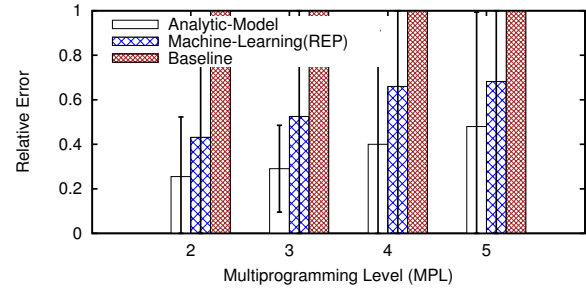
The prediction errors of the baseline approach remain large on the workloads MB2 and MB3 (Figure 10 and 11). This is not surprising, since the query interactions in MB2 and MB3 are expected to be much more diverse than they are in the TPC-H workloads. It is hard to believe that a model ignoring all these interactions can work for these workloads. Meanwhile, the analytic-model based approach is still better than the machine-learning based approach on MB2, by reducing the prediction errors by 20% to 25%, and they are comparable on MB3. One possible reason for this improvement of the machine-learning based approach may be that the interactions in MB2 and MB3 are closer to what it learnt during training. Recall that we intentionally enforce no data sharing among the index scans used to generate MB2 and MB3, and hence the index scans are somewhat independent of each other. This is similar to what LHS did in training, for which the scans in a mix are independently generated. This is quite different for MB1, however, where the queries are correlated due to data sharing.
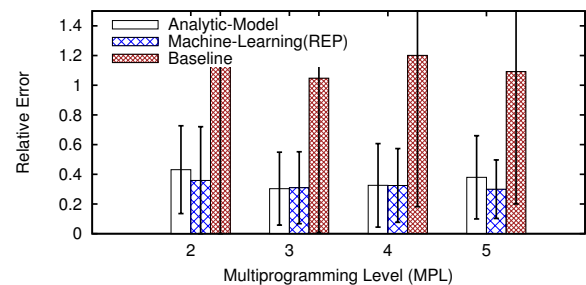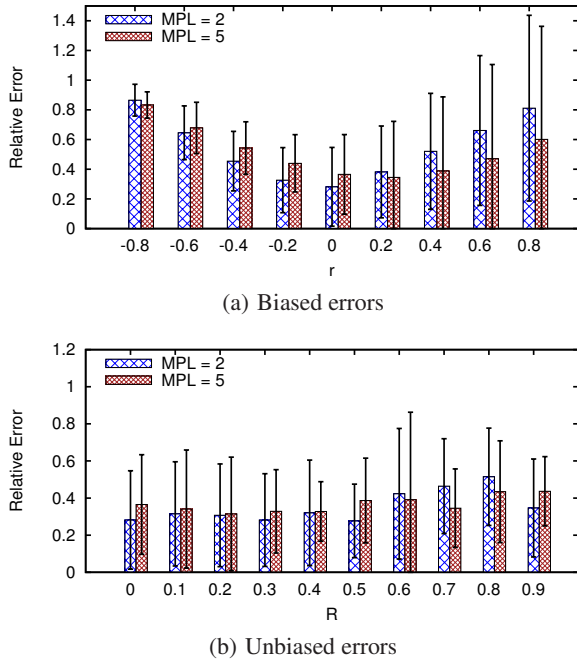
4.2.3 Sensitivity to Errors in Cardinality Estimates

Figure 11: Prediction error on MB3 for different approaches

Both the machine-learning based and the analytic-model based approach rely on the n's from the query plans. Since the accuracy of the n's depends on the quality of cardinality estimates, which are often erroneous in practice, a natural question is how sensitive the proposed approaches are to errors in cardinality estimates.

We investigated this question for the analytic-model based approach, which, as we have seen, outperformed its machine-learning counterpart on the workloads we tested. We studied this by feeding the optimizer cardinalities generated by perturbing the *true* cardinalities. Specifically, consider an operator O with true input cardinality N_O. Let r be the *error rate*. In our perturbation experiments, instead of using N_O, we used $N_O' = N_O(1+r)$ to compute the n's of O. We considered both *biased* and *unbiased* errors. The errors are biased if we use the same error rate r for all operators in the query plan, and the errors are unbiased if each operator uniformly randomly draws r from some interval $(-R, R)$.

(a) Biased errors

(b) Unbiased errors

Figure 12: Sensitivity of prediction accuracy on TPC-H1

Figure 12 shows the results on the TPC-H1 workload. We observe that in the presence of biased errors, the prediction errors increase in proportion to the errors in cardinality estimates. However, the prediction errors often increase more slowly than the cardinality estimation errors. For example, in Figure 12(a), the mean prediction error increases from 0.36 to 0.47 for MPL 5 when r increases from 0 to 0.6. On the other hand, the prediction accuracy is more stable in the presence of unbiased errors. As shown in Figure 12(b), the prediction errors are almost unchanged when R increases from 0 to 0.4. The intuition for this is that if the errors are unbiased, then for each operator in the query plan, it is equally likely to overestimate or underestimate its cardinalities. Therefore, the errors might cancel each other when making prediction for the entire query. More results and analysis can be found in [27].

4.3 Additional Overhead

Both the machine-learning based and the analytic-model based approach need to calibrate the optimizer's cost model. As shown in [28], calibrating the c's is a one-time procedure and usually can be done within a couple of hours. The overhead of calibrating the
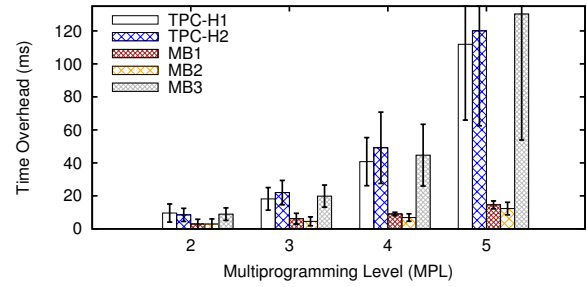
Figure 13: Runtime overhead in evaluating analytic models

n's via sampling depends on the sample size. For the sampling ratio 0.05, it takes around 4% of the query execution time, when the samples are disk-resident. This overhead could be substantially reduced using the common techniques of keeping the samples in memory [19].

In addition to the overhead in calibrating the cost model, the machine-learning based approach needs to collect the training data. Although the training is offline, this overhead is not trivial. The time spent in the training stage depends on several factors, such as the number of sample scan mixes and the overhead of each scan instance. For the specific settings used in our experiments, the training stage usually takes around 2 days.

On the other hand, the analytic-model based approach needs to evaluate the analytic models when making the prediction. This includes the time of solving the systems of nonlinear equations required by both the queueing model and the buffer pool model. Figure 13 presents the average total time spent in the evaluation as well as the standard deviation (as error bars). As expected, the time overhead increases as the MPL grows, since the queueing model becomes more complicated. However, the overall time overhead is ignorable (e.g., around 120 ms when MPL is 5), compared with the execution time of the queries (usually hundreds of seconds).

5. RELATED WORK

Predicting query execution time has recently gained significant interest in database research community [2, 3, 6, 8, 13, 28]. In [8], the authors considered the problem of predicting multiple performance metrics such as execution time and disk I/O's for database queries, by representing the queries with a set of handpicked features and using Kernel Canonical Correlation Analysis (KCCA) [4] as the predictive model. A similar idea was proposed in [3], where the authors advocated the use of support vector machines (SVM) instead of KCCA as the specific machine-learning model. They further proposed a different approach by first building individual predictive models for each physical operator and then combining their predictions. In [13], the authors focused on using machine learning to estimate CPU time and logical I/O's of a query execution plan, and addressed the problem of robust estimation for queries not observed in the training stage. Different from these machine-learning based approaches, in [28] we proposed approaches based on refining the query optimizer's cost estimates and showed comparable or even better prediction accuracy. None of them, however, addressed the problem of concurrent queries.

The problem of predicting concurrent query execution time was studied in [2] and [6]. In [2], the authors proposed an experiment-driven approach by sampling the space of possible query mixes and fitting statistical models to the observed execution time of these samples. Specifically, they used Gaussian processes as the particular statistical model. A similar idea was used in [6], where the

authors proposed predicting the buffer access latency (BAL) of a query, which is the average delay between the time when an I/O request is issued and the time when the requested block is returned. BAL was found to be highly correlated with the query execution time, and they simply used linear regression mapping BAL to the execution time. To predict BAL, the authors collected training data by measuring the BALs under different query mixes and then built a predictive model based on multivariate regression. The key limitation of both work is that they both assumed static workloads, which is usually not the case in practice. To the best of our knowledge, we are the first that addresses the concurrent query execution time prediction problem under dynamic workloads.

Queueing networks have been extensively used in computer system modeling, including database systems (e.g., [17, 22, 24]). However, the focus in this work is quite different from ours. Previous work used queueing networks to predict macro performance metrics such as the throughput and mean response time for different workloads. Their goal, as pointed out by Sevcik [22], was "predicting the *direction* and approximate *magnitude* of the change in performance caused by a particular design modification." As a result, the models were useful as long as they could correctly predict the trend in system performance, although "significant errors in absolute predictions of performance" were possible. In contrast, our goal is to predict the exact execution time for each individual query. Due to this discrepancy, we applied queueing networks in a quite different manner. Previous work modeled the system as an open network (e.g., [17, 24]), the evaluation of which heavily relies on assumptions about query arrival rates and service time distributions (e.g., M/M/1 and M/G/1 queues). In contrast, we do not assume any additional workload knowledge except for the current query mix to be predicted, since we target dynamic workloads. Therefore, we modeled the system as a closed network, and used the mean value analysis (MVA) technique to solve the model. Moreover, we treated pipelines rather than the entire queries as customers of the queueing network, motivated by the observation that query interactions happen at the pipeline level rather than at the query level. We further incorporated the progressive predictor (Section 2.3) to stitch together the predictions for pipeline mixes and the buffer pool model (Section 3.2) to account for the effect of the buffer pool. Without these constructs, the prediction accuracy by directly applying queueing network theory would often be awful (see [27]).

6. CONCLUSION

In this paper, we studied the problem of predicting query execution time for concurrent and dynamic database workloads. Our approach is based on analytic models, for which we first use the optimizer's cost model to estimate the I/O and CPU operations for each individual query, and then use a queueing model to combine these estimates for concurrent queries to predict their execution times. A buffer pool model is also incorporated to account for the cache effect of the buffer pool. We show that our approach is competitive to and often better than a variant of previous machine-learning based approaches, in terms of prediction accuracy.

We regard this paper as a first step towards this important but challenging problem. To improve the prediction accuracy, one could either try new machine-learning techniques or develop better analytic models. While previous work favored the former option, the results shown in this paper shed some light on the latter one. Moreover, a hybrid approach combining the merits of both approaches is worth consideration for practical concern, since most database workloads are neither *purely* static nor *purely* dynamic. All these directions deserve future research effort.

7. REFERENCES

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

[2] 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.

[3] 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.

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

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

[6] J. Duggan, U. Çetintemel, O. Papaemmanouil, and E. Upfal. Performance prediction for concurrent database workloads. In *SIGMOD*, pages 337–348, 2011.

[7] J. H. Friedman. Stochastic gradient boosting. *Comput. Stat. Data Anal.*, 38(4):367–378, 2002.

[8] 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.

[9] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

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

[11] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA data mining software: an update. *SIGKDD Explorations*, 11(1):10–18, 2009.

[12] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative system performance - computer system analysis using queueing network models*. Prentice Hall, 1984.

[13] 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.

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

[15] C. Mishra and N. Koudas. The design of a query monitoring system. *ACM Trans. Database Syst.*, 34(1):1–51, 2009.

[16] V. F. Nicola, A. Dan, and D. M. Dias. Analysis of the generalized clock buffer replacement scheme for database transaction processing. In *SIGMETRICS*, pages 35–46, 1992.

[17] R. Osman, I. Awan, and M. E. Woodward. Application of queueing network models in the performance evaluation of database designs. *Electr. Notes Theor. Comput. Sci.*, 232:101–124, 2009.

[18] J. R. Quinlan. Simplifying decision trees, 1986.

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

[20] M. Reiser and S. S. Lavenberg. Mean-value analysis of closed multichain queuing networks. *J. ACM*, 27(2):313–322, 1980.

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

[22] K. C. Sevcik. Data base system performance prediction using an analytical model (invited paper). In *VLDB*, pages 182–198, 1981.

[23] R. Suri, S. Sahu, and M. Vernon. Approximate mean value analysis for closed queueing networks with multiple-server stations. In *Proc. of Industrial Engineering Research Conf. (IERC)*, 2007.

[24] N. Tomov, E. W. Dempster, M. H. Williams, A. Burger, H. Taylor, P. J. B. King, and P. Broughton. Analytical response time estimation in parallel relational database systems. *Parallel Computing*, 30(2):249–283, 2004.

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

[26] 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*, pages 7–13, 2004.

[27] W. Wu, Y. Chi, H. Hacıgümüş, and J. F. Naughton. Towards predicting query execution time for concurrent and dynamic database workloads. Technical Report TR-NECLA-DM-2013-13, NEC Laboratories America, 2013.

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