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* [32], *query scheduling* [11], *progress monitoring* [19], and *system sizing* [33]. 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 [3, 4, 7, 9, 15, 34]. Much of this work focuses on predicting the execution time for a single standalone query [4, 9, 15, 34], while only a fraction of this work considers the more challenging problem of predicting the execution time for multiple concurrently-running queries [3, 7].

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 [3, 7], 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 [2]), 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.

For multiple concurrently-running queries, one could try to build a generalization of the optimizer's cost model that explicitly takes

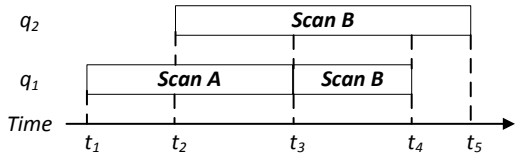


Figure 1: Interactions between queries

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 [26, 29]. 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 [20], 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 [10]. 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 [2, 3]. 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.

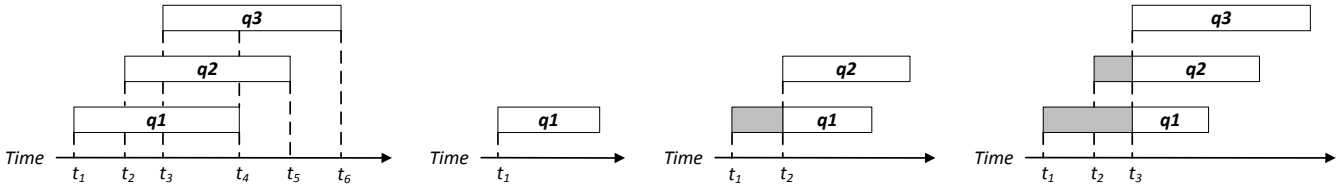
• *Lightweight*: By getting rid of the expensive training phase, the analytic-model based approach is more efficient in terms of the setup time required to deploy the system.

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



(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

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.

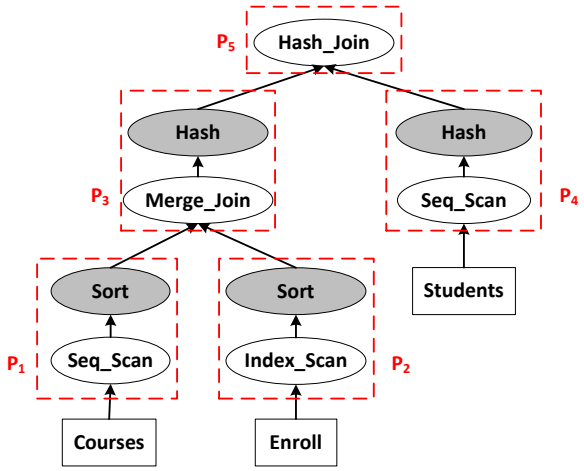
Based on the notion of blocking/nonblocking operators, the execution of the query can then be divided into multiple *pipelines*. As in previous work [6, 17], we define pipelines inductively, starting from the leaf operators of the plan. Whenever we encounter a blocking operator, the current pipeline 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 in Figure 3. We assume that at any time, only one pipeline of the

Tables:
Students (sid, sname)
Enroll (sid, cid)
Courses (cid, 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.

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: $P_1P_2P_3P_4P_5$.

We further note here that, while in PostgreSQL each time there is only one pipeline running for each query (since each query is a single process in PostgreSQL), it is reasonable that multiple independent pipelines in a query could be executed in parallel on multi-core machines. To incorporate this into the current framework, we would require further information from the query plan about the parallelism of operators in their execution. We might then be able to represent the execution of a query as a dependency graph of pipelines (currently it is a chain of pipelines), and the queueing model (see Section 3.2.1) would sometimes have more than n

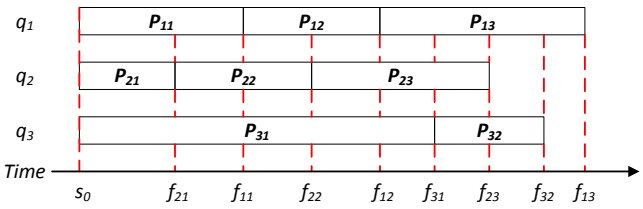


Figure 4: Progressive predictor

pipelines (suppose that we have n queries). However, since we have not tried such database systems yet, it is difficult for us to figure out the details at this time. We leave the investigation of this aspect on parallel pipelines within the same query as one for interesting future work.

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 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^r_{ij} 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^r_{i1} 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

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 ≤ i ≤ n do
2      Plan_i ← GetPlan(q_i);
3      P_i ← DecomposePlan(Plan_i);
4  end
5
6  CurrentMix ← ∅;
7  for 1 ≤ i ≤ n do
8      Add P_i1 ∈ P_i into CurrentMix;
9      s_i1 ← 0; ρ^r_i1 ← 1.0;
10 end
11
12 CurrentTS ← 0;
13 MakePrediction(CurrentMix, M_ppl);
14 while CurrentMix ≠ ∅ do
15     t_min ← MinPredictedTime(CurrentMix);
16     CurrentTS ← CurrentTS + t_min;
17     P_ij ← ShortestPipeline(CurrentMix);
18     f_ij ← CurrentTS;
19     T_ij ← f_ij − s_ij;
20     Remove P_ij from CurrentMix;
21     foreach P_ik ∈ CurrentMix do
22         t^r_ik ← t_ik − t_min; // t_ik is M_ppl's prediction for P_ik
23         ρ^r_ik ← ρ^r_ik · t^r_ik / t_ik;
24     end
25     if HasMorePipelines(P_i) then
26         Add P_i(j+1) into CurrentMix;
27         s_i(j+1) ← CurrentTS;
28         ρ^r_i(j+1) ← 1.0;
29     end
30     MakePrediction(CurrentMix, M_ppl);
31     foreach P_ik ∈ CurrentMix do
32         t_ik ← ρ^r_ik · t_ik;
33     end
34 end
35
36 for 1 ≤ i ≤ n do
37     T_i ← 0;
38     foreach pipeline P_ij in q_i do
39         T_i ← T_i + T_ij;
40     end
41 end
42 return {T_i}_{i=1}^n;
```

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^r_{ik} by multiplying it by $\frac{t^r_{ik}}{t_{ik}}$, where t_{ik} is the predicted time of P_{ik} (at the beginning time of the current mix of pipelines), and t^r_{ik} is the remaining (predicted) time of P_{ik} when P_{ij} finishes and exits the current mix. $t^r_{ik} = t_{ik} - t_{min}$ by definition (line 21 to 24). Intuitively, $\frac{t^r_{ik}}{t_{ik}}$ 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 $\rho^r_{i(j+1)}$ 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 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^r_{ik} (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 $\left(\sum_{i=1}^{n} T_i\right)^2 = \sum_{i=1}^{n} T_i^2 + 2\sum_{1 \le i < j \le n} T_i T_j$, we have $\left(\sum_{i=1}^{n} T_i\right)^2 \ge \sum_{i=1}^{n} T_i^2$ and hence $Var[D] \le \sigma^2$, 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 [3, 7], 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 on sharing CPU usage. On the other hand, I/O interactions can be either *positive* or *negative* [2] (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 in [2] based on machine-learning techniques to build regression models for the I/O-related cost units c_s and c_r. Ideally, we need to handle c_s and c_r separately, and hence we need the ground truth of c_s and c_r in the training data. Unfortunately, while we can know the total I/O time of a query, we have no idea of how much time is spent on sequential and random I/O's respectively. Therefore, we are not able to build separate predictive models for c_s and c_r. Instead we build a single model to predict c_{disk}, and the I/O time is computed as $(n_s + n_r)c_{disk}$. c_{disk} thus can be thought of as the *average* I/O time per request.

3.1.1 Modeling CPU Interactions

In the following discussion, 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 $m \ge n$, 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 $c_{cpu} = \frac{n}{m}\tau$. Of course, the model here is simplified. In practice, the CPU sharing among pipelines is not perfect, and some CPU cores may become bottlenecks due to the unbalanced assignment of pipelines to CPU cores. On the other hand, it may be too pessimistic to assume that CPU contention will always happen. In practice, due to CPU and I/O interleaving, not every pipeline is trying to seize the CPU at the same time. So the real CPU contention can be less intense than assumed by the model. We leave the improvement of modeling CPU interactions for future work.

3.1.2 Modeling I/O Interactions

In previous work [2], the authors proposed an experiment-driven approach based on machine learning. The idea is, assuming that we know all possible queries (or query types/templates whose instances have very close execution time) 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, however, cannot be directly applied to dynamic workloads, since the number of possible unknown queries can be infinitely many.

We now extend this idea to mixes of pipelines. 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.*

Basically, there are two kinds of scan operators when accessing a table. One is *sequential scan* (SS), which directly scans the table, one page after another. The other is *index scan* (IS), which first finds the relevant search keys via some index, and then fetches the corresponding records from the table if necessary. A particular database system may also have other variants of these two scan operators. For example, PostgreSQL also implements another version of index scan called *bitmap index scan* (BIS). It first builds a list for the qualified record ID's by looking up the index, and then fetches the corresponding table pages according to their *physical order* on the disk. If the number of qualified records is big (but still much smaller than the total number of records in the table), then bitmap index scans can be more efficient than pure index scans. Nonetheless, the total number of scan operators in a specific database system is fixed.

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.

Based on these observations, we can use the previous machine-learning based approach for scan types instead of queries. 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. Next, we discuss the feature selection and model selection problem for this learning task.

Feature Selection. In [2], the following features were used. Let $Q_1, ..., Q_m$ be the query types, and let $\{q_1, ..., q_n\}$ be the mix of queries to predict. The feature vector of the mix is then $(N_1, ..., N_m)$, where N_j is the number of q_i's that are instances of the type Q_j. However, this is based on the assumption that the instances of Q_j have very similar execution time. In our case, different instances from the same scan type can have quite different execution time. For example, an index scan with an equality predicate is expected to be much faster than one with a range predicate.

We therefore add additional information from the query plan as features as well. Intuitively, the I/O interactions are related to the tables that the scans touch. If two scans are over different tables, then they will interact negatively due to contention for the buffer pool. Furthermore, if the two tables are on the same disk, this may introduce additional random I/O's into the scans and hence cause more negative interactions between these two scans.However, if two scans are over the same table, then they may also benefit each other due to buffer pool sharing.

Moreover, the I/O interactions are also related to the number of I/O's the scans perform. For instance, two huge scans over different tables are likely to suffer more severe buffer pool contention than two small scans, while two huge scans over the same table may perhaps benefit more on buffer pool sharing than two small scans. For these reasons, we use the features in Table 1 to represent an (instance) scan s_i in the mix, where tbl_i is the table accessed by s_i, and $N(s_i)$ is the set of neighbor scans of s_i in the mix.

The features F3 to F8 might be further split for each different scan type. For example, F3 can be split into the number of neighbor SS, IS and BIS instances that are over tbl_i. We compared both options in our experiments. In addition, there is a special case when a scan is over the inner relation of a nested-loop join operator. In this case, the scan is performed multiple times, and therefore the associated feature values such as F1 and F2 should be scaled up accordingly.

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 1: Features of s_i

Model Selection. 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) [24]. 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 [8] here, with shallow REP trees as base learners. All of these models can be obtained from the WEKA software package [12].

Training. To train the model, we constructed a set of training mixes of queries. The queries were designed to be scans, and we achieved this by using the very simple query template:

```
SELECT * FROM R WHERE condition
```

Here `R` is a table, and `condition` is a selection predicate over the attributes of `R`. We used predicates with different selectivities so that the query optimizer could pick different scan operators.

For each scan type, we generated a number of instance scans. For each multiprogramming level (MPL), we then generated mixes of instance scans via Latin Hypercube Sampling (LHS) [3, 7]. 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 scan types. The interval i represents instances of the scan type 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.

After generating the training mixes of scans, we need to run them to collect their execution times. Note that, the phrase "the execution time of a scan when it is running with other scans" implicitly assumes that the neighbors will not change during the execution of the scan. To simulate this, we kept on running each scan in the mix, until every scan finished at least k times (we set $k = 3$ in our experiments). This means, whenever a scan finished, we would immediately run it again. We took the average of the k execution times recorded as the ground truth for the scan.

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. The additional I/O's are then deemed as additional sequential scans over R and S, respectively.

3.2 An Analytic-Model Based Approach

The machine-learning based approach suffers the same problem of infinite number of unknown queries as before. Specifically, the sample space of training data 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) [14]. 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 [26] 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 [14]. However, if some service center has multiple servers, the standard technique cannot be directly used, and we instead use the extended approach presented in [29].

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

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

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

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

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

where $k \in \{cpu, disk\}$, and $1 \leq m \leq M$ (M is the number of customers). Table 2 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.

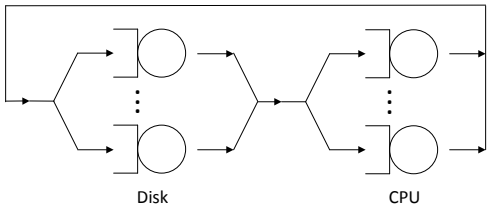


Figure 5: A queueing network

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 2: Notations used in the queueing model

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 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 4 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 [29], 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 [27] 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

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 3: Notations used in the buffer pool model

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 [20] for the "clock" algorithm that is used 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 [20] 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:

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

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

$$h_p = \frac{N_p}{S_p}. \tag{8}$$

The notations used in the above equations are illustrated in Table 3.

By Equation (7) and (8),

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

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

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

We have $F(0) = -B < 0$, and $F(+\infty) = \lim_{t\to+\infty} F(t) = \big(\sum_{p=1}^{P} S_p\big) - B > 0$, 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 [20]. 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 [20].

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} ← PredictHitRate(M_buf);
2  for 1 ≤ i ≤ n do
3  |   V_{disk,i} ← 0;
4  |   foreach partition p accessed by P_i do
5  |   |   V_{disk,i} ← V_{disk,i} + V_{disk,i,p}(1 − h_p);
6  |   end
7  end
8  {R_{k,i}}_{i=1}^{n} ← PredictResTime(M_queue, {V_{k,i}}_{i=1}^{n});
9  for 1 ≤ i ≤ n do
10 |   T_i ← 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.

[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.

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 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 results of the proposed approaches. We measure the prediction accuracy in terms of *mean relative error* (MRE), a metric used in [3, 7]. 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 time of the testing query i. We measured the additional overhead of the prediction approaches as well.

4.1 Experimental Setup

We evaluated our approaches with the TPC-H 10GB benchmark database. We used TPC-H workloads as well as workloads for micro-benchmarking purposes. 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.

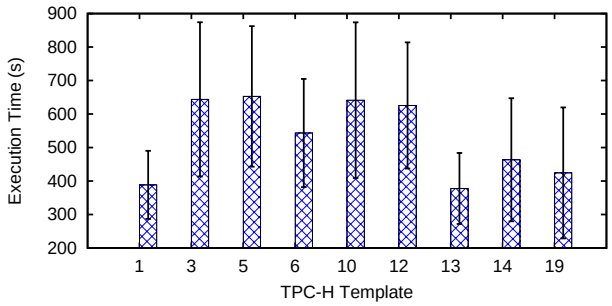
4.1.1 Workloads

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

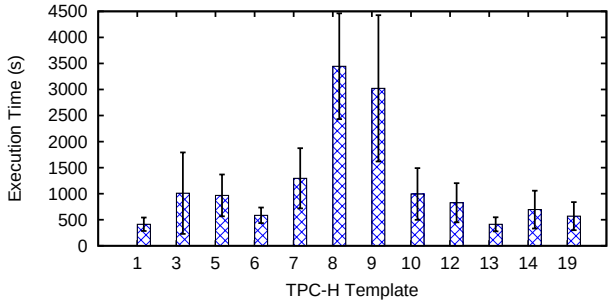
I. TPC-H workloads

• **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 [7]. For each MPL, we then generated mixes of TPC-H queries via Latin Hypercube Sampling (LHS) [3, 7]. 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 we created TPC-H1. In addition to the 9 templates there, 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

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 4: Actual values of PostgreSQL optimizer parameters

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

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 [34] 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 4 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 our following experiments, we set the sampling ratio to be 0.05 (i.e., the sample size was 5% of the database size).

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.

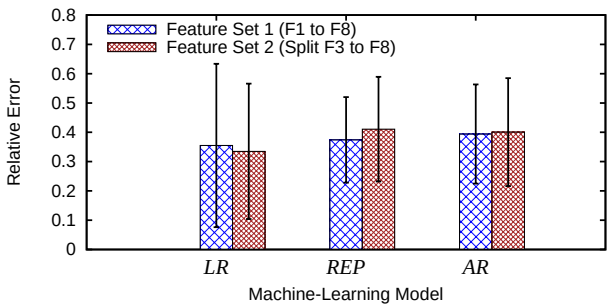
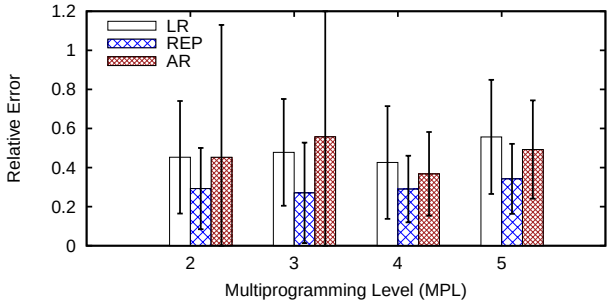


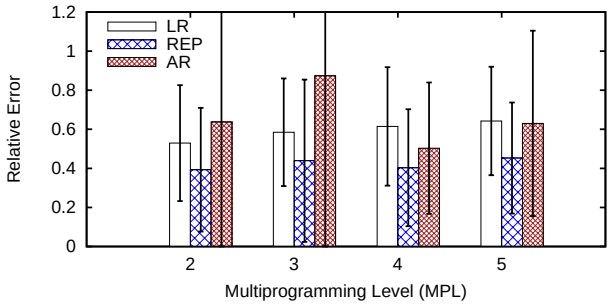
Figure 7: Prediction error on 5-way cross validation

Impact of Features. In Figure 7, we compare the average MRE and the standard deviation (shown as error bars) over the 16 scan types for different machine-learning models. Here, *LR*, *REP* and *AR* are linear regression, REP trees, and additive regression, respectively. For each machine-learning model, we tested two different sets of features. For the first set the features F1 to F8 (see Section 3.1.2) are used, and for the second set the features F3 to F8 are further split with respect to different scan types (i.e., SS, IS, and BIS).

As we can see, using more features does not help reduce the prediction error. In fact, the observed prediction errors when more features are used are even a little bit larger than their counterparts. One possible reason might be that the correlation between the features and the target variable does not change much or is even weaker when splitting F3 to F8. Therefore, in our following experiments we use the features F1 to F8 without further split.



(a) TPC-H1



(b) TPC-H2

Figure 8: Prediction errors on the TPC-H workloads for different machine-learning models

Impact of Machine-Learning Models. We tested the prediction accuracy of different machine-learning models over the two

TPC-H workloads. Figure 8 presents the results. We find that REP trees outperform linear regression across the four MPL's we tested, which implies the nonlinear correspondence between the features and the target variable c_{disk}. Moreover, it is a bit surprising that REP trees are also better than additive regression. One possible reason might be, although boosting is believed to be better than a single learner, this belief is based on the assumption that the testing data and training data are generated following the same distribution. In our case, this assumption does not hold. We collect training scans via LHS, but this distribution may not match the distribution of scans in the TPC-H workload. Therefore, in our following experiments we choose to use REP trees as the machine-learning model.

4.1.4 Settings for Analytic Models

The queueing model needs the calibrated c's (in Table 4) 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 [34]. 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 9 and 10 present the prediction errors over the two TPC-H workloads TPC-H1 and TPC-H2. On TPC-H1, the accuracy of the analytic-model based and the machine-learning based approach are close, both outperforming the baseline approach by reducing the error by 15% to 30% (Figure 9). This performance improvement may not be very exciting, regarding the simplicity of the baseline approach. However, we note here that this is because of the way the workload is generated rather than the problem of 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 10), while its two competitors retain close prediction accuracy as what is observed on TPC-H1. Nonetheless, it is important to include the baseline to show that sometimes it does surprisingly well, and

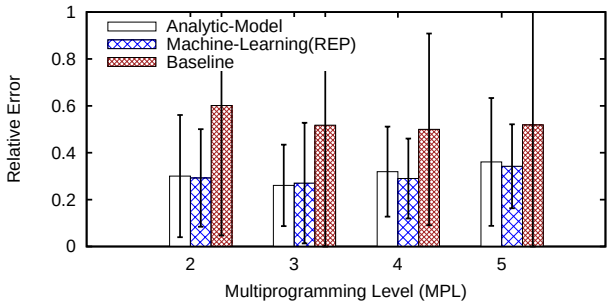


Figure 9: Prediction error on TPC-H1 for different approaches

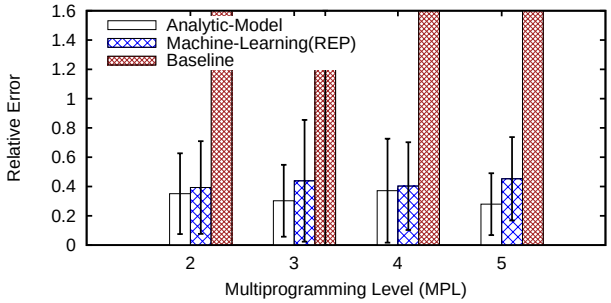


Figure 10: Prediction error on TPC-H2 for different approaches

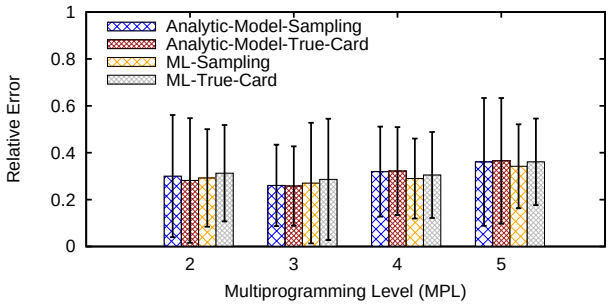
makes it challenging to improve substantially over that baseline. We also observe that, on TPC-H2, the analytic-model based approach slightly outperforms the machine-learning based approach by improving the prediction accuracy by about 10%.

We further compared the prediction accuracy by using the true cardinalities instead of using the refined ones via sampling. The purpose is to investigate the effectiveness of the proposed approaches if we were able to get perfect cost estimates. Figure 11 presents the results. As we can see, the prediction accuracy by using the refined cardinalities via sampling is quite close to that by using the true cardinalities.

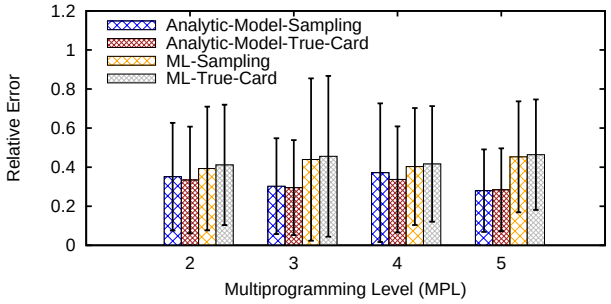
4.2.2 Results on Micro-Benchmarking Workloads

Since the TPC-H workloads were generated via LHS, they only covered a small fraction of the whole 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 12 to 14 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 12). 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 focuses 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



(a) TPC-H1



(b) TPC-H2

Figure 11: Comparison of prediction errors on the TPC-H workloads by using the true cardinalities v.s. the refined cardinalities via sampling

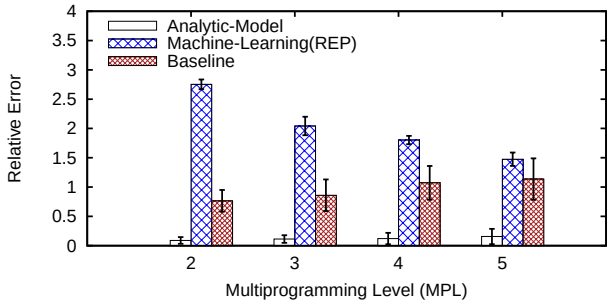


Figure 12: Prediction error on MB1 for different approaches

of query mixes.

The prediction errors of the baseline approach remain large on the workloads MB2 and MB3 (Figure 13 and 14). This is not surprising, since the query interactions in MB2 and MB3 are expected to be much more complicated and 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.

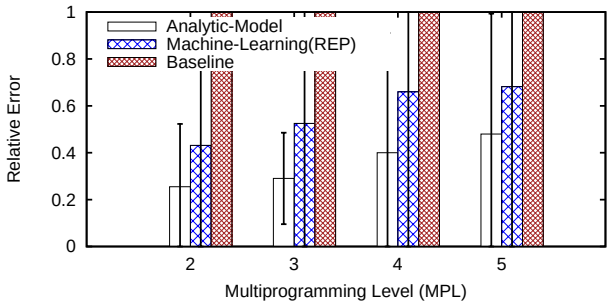


Figure 13: Prediction error on MB2 for different approaches

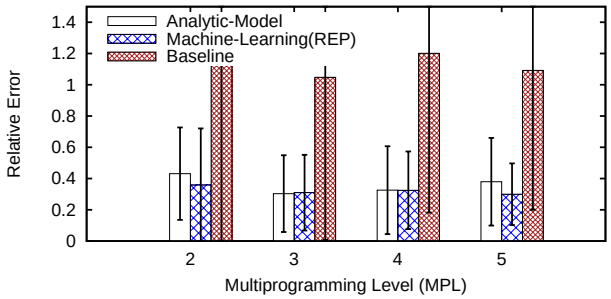


Figure 14: Prediction error on MB3 for different approaches

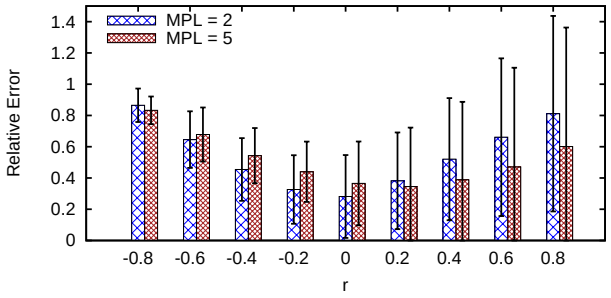
4.2.3 Sensitivity to Errors in Cardinality Estimates

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 then how sensitive the proposed approaches are to the potential errors presented in cardinality estimates.

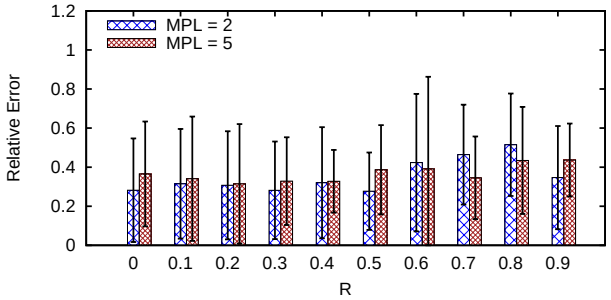
We investigated this question for the analytic-model based approach, which, as we have seen, outperformed its machine-learning counterpart on the workloads we tested. We studied this by feeding the optimizer with cardinalities generated by perturbing the *true* cardinalities. Specifically, consider an 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 $(-R, R)$.

Figure 15 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 15(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 15(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. Figure 16 further presents results on the TPC-H2 workload, which are similar to that observed on TPC-H1.

The robustness of the proposed approach to small or medium errors in cardinality estimates might be partially attributed to the

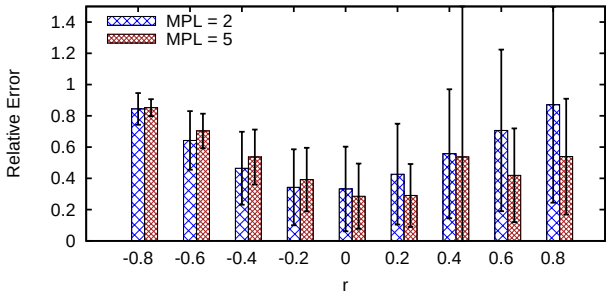


(a) Biased errors

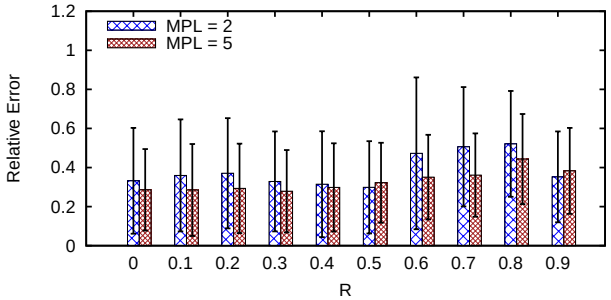


(b) Unbiased errors

Figure 15: Sensitivity of prediction accuracy on TPC-H1



(a) Biased errors



(b) Unbiased errors

Figure 16: Sensitivity of prediction accuracy on TPC-H2

robustness of the progressive predictor (Section 2.3) in determining the next pipeline to be finished. As long as every time it can make the right choice, the predicted transitions between consecutive pipeline combinations will be the same as that in the real execution, and therefore the prediction errors will intuitively be proportional to the errors in cardinality estimates. To verify this, we further conducted experiments to analyze the variation in the transitions of pipeline combinations with respect to cardinality estimation errors.

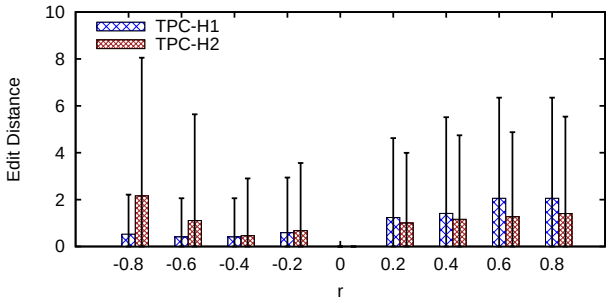
Specifically, we use the following encoding scheme to represent the execution of a query mix. As in Example 3 (Section 2.3), let P_{ij} be the jth pipeline of the ith query. We use the string "`(i,j)`" to encode P_{ij}. A pipeline combination is represented by sorting its pipelines $\{P_{ij}\}$ with respect to i then j. For example, the first combination in Figure 4 $\{P_{11}, P_{21}, P_{31}\}$ is encoded with the string "`(1,1)(2,1)(3,1)`". The execution of a query mix is then represented by cascading the pipeline combinations with "–". For instance, the execution of the query mixes in Figure 4 is encoded as "`(1,1)(2,1)(3,1)-(1,1)(2,2)(3,1)-(1,2)(2,2)(3,1)-(1,2)(2,3)(3,1)-(1,3)(2,3)(3,1)-(1,1)(2,3)(3,2)-(1,3)(3,2)-(1,3)`". We then measure the similarity of two executions with the *edit distance*[2] between their string representations.

Figure 17 presents the results with respect to biased errors. Here, the edit distance is between the predicted execution (i.e., transitions of pipeline combinations) by using the perturbed cardinalities and that by using the true cardinalities. As we can see, the average distance is usually below 2 when MPL is 2 and below 25 when MPL is 5. Since we used 5 characters to encode one pipeline, it implies that there is almost no change in the transitions of pipeline combinations when MPL is 2 and up to 5 changes when MPL is 5. Therefore, transition change is more likely to happen with higher MPL's. However, the number of transition changes is still small compared with the number of pipeline combinations during the execution (usually more than 20) when MPL is 5. This demonstrates certain kind of robustness of the proposed approach to cardinality errors. The intuition is that, while there might be errors in the predicted execution times for each pipeline in the combination, the predictive predictor only cares about the one with the *minimum* execution time. As long as the errors do not cause changes in the *ranking* of the pipelines with respect to their execution times, the predictor can still make the right decision. This is akin to the case of query optimization, in which the optimizer only cares about the ranking of the query plans based on their costs, and many times it can succeed in picking the right plan even if there might be significant errors in the cost estimates.

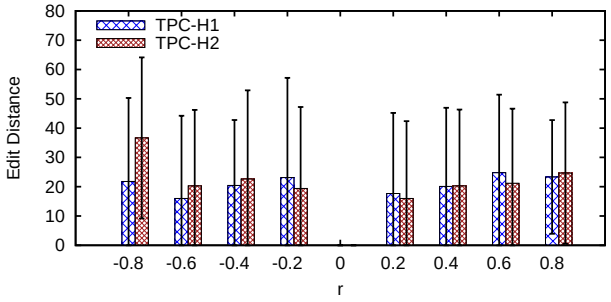
Figure 18 further presents the results with respect to unbiased errors. An interesting observation here is that the distances seem to be larger than that with respect to biased errors. This is because unbiased errors actually make it harder for the predictor to make the correct decision since the ranking of the pipelines based on their finish time is more sensitive to unbiased errors than biased errors. When the errors are biased, the predictions for the execution times of the pipelines will also be biased towards one direction, and hence it is more likely that the correct ranking is preserved. When the errors are unbiased, the predictor may underestimate the execution times for some pipelines while overestimate the others, making it easier to break the correct ranking. However, the overall prediction errors with respect to unbiased cardinality estimation errors could still be smaller, due to the cancelation of prediction errors for different operators in the query plan.

Of course, although edit distances provide some insights into the variation of pipeline combination transitions, it is still far from perfect in capturing all the differences. For instance, some pipeline combinations may be more important than the others for accurate predictions, and hence a mistake there will cause more severe impacts. Edit distances do not consider this effect. Moreover, as we

[2]http://en.wikipedia.org/wiki/Edit_distance

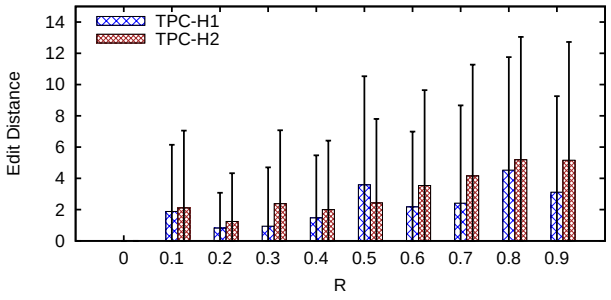


(a) MPL = 2

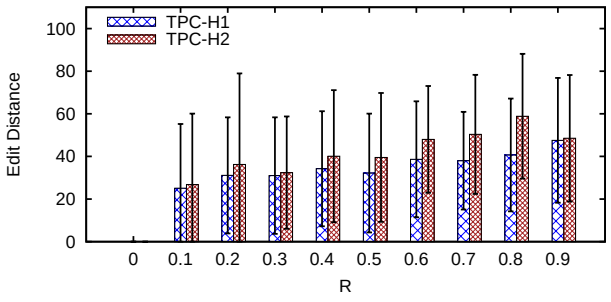


(b) MPL = 5

Figure 17: Edit distances with respect to biased errors



(a) MPL = 2



(b) MPL = 5

Figure 18: Edit distances with respect to unbiased errors

have seen, the relationship between the overall prediction accuracy and the cardinality estimation errors is quite complicated. Larger variation in the transitions of pipeline combinations does not necessarily mean larger overall prediction errors. Other factors, such as the cancelation of prediction errors on different operators, may also play an important role and cannot be ignored. A thorough theoretical study on all these issues is beyond the scope of the current paper, and we leave the formal sensitivity analysis, such as the worst-case and the average-case prediction accuracy, as one interesting direction for future exploration.

4.2.4 *Comparison with Previous Work on Queueing Networks*

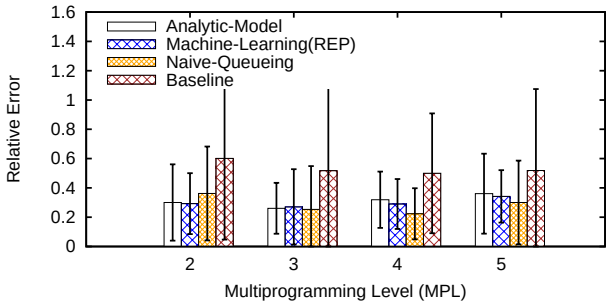
Queueing networks have been extensively used in computer system modeling, including database systems (e.g., [1, 21, 22, 28, 30, 31]). However, the focus in this work is quite different from ours. Previous work used queueing networks to predict macro performance metrics such as the throughput and mean response time for different workloads. Their goal, as pointed out by Sevcik [28], 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 [21, 22, 30, 31], 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.

To demonstrate this, we compared the prediction accuracy of our proposed approach with one based on a straightforward application of queueing models, which simply treats each query (instead of each pipeline) as a customer to the queueing network described in Section 3.2.1. We call this approach "naive queueing". Figure 19 compares the prediction accuracy of our proposed approaches with naive queueing. While naive queueing performs quite well on TPC-H1 (not very surprisingly since even the baseline can give reasonable predictions), its prediction accuracy is much worse on the other workloads, especially for TPC-H2, MB1, and MB2.

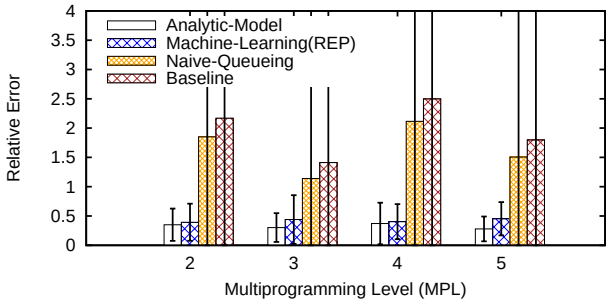
4.3 Additional Overhead

Both the machine-learning based and the analytic-model based approach need to calibrate the optimizer's cost model. As shown in [34], calibrating the c's is a one-time procedure and usually can be done within a couple of hours. The overhead of calibrating the 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 drastically reduced if the samples could be kept in memory [25].

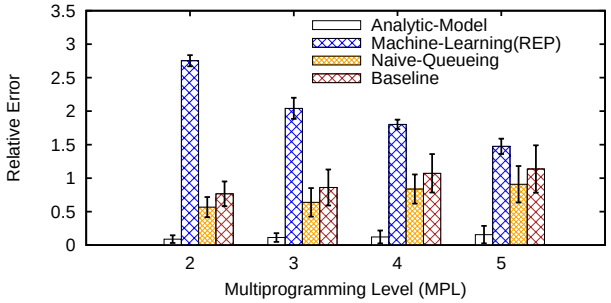
In addition to the overhead in calibrating the cost model, the machine-learning based approach needs to collect the training data. Although the training is offline, this overhead is not trivial. The time spent in the training stage depends on several factors, such as the number of sample scan mixes and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



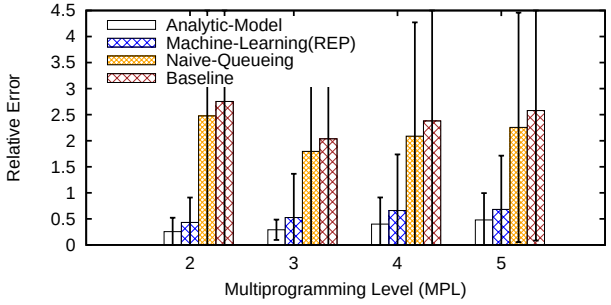
(a) TPC-H1



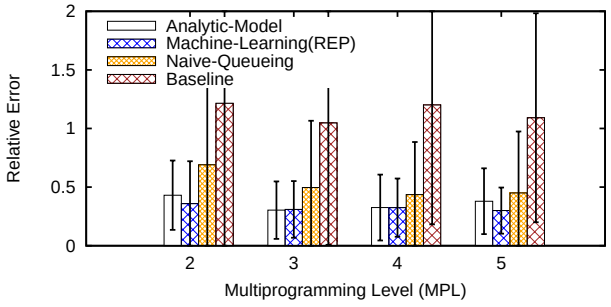
(b) TPC-H2



(c) MB1



(d) MB2



(e) MB3

Figure 19: Comparison with naive queueing

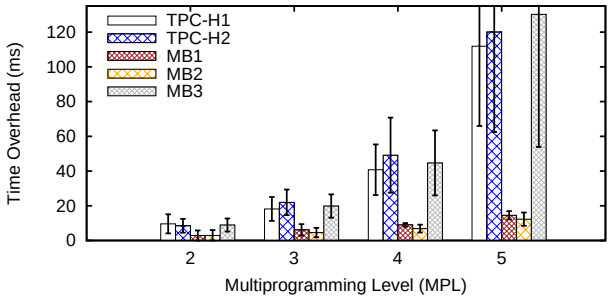


Figure 20: Runtime overhead in evaluating analytic models

Optimizer Parameter	Calibrated μ (ms)	Default
seq_page_cost (c_s)	5.06e-2	1.0
rand_page_cost (c_r)	9.07e-1	4.0
cpu_tuple_cost (c_t)	1.55e-4	0.01
cpu_index_tuple_cost (c_i)	1.23e-4	0.005
cpu_operator_cost (c_o)	9.22e-5	0.0025

Table 6: Calibrated cost units on the 8-core machine

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 20 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).

4.4 Discussion

The approaches proposed in this paper relies on the optimizer's cost model to provide reasonable cost estimates. Although we have used the framework in [34] to calibrate the cost model, it still contains some flaws. For example, in the current implementation of PostgreSQL, the cost model does not consider the heterogeneous resource usage at different stages of an operator. This may cause some inaccuracy in the cost estimates. For instance, the I/O cost per page in the building phase of hash-based joins might be longer than that predicted by the cost model, due to potential read/write interleaves if spilling occurs. In this case, the current approach might underestimate the execution time of a hash join operator. One way to fix these issues is to improve the cost model.

For example, in [23], the authors proposed a more accurate analytic model for the hybrid hash join algorithm, by further considering the read/write interleavings in the building phase. A thorough revision to the PostgreSQL's cost model, however, might require considerable development efforts and is beyond the scope of the current paper. Our goal here is just to see how effective the proposed approach is, based on the currently-used imperfect cost models. We believe that an improved cost model could further enhance our approach by delivering more accurate predictions, and we leave the development of a better cost model as an interesting future work.

4.5 Repeatability of the Results

We further tested the analytic-model based approach on a different machine configured with a quad-core (8 threads) Intel 2.40GHz CPU and 4GB of memory. Table 6 lists the calibrated values for the

cost units, and the parameter values of the buffer pool model are the same as that in Table 5. As shown in Figure 21, the prediction accuracy on this quad-core machine for the two TPC-H workloads and the three micro-benchmarking workloads is close to that observed on our previous dual-core machine.

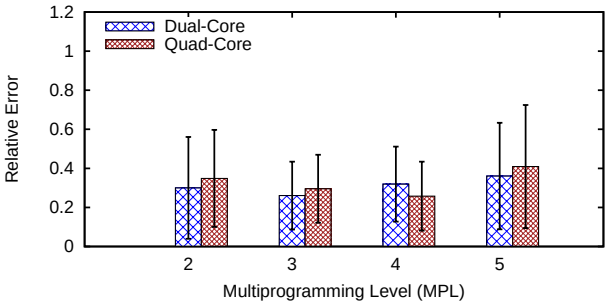
5. RELATED WORK

Predicting query execution time has recently gained significant interest in database research community [3, 4, 7, 9, 15, 34]. In [9], 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) [5] as the predictive model. A similar idea was proposed in [4], 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 [15], 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 [34] 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.

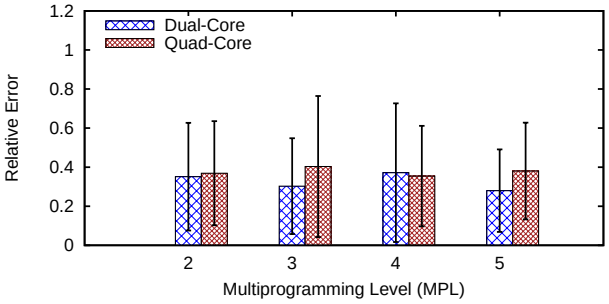
The problem of predicting concurrent query execution time was studied in [3] and [7]. In [3], the authors proposed an experiment-driven approach by sampling the space of possible query mixes and fitting statistical models to the observed execution time of these samples. Specifically, they used Gaussian processes as the particular statistical model. A similar idea was used in [7], 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.

One closely related line of studies is query progress indicators, which has been extensively studied in literature [6, 13, 16, 17, 18]. Progress indicators for a single query were first proposed simultaneously and independently in [6] and [17]. Recent work further improved their accuracy [16] and robustness [13]. Moreover, Luo et al. [18] investigated multi-query progress indicators. The key difference between query execution time prediction and query progress indicators is that queries are not allowed to run when the prediction needs to be made. As a result, runtime information such as the real input/output cardinality for each physical operator, which is important for the accuracy of progress indicators, is not available in execution time prediction.

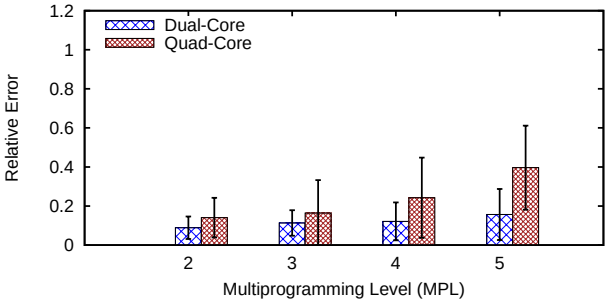
Concurrent query time prediction has also been leveraged in solutions to admission control [32] and query scheduling [2]. In [32], the authors proposed an admission control framework based on a linear regression model of expected query execution time that accounts for the mix of queries being executed. In [2], similar idea was used to develop a query scheduler by solving a linear programming problem. However, for the predictive model to work,



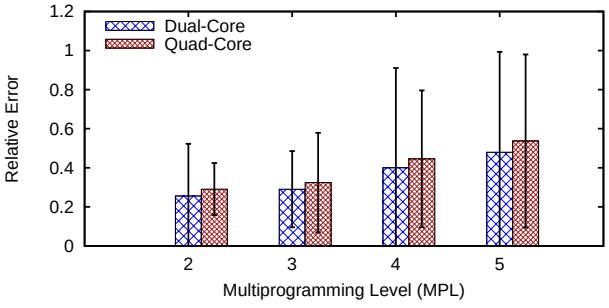
(a) TPC-H1



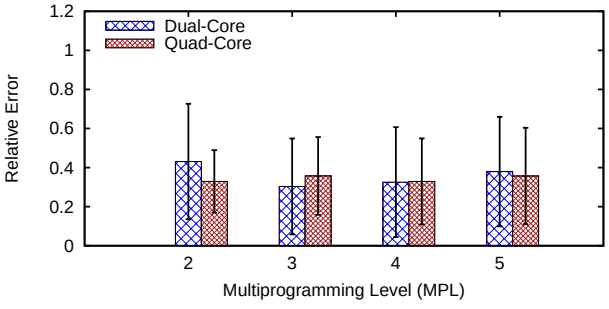
(b) TPC-H2



(c) MB1



(d) MB2



(e) MB3

Figure 21: Prediction accuracy on a multi-core machine

both work still assumed that the workload needs to be static. It is interesting future work to develop admission control and query scheduling frameworks for dynamic query workloads, based on the predictive models proposed in this paper.

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] E. J. Adams. Workload models for dbms performance evaluation. In *ACM Conference on Computer Science*, pages 185–195, 1985.

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

[3] 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.

[4] 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.

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

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

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

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

[9] 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.

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

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

[12] 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.

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

[14] 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.

[15] 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.

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

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

[18] G. Luo, J. F. Naughton, and P. S. Yu. Multi-query SQL progress indicators. In *EDBT*, 2006.

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

[20] 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.

[21] R. Osman, I. Awan, and M. E. Woodward. Queueing networks for the performance evaluation of database designs. In *UKPEW*, pages 172–183, 2008.

[22] 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.

[23] J. M. Patel, M. J. Carey, and M. K. Vernon. Accurate modeling of the hybrid hash join algorithm. In *SIGMETRICS*, pages 56–66, 1994.

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

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

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

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

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

[29] R. Suri, S. Sahu, and M. Vernon. Approximate mean value analysis for closed queueing networks with multiple-server stations. In *IERC*, 2007.

[30] N. Tomov, E. W. Dempster, M. H. Williams, A. Burger, H. Taylor, P. J. B. King, and P. Broughton. Some results from a new technique for response time estimation in parallel dbms. In *HPCN Europe*, pages 713–721, 1999.

[31] 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.

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

[33] 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.

[34] 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*, 2013.