Contender: A Resource Modeling Approach for Concurrent Query Performance Prediction

Jennie Duggan
Brown University
jennie@cs.brown.edu

Olga Papaemmanouil
Brandeis University
olga@cs.brandeis.edu

Ugur Cetintemel
Brown University
ugur@cs.brown.edu

Eli Upfal
Brown University
eli@cs.brown.edu

ABSTRACT

Predicting query performance under concurrency is a difficult task that has many applications in capacity planning, cloud computing, and batch scheduling. We introduce *Contender*, a new resource-modeling approach for predicting the concurrent query performance of analytical workloads. Contender's unique feature is that it can generate effective predictions for both static as well as ad-hoc or dynamic workloads with low training requirements. These characteristics make Contender a practical solution for real-world deployment.

Contender relies on models of hardware resource contention to predict concurrent query performance. It introduces two key metrics, *Concurrent Query Intensity* (*CQI*) and *Query Sensitivity* (*QS*), to characterize the impact of resource contention on query interactions. CQI models how aggressively concurrent queries will use the shared resources. QS defines how a query's performance changes as a function of the scarcity of resources. Contender integrates these two metrics to effectively estimate a query's concurrent execution latency using only *linear time* sampling of the query mixes.

Contender learns from sample query executions (based on known query templates) and uses query plan characteristics to generate latency estimates for previously unseen templates. Our experimental results, obtained from PostgreSQL/TPC-DS, show that Contender's predictions have an error of 19% for known templates and 25% for new templates, which is competitive with the state-of-the-art while requiring considerably less training time.

1. INTRODUCTION

Concurrent query execution offers numerous benefits for database applications. It can decrease the time required to execute analytical workloads [1, 2] and lead to better use of hardware resources by exploiting parallelism. At the same time, concurrent execution raises numerous challenges, including reasoning about how interleaving queries will affect one another's rate of progress. As multiple queries compete for hardware resources, their interactions may be positive, neutral, or negative [3]. For example, a positive interaction may occur if two queries share a table scan; one may prefetch data for the other and they both enjoy a modest speedup. In contrast, if two queries access disjoint data and are I/O-bound, they will slow each other down.

Accurate *concurrent query performance prediction* (*CQPP*) stands to benefit a variety of applications. This knowledge would allow system administrators to make better scheduling decisions for large query batches, reducing the completion time of individual queries and that of the entire batch [4]. With CQPP, cloud-based database applications would be able to make more informed resource provisioning and query-to-server assignment plans [5, 6]. High quality predictions would also pave the way for more refined query progress indicators by analyzing in real time how resource availability affects a query's estimated completion time. Moreover, accurate CQPP could enable query optimizers to create concurrency-aware execution plans.

Because of such important applications, there has been much recent work on CQPP for both transactional [7] and analytical workloads [1, 8]. We focus primarily on CQPP for *analytical* queries (OLAP), for which existing techniques [1, 8] suffer from two main limitations. First, they require that performance samples of different query mixes are collected before predictions may be produced. Hence, their prediction models are valid for only known query templates. Second, the sampling requirements for existing approaches grow *polynomially* in proportion to the complexity of their workloads, limiting their viability in real-world deployments.

In this paper, we propose a general CQPP framework, called *Contender*, that is practical to use for static as well as *dynamic* and *ad-hoc* workloads. Dynamic workloads are present in many exploration-oriented database applications including science, engineering, and business. Contender relies on models of *resource contention* for analytical, *concurrently* executing queries. It leverages *both* resource usage statistics and semantic information from query execution plans to model a query's performance as it changes due to concurrent queries that are competing for the same resources. Specifically, our query performance predictions are based on modeling the query's I/O bandwidth and how it is affected by memory availability. Scarcity of I/O bandwidth and memory are the dominant sources of slowdown for analytical queries executing under concurrency; these queries access very large data sets while filling the available memory with intermediate results, further limiting the available resources [9].

Contender first models how query performance varies under different resource availability scenarios. Specifically, for each query for which we want to predict its performance (the *primary* query), we create its performance range, which we call the *continuum*. We define the lower bound of this continuum as the query's execution time in isolation, which gives its minimum execution latency. The upper bound of this range is provided by limiting the availability of I/O bandwidth and memory to simulate the worst-case scenario for a query executing under concurrency.

In the next steps, Contender quantifies the resource availability during the primary query's execution to predict where in the performance range (continuum) the primary's latency will reside. The framework leverages query plan information of the *concur-*

rent queries - those running simultaneously with a primary - to estimate the conditions under which the primary query will be executed. Specifically, we propose the *Concurrent Query Intensity (CQI)* metric to quantify the I/O usage of concurrent queries and use this metric to estimate the availability of I/O bandwidth for the primary query. A unique feature of the CQI metric is that it models the percentage of time concurrent queries compete directly with the primary for shared resources. We show that CQI is highly correlated with concurrent query latency. Given the CQI value, Contender builds a performance prediction model (*Query Sensitivity (QS)*) to predict the primary query's latency.

A unique feature of Contender is that it can make performance predictions for new templates *without* requiring a priori models. Instead, our framework first assembles a set of *QS reference models* for the templates it has seen already. Next it learns the predictive models for new templates based on these reference models. An interesting discovery is that new templates with similar behavior *in isolation* (e.g., latency, I/O-to-CPU ratios), have similar performance prediction models (e.g., QS models). This approach eliminates the need to collect samples on how new queries interact with the existing workload, which is the main bottleneck of previous work [1, 8]. Therefore, Contender dramatically simplifies the process of supporting unpredictable or evolving query workloads.

Lastly, Contender reduces training requirements as it eliminates the need to collect samples of query performance under varying resource availability conditions. We show that query plan characteristics, paired with resource profiling on a template's isolated performance, are effective in predicting how a query reacts to differing levels of hardware availability by comparing it to other, similar queries. This permits us to estimate the worst-case performance scenario with reduced sampling of new templates. Therefore, Contender significantly lowers training time for new queries compared with existing work [1, 8]. We show that our sampling requirements can be reduced from polynomial to linear, and with further restrictions, to even *constant* time.

For completeness, our work also includes a study of CQPP using well-known machine learning techniques. These algorithms effectively predict the performance of queries executed in isolation (with a prediction error of around 25%) [10, 11]. Our results demonstrate that these models are poorly fitted to the complex case of concurrent query executions and motivate the need for more advanced techniques.

Our main contributions can be summarized as follows:

- We evaluate established machine learning approaches for CQPP and establish the need for more advanced techniques.
- We introduce novel resource-contention metrics to predict how analytical queries behave under concurrency: *Concurrent Query Intensity* quantifies the resources to which a query accesses when executing with others, and *Query Sensitivity* models how a query's performance varies as a function of resource availability.
- We leverage these metrics to predict latency of unseen templates using linear-time sampling of query executions.
- We further generalize our approach by predicting worst-case scenario performance of templates, reducing our sampling overhead to constant time.

Our paper is organized as follows. In Section 2, we briefly introduce the characteristics of the analytical workload used for evaluating our performance predictions. In Section 3, we evaluate sophisticated machine learning techniques to predict new template performance. Section 4 examines the relationship between I/O contention and query performance. Next, we describe our approach to building models for CQPP. We present our experimental results in Section 6, survey related work in Section 7, and conclude.

Template	1	2	3	4	5
1		X			
2					X
3	X				
4			X		
5				X	

Figure 1: Example of 2-D Latin Hypercube Sampling

2. ANALYTICAL WORKLOADS

The following sections include a number of experimental observations and results. This section describes the assumptions and characteristics of our workload. A more detailed discussion is available in Section 6.

Workload Characteristics In this paper, we target analytical workloads. We assume that all queries are derived from a set of known query classes (e.g., TPC-DS query templates) and that they are primarily I/O-intensive. Thus, we focus on modeling I/O and memory availability for our latency predictions. Our models do not directly address CPU utilization because in modern architectures the number of cores per host usually exceeds the concurrency level. Hence, this resource is not a source of contention for queries executing under concurrency.

The workload consists of 25 templates of moderate running time with a latency range of 130-1000 seconds when executed in isolation. This decision is based on the observation that long running queries are poorly suited for running under concurrency [12], and the shortest queries are not useful to model due to their limited duration. We created our workload by recording the latency of each template when run in isolation with a cold cache. The workload consists of the template with the median latency and the 24 others immediately less than it.

Contender builds a prediction model for each district query template. Query templates (or classes/types) are parameterized SQL statements, and examples of the same template share a structure, differing only in the predicates they use. For example, the template:

```
SELECT * FROM STORE_SALES
WHERE ITEM_KEY=[IKEY];
```

is used repeatedly to look up different instances of sales by an item's key. We note that recurring use of query templates is a common case for analytical workloads [13]. For the duration of this paper, we use the terms query and template interchangeably.

Sampling Concurrent Query Mixes To build our predictors, we learn from samples of concurrent query mixes. Here, queries are drawn from a set of n templates. Concurrent mixes where k queries are executing at the same time are said to have a multiprogramming level (MPL) of k[12]. When evaluating concurrent mixes for each MPL k, with n distinct templates, there exist n-choose-k with replacement combinations. This results in $\binom{n+k-1}{k} = \frac{(n+k-1)!}{k!(n-1)!}$ distinct mixes. For pairwise interactions (MPL 2) we sampled all pairs to avoid bias in our evaluation. As the MPL increases, the number of distinct concurrent mixes grows exponentially. For instance, at MPL 5 our 25 templates result in 118,755 unique mixes. Clearly, evaluating this space exhaustively is prohibitively expensive.

Therefore, for MPLs higher than 2 we used Latin Hypercube Sampling (LHS) to select mixes as in [3]. This technique uniformly

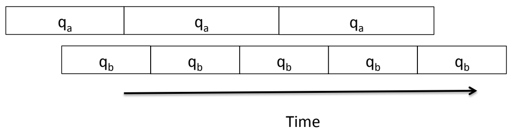


Figure 2: An example of steady state query mixes, q_a running with q_b at MPL 2

distributes the samples over the query mix space. The sampling is done by creating a hypercube with the same dimensionality as the MPL, k, and each dimension value corresponds to a distinct query template. The experiment selects samples such that every value on each dimension is intersected exactly once. We evaluated four disjoint LHS samples for MPLs 3-5 over our 25 templates.

An example of LHS at MPL 2 is shown in Figure 1. Here, each template appears in up to two query mixes. For instance, Template 1 is evaluated with T2 and T3. Therefore, each LHS run produces n mixes, where n is the number of unique query templates in the sampling workload.

In each experiment we hold the query mix constant using a technique we call *steady state*, as demonstrated in Figure 2. We create one stream for each template represented in a mix and introduce new examples of a template when prior ones end. This is analogous to the TPC-DS throughput test, but simplified to focus on CQPP for individual mixes. We continue the experiment until each stream has collected at least n samples (for our trials $n = 5$). We omit the first and last few queries to insure that conditions are consistent for the duration of each experiment. Steady state execution captures how each template is affected by a mix when concurrent queries start at different time offsets.

Prediction Quality Metrics We assessed each model's prediction quality using mean relative error (MRE):

$$MRE = \frac{1}{n} \sum_{i=1}^{n} \frac{|observed_i - predicted_i|}{observed_i} \qquad (1)$$

This normalized metric denotes the high-level impact of each error. MRE is the standard for query performance prediction research [3, 7, 8, 11].

Finally, unless otherwise specified, we conduct our evaluation using k-fold cross validation ($k = 5$). Hence, the data is subdivided into k equally-sized pieces, and the system trains on $k - 1$ folds, evaluating its models on the remaining data. We cycle through the folds, creating predictions for each one, and average all predictions to determine overall model quality.

3. EXTENDING EXISTING LEARNING TECHNIQUES

In this section we discuss well-known machine learning techniques and study their effectiveness in addressing the CQPP problem. Specifically, we tried to leverage existing techniques for predicting performance of isolated queries [10, 11] by adapting them to concurrent workloads. These approaches rely on the query execution plans (QEPs) of individual queries to obtain the steps that a query executes. These steps are then provided as a feature set to statistical machine learning techniques such as Kernel Canonical Correlation Analysis (KCCA) and Support Vector Machines (SVMs) to model query performance. Next we describe our attempts to adapt these techniques to CQPP and explain their ineffectiveness for predictions of concurrent query performance.

Feature Set Extension Our first step was to adapt the models in [10, 11] for isolated query latency prediction to include features from the concurrent executions. We begin with the query plan features: our feature vector includes a list of distinct QEP nodes for *all* of the queries in our training set as opposed to a single query. This defines a global feature space to describe all concurrent queries in our workload. For each unique execution step (e.g., table scan, index scan, join) we record i) the number of times it occurs in a plan, and ii) the summed cardinality estimate of its instances. For example, if a query plan has two nested loop joins, one with a cardinality estimate of 10,000 rows and the second with an estimate of 1,000, our nested loop join feature would be (2, 11,000). If there are n potential steps that a query may execute, we describe a given query with $2n$ features, which we call the *primary features*.

Our next goal was to add features to capture interactions among queries. Sequential table scans are an ideal candidate for analytical workloads; they are likely to cause reuse among concurrently executing queries or resource contention if there is no intersection in the query plans of the current batch. To achieve this, we treat sequential scans on different tables as distinct features. Therefore, we extended the feature vector of a query to include one feature per table in our schema. Like our prior features, the table-based ones each had a *(scan count, summed row estimate)* pair.

To predict the performance of a primary, we also model the impact of concurrent queries using their QEPs. We sum up the features of each concurrent template and create a second $2n$ vector of features. We then concatenate this vector with the primary features to create a single feature set per prediction. This brings our total number of features to $4n$, where n is the number of distinct QEP steps. In our experiments, we had 168 features for 42 QEP steps, which resulted in very complex models.

ML Prediction Techniques As in [10, 11], we evaluated two machine learning techniques to created predictions for new templates based on the features described above. In the first technique, KCCA, we create predictions by identifying similar examples in the training data. In the second technique, SVM, we learn by classifying queries to latency labels. For both cases, we work at the granularity of the query plan and used well-known implementations of each technique [14, 15].

To train KCCA, we create one feature vector per query and a corresponding performance vector. Our approach applies a Gaussian kernels to compare each input to the other training examples. If we have N queries, this produces two NxN matrices, one for each feature and performance vector. KCCA then solves the eigenvector problem, producing two maximally correlated projections based on the feature and performance space matrices. These projections take into account both QEP characteristics and how they compare to correlations in performance. To create a new prediction, we apply a set of basis vectors learned from the eigenvector problem. This projects new feature sets into feature space. We take our k nearest neighbors (k=3) and average their latency each prediction. Our implementation uses the Kernlab [15] package for R [16].

For SVM, we take as input a feature vector for each training query and and use their latency as a label. SVM creates a series of vectors to subdivide feature space by coarse-grained labels. When a new query arrives, the model places it in global feature space and derives the corresponding label for that location. SVM returns this label for its latency estimate. We use the LibSVM [14] implementation of this approach. Both learners use k-fold cross validation (k=6) to tune their models.

Prediction Accuracy We found moderate success with this approach for static workloads at MPL 2 (i.e., with the same templates in the training and test set but differing concurrent mixes). For this experiment, we trained on 250 query mixes and tested on 75 mixes, resulting in a 3.3:1 ratio of training to test data. Each template was proportionally distributed between the training and test set. We

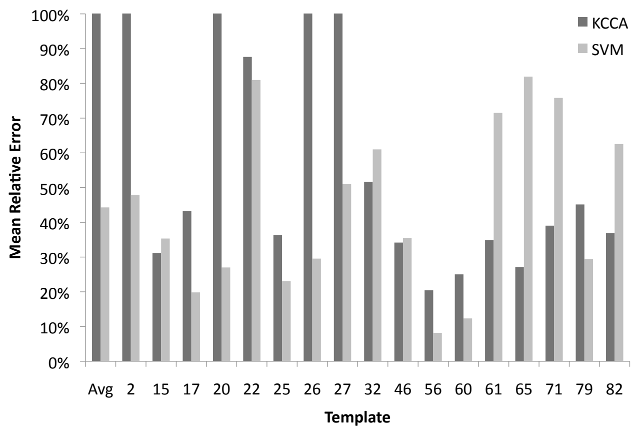


Figure 3: Relative error for predictions at MPL 2 using machine learning techniques on new templates.

found that we could predict latency within 32% of the correct value with KCCA on average and within 21% for SVM. Their predictions are competitive with the work [8], however they are more complex to produce and time-intensive to train.

We also evaluated these approaches on dynamic workloads, i.e., on "new" query templates not present in the training phase. For this study we reduced our workload from 25 to 17 templates, omitting ones having features that do not appear in any other template. These statistical approaches require significant training sets, so for these experiments we learn on 16 templates and test on the remaining one from our workload. As shown in Figure 3, we found that *neither* technique was very good at estimating latency for previously unseen templates.

KCCA was very sensitive to changes in the QEP steps used because it takes the Euclidean distance between the new template and its n nearest training examples. When a new template is far in feature space from all of the training examples, its predictions are of lower quality. Likewise, SVM over-fits the templates that it has rather than learning interactions between QEP nodes. This is problematic because execution steps that are salient in one QEP may be less important or absent in others. For example, templates 82 and 22 share a scan on the inventory fact table, unlike all of the remaining templates. In KCCA, they are regarded as similar despite differing significantly in the rest of their structure. Our prediction quality for both is very low. Likewise several of our templates use a hash aggregate plan node, including 15, 26 and 27, but for only template 22 it is a bottleneck. In SVMs, this behavior is not well-captured by broad classification. Furthermore, both techniques perform well on queries where they have one or more templates that are close in plan structure (e.g., templates 56 and 60).

Conclusions In this context, we concluded that the machine learning techniques applied poorly fit the complex interactions presented by CQPP. On one hand, such solutions work proportionally to the quality of their training data. The QEP feature space, however, is sparsely populated in practice and this results in a dearth of relevant examples for many templates. Furthermore, the set of features varies across database engines and hence, solutions based on QEP features are not platform independent.

Finally, although accurately modeling individual QEP step interactions (i.e., a table scan running with a join) could improve our predictions, such models expand our prediction space exponentially. An accurate prediction framework would model both the execution time of each execution step and how it is affected by others. One important challenge we faced was identifying which QEP operators/steps affect one another as they have varying levels of overlap. Therefore, in the above study we had difficulty collecting samples, building a robust model, and making predictions within a reasonable amount of time. To address the challenge of CQPP, we propose an alternative novel approach that casts the problem in terms of resource contention rather than interactions of individual QEP steps.

4. MODELING RESOURCE ACCESS UNDER CONCURRENCY

This section describes how we model I/O contention between a primary query and concurrent queries in a mix. Specifically, we introduce the *Concurrent Query Intensity* (CQI) metric, which measures the I/O usage of concurrent queries and quantifies the bandwidth available to the primary for a given mix. Contender uses CQI to describe the environment in which the primary query is executing, providing input for the models in Section 5.5. In the following sections we provide evidence that CQI is highly correlated with query latency for known templates, and therefore an effective proxy for predicting query execution times.

In our models we focus on interactions at the level of query templates rather than individual query execution plans (QEPs). The reasons for this are twofold. First, as discussed in Section 3, modeling interactions at the QEP level is not very effective, due to sparsity in the sampling of feature space. Second, we observed that on average our templates exhibited a standard deviation in latency of 6% when executing in isolation, which is a manageable level of uncertainty. Were we to find higher variance, we could use the techniques in [1] to divide our templates into subclasses.

4.1 Concurrent Query Intensity

Given a primary query t executing concurrently with a set of concurrent queries, $C = \{c_1, ..., c_n\}$, Contender uses the CQI metric to model how aggressively concurrent queries will use the underlying resources. Specifically, we focus on how to accurately model the usage of I/O by the concurrent queries. This is important because the I/O bottleneck is the main source of analytic query slowdown [8]. This metric first examines how much of the I/O bandwidth each concurrent query uses when it has no contention. Next, it estimates the impact of shared scans between the primary and concurrent queries. Finally, CQI evaluates how shared I/O among the non-primary concurrent queries may further reduce I/O scarcity.

Baseline I/O Usage First, we examine the behavior of each concurrent template, c_i, by estimating its I/O requirements. In particular, we measure the percent of its isolated execution time that uses I/O, p_{c_i}. This approach deliberately does not differentiate between sequential and random access. By measuring only the time expended on the I/O bus, all of our comparisons share the same units. We calculate p_{c_i} by recording (using Linux's `procfs`) the time elapsed executing I/O while c_i is running in isolation, and divide by the query's latency.

Positive Primary Interactions In the next step, Contender estimates the positive impact of interactions between concurrent query c_i and the primary. One way for c_i to create less interference with the primary query is for it to share I/O work. Sharing needs to be large enough for the interaction to make an impact on end-to-end latency. We expected the bulk of our positive interactions to occur from shared fact table scans, and empirically confirmed this assumption in our experiments. The reason for these positive interactions is that fact tables are the largest source of I/O for analytical

Symbol	Definition
p_t	% of exec. time spent on I/O when t runs in isolation
$g_{f,c}$	Boolean for if c and primary scans f
s_f	Time required to scan table f in isolation
ω_c	I/O time shared between concurrent query c and primary
$z_{f,c}$	Boolean for if c scans f and primary does not
h_f	Number of concurrent queries scanning table f
τ_c	I/O time in which concurrent query c executes shared scans with other non-primaries
r_c	I/O time concurrent query c spends competing with primary
$r_{t,m}$	CQI for primary t in mix m
l_{min_t}	Minimum (isolated) latency of template t
l_{max_t}	Maximum (spoiler) latency for template t
$l_{t,m}$	Observed latency of template t when run in mix m

Table 1: Notation for a query t, table scan f, and concurrent query c.

queries: their data are often requested and hence cached in shared buffers and reused by concurrently running queries.

Based on this observation, we quantify for each concurrent query how it interacts with the primary by estimating its time spent on shared I/Os. To this end, we first determine the fact tables scans shared by the primary, t, and each concurrent query, c_i as:

$$g_{f,c_i} = \begin{cases} 1 & \text{if both template } c_i \text{ and primary scan table } f \\ 0 & \text{otherwise} \end{cases}$$

Then, we estimate the time the concurrent template c_i will not require the I/O bus *exclusively*, ω_{c_i}. This represents the I/O time spent on I/O requests shared with the primary and is:

$$\omega_{c_i} = \sum_{f=1}^{n} g_{f,c_i} \times s_f \tag{2}$$

where s_f is the time required to perform a table scan on the fact table f. We empirically evaluate the time of each table scan by executing a query consisting of only the sequential scan. The above formula sums up the estimated time required for each shared fact table scan.

Concurrent Query Interactions In addition to determining the I/O that is likely to be shared between the primary and its concurrent queries, we take into account the work that will be shared *among* concurrent queries. For example, if the primary is executed with queries a and b, this metric estimates the savings in I/O time for scans that are shared by a and b.

We first determine the table scans that are shared between c_i and other non-primary queries with:

$$z_{f,c_i} = \begin{cases} 1 & \text{if } c_i \text{ scans } f\text{, the primary does not and } h_f > 1 \\ 0 & \text{otherwise} \end{cases}$$

where h_f counts the number of concurrent queries sharing a scan of table f. Because we are only interested in shared scans, we add the limit that h_f must be greater than one. We also require that fact table f must not appear in the primary to avoid double counting. Our model assumes that concurrent templates will equally split the cost of shared sequential scans. We calculate the reduction in I/O due to shared scans for each concurrent query c_i as:

$$\tau_{c_i} = \sum_{f=1}^{n} z_{f,c_i} \times (1 - \frac{1}{h_f}) \times s_f \tag{3}$$

	Baseline I/O	Positive I/O	CQI
MPL 2-5	25.4%	20.4%	20.2%

Table 2: Mean relative error for CQI-based latency prediction.

Concurrent I/O Requirements Given the I/O requirement reductions owing to positive interactions among the primary and its concurrent queries, we estimate the I/O requirements of a concurrent query c_i as:

$$r_{c_i} = (l_{min_{c_i}} \times p_{c_i} - \omega_{c_i} - \tau_{c_i})/l_{min_{c_i}} \tag{4}$$

The above equation estimates the time c_i spends directly competing for I/O bandwidth with the primary as follows. It evaluates the total I/O time by multiplying the latency of c_i executing in isolation, $l_{min_{c_i}}$, by p_{c_i}, the percentage the query spend executing I/Os. We then subtract the I/O savings from its positive interactions with the primary, ω_{c_i}, and other concurrent queries, τ_{c_i}. We now have the percentage of c_i's "fair share" of hardware resources it is likely to use. Higher r_{c_i} values indicate the concurrent query will spend more time competing with the primary for the I/O bus, delaying the primary's progress. A lower r_{c_i} indicates that a concurrent query will create little contention for the primary.

Given a query mix m where m is comprised of the primary t and concurrent queries, $C = \{c_1, ..., c_n\}$, we define the *Concurrent Query Intensity* (*CQI*), quantifying the average percentage of I/O time during which each concurrent query competes for I/O, r_{c_i}:

$$r_{t,m} = \frac{1}{n} \sum_{i=1}^{n} r_{c_i} \tag{5}$$

We truncate all negative I/O estimates to zero. This occurs when queries have negligible I/O expectations outside of their shared scans. We use this estimate as the independent variable for our predictions of the primary's continuum point, $c_{t,m}$ in Equation 7.

Latency Predictions For Known Templates We now evaluate the CQI's utility in predicting query latency for known templates. We propose a linear relationship between the two metrics, and tested this hypothesis by generating one prediction model per template. We experimented with MPLs of 2-5, using k-fold cross validation for each template t. Hence, this evaluation predicted latency for $1/k$th of the sampled concurrent mixes at a time after training on the remaining examples. For each training mix, we collect the primary's average execution latency and CQI. Taken together, the training pairs produce a linear regression model with the primary's latency as the dependent variable, derived from the CQI figure in Equation 5. The prediction errors for MPLs 2-5 are displayed in Table 2.

We compare the accuracy of the linear model using CQI as its independent variable with models derived from subsets of CQI's three components. The first variant (*Baseline I/O*), predicts the query latency based only on the average percent of I/O time used by the concurrent queries when run in isolation (i.e., the average of the p_{c_i} values). The hypothesis here is that if we knew how much of the I/O bandwidth each concurrent query would uses to complete its work, we can average it out to determine the amount of "slack" available. This approach has moderate accuracy with a mean error rate of 25.4%.

The second variant (*Positive I/O*), predicts the primary's latency based on the baseline I/O requirements *and* the positive interactions of the concurrent queries with the primary. Specifically, we evaluated whether subtracting from the baseline I/O time (i.e., p_{c_i} values) the I/O time spent on shared scans between the concurrent queries and primary (i.e., w_{c_i} values) improves our estimates. This metric indeed offered improvements in our accuracy, compared with the Baseline I/O, bringing our mean relative error down to 20.4%.

Lastly, the CQI metric, which takes into account the positive interactions among the concurrent queries (i.e., t_{c_i} values) further refines our estimates. This results indicate that the CQI metric is a good predictor for query's execution time. The prediction improvement offered by CQI over the Positive I/O metric is relative small, implying that savings among the concurrent queries only reduces their duration and not that of the primary. In this work, we opt to use the CQI metric for our predictions since it takes into account all sources of query interaction: it is the only metric that considers the interactions among non-primary queries and in addition to interactions with the primary. Hence, it provides a more holistic metric than the Positive I/O alternative.

5. PREDICTIVE MODELS FOR QUERY PERFORMANCE

This section describes the steps Contender completes to learn the relationship between resource contention and query performance for static, known queries as well as new, previously unseen templates. First, we generate a latency range for the primary based on the minimum and maximum I/O and memory contention. We refer to this range as the primary's *performance continuum*. Then, we introduce our *Query Sensitivity* (QS) framework, a linear model that predicts the performance of a query (i.e., where its latency lies on the continuum) based on the resource contention metric, CQI, introduce in Section 4.

To handle ad-hoc templates, Contender first builds a set of reference QS models for the known templates available in its training workload. After that, it learns the predictive QS model for unknown templates based on the reference models. Our approach is discussed in Section 5.2. Finally, we analyze the sampling requirements of our framework and demonstrate how its sampling time can be reduced, from linear to constant time, by predicting the worst-case performance of a new template. We do so by using a mix of semantic and empirical modeling of individual templates.

5.1 Performance Continuum for Query Templates

Contender creates a continuum that establishes a range for the latency of the primary query template. To make new templates comparable to known ones, we normalize the query's latency as a percentage of its continuum and we estimate the upper and lower performance by simulating the best and worst resource contention scenario. By quantifying where a template's performance is on the continuum for individual mixes, we more generally learn how the query responds to contention.

We set the lower bound of our continuum to be the latency of the primary query t when running in isolation, l_{min_t}. This metric captures the best-case scenario, when the query makes unimpeded progress. For each template t and a given MPL, we use a *spoiler*[1] to calculate its continuum's upper bound, l_{max_t} (*spoiler latency*). The spoiler charts how a template's latency increases as contention for I/O bandwidth *and* memory grows. The spoiler simulates the highest-stress scenario for the primary query at MPL n. It allocates (1-1/n)% of the RAM, pinning it in memory. The spoiler circularly reads $n-1$ large files to continuously issue I/O requests.

Thus, if $l_{t,m}$ is the query latency of a template t executing with a query mix m, its continuum point is defined as:

$$c_{t,m} = \frac{l_{t,m} - l_{min_t}}{l_{max_t} - l_{min_t}}. \quad (6)$$

Contender predicts the continuum point, $c_{t,m}$ by inserting a mix's CQI into its QS model, and then based on Equation 6 estimates the latency of the primary query t, $l_{t,m}$. The following sections describe how this framework constructs a prediction model for $c_{t,m}$.

5.2 Query Sensitivity Models for Known Templates

In Section 4, Table 2, we demonstrated that we can build accurate latency predictions using the CQI metric for known templates. This implies that, given a new query mix m and an established primary query t, we can calculate the CQI metric in Equation 5 for the template t in the mix m and then use a linear regression model to predict the template's performance (i.e., its continuum point). We refer to this linear model as the *Query Sensitivity* (*QS*), because it captures how a template responds to resource contention. This section describes how such a model is built.

In Section 4 we established that there is a linear relationship between CQI and query duration. Therefore, the performance prediction model for template t executed in a query mix m has the form:

$$c_{t,m} = \mu_t \times r_{t,m} + b_t \quad (7)$$

Each QS model consists of an y-intercept (b_t) and a slope (μ_t). Next, we describe how to learn these model coefficients for the known templates in a pre-existing workload. Such examples are *reference models*, and new templates use them to build their own prediction models (see Section 5.3).

To build the QS reference models for a known template t and MPL n, we evaluate sample mixes of n queries including the primary t. We collect these training executions using LHS as outlined in Section 2. For each mix m generated by LHS, we calculate the CQI $r_{t,m}$ and the primary's performance, $l_{t,m}$. We also obtain the isolated latency, l_{min_t} and spoiler latency for the template t, l_{max_t}. Then, based on Equation 6, we evaluate the continuum point $c_{t,m}$. This provides a collection of $(r_{t,m}, c_{t,m})$ pairs based on which we build the QS regression model for the training template t. This model accepts as an input the CQI for a primary query in a given mix and predicts its continuum point. Given the continuum point, we use the isolated and spoiler latencies for the template to scale it into a latency estimate, reversing Equation 6.

5.3 Query Sensitivity Models for New Templates

Building QS models for known templates is a simple way to predict latency for queries executing under concurrency based on the mix's CQI. Contender also learns the QS model (i.e., μ_t and b_t) for a *new* template, by comparing it to ones we have already sampled under concurrency. Contender addresses this challenge for ad-hoc queries, setting it apart from existing CQPP approaches [1, 8]. Next we discuss our approach.

Coefficient Relationship We first examined the relationship between μ_t and b_t for each template in a workload. The study in Figure 4 assembles pairs of the coefficients from QS models generated for MPL 2, applying linear regression to the set. This graph shows the linear relationship between the coefficients.

The y-intercept (b_t) represents the minimum performance of the primary template t under concurrency. The results show that even when the concurrent queries have an I/O usage of near zero ($r_{t,m} = 0$) the primary query still experiences some slow down due

[1]Based on the American colloquialism "something produced to compete with something else and make it less successful"

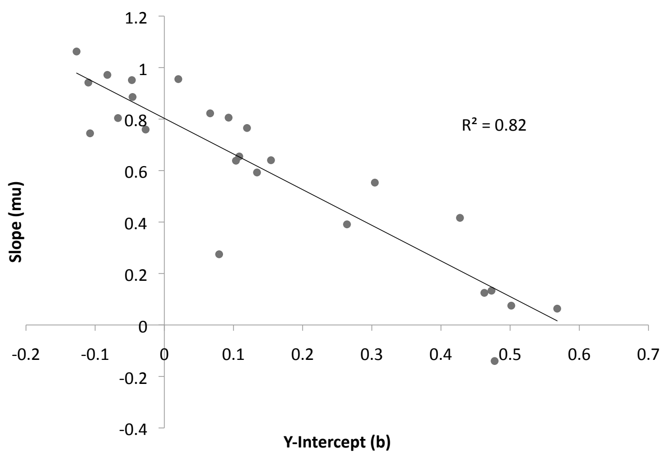


Figure 4: Linear relationship between QS coefficients.

to fixed costs of concurrency such as reduced memory access. Furthermore, the y-intercept may be also negative. This corresponds to query mixes having a positive impact on the performance of the primary, leading to faster execution times than it executing in isolation. These are mostly queries that share the bulk of their I/O work with concurrent queries and have lightweight CPU requirements.

The model slope (μ_t) denotes how quickly the template responds to changes to resource availability. A higher slope indicates that the primary's performance is more sensitive to variations in the I/O requirements of its concurrent queries. These are typically I/O-bound queries executing sequential scans having small intermediate results.

The figure demonstrates that for the majority of our queries the y-intercept and slope are highly correlated (they lie closely along the trend line). This demonstrates that *we can learn the the performance prediction model of a new template (i.e., its QS model) by only estimating one of these parameters and without collecting sample executions of the query with the preexisting workload.* We focus on this in the following paragraph.

Learning QS Coefficients Next, we identify template features for predicting the coefficients of the QS model for previously unseen templates. We examined both performance features, such as the percent of time spent executing I/O in isolation, p_t, and maximum working set size, the size of the largest intermediate result. We also considered query complexity features, by examining how closely the number of plan steps and records accessed correlate with the coefficients. Finally, we studied whether the isolated run time, spoiler latency, and spoiler slowdown (spoiler latency divided by the corresponding isolated query time) are correlated with the QS coefficients.

Our results are charted in Table 3. We use the coefficient of determination (R^2) to assess the goodness of fit between the model and each of the features. R^2 is a normalized measure that relates the error of the model to the overall variance of the data set. It ranges from 0...1, with higher values denoting better fit. Our performance features, I/O time and working set size, were poorly correlated with the QS model parameters. This is not surprising because such indicators are too fine-grained to summarize the query's reaction to concurrency. Likewise, the number of QEP steps and records accessed also yielded too little information about overall query behavior. Spoiler slowdown only captures the worst-case scenario, stripping out valuable information regarding the query as a whole.

We found that isolated latency is inversely correlated with slope of our model. Isolated latency is a useful approximation of the

Query Template Features	Y-Intercept b	Slope μ
% execution time spend on I/O	0.18	-0.05
Max working set	-0.24	0.11
Query plan steps	0.31	-0.29
Records accessed	0.12	-0.22
Isolated latency	0.36	-0.51
Spoiler latency	0.27	-0.49
Spoiler slowdown	0.08	-0.24

Table 3: R^2 for linear regression correlating template features with y-intercept and slope of the QS models.

"weight" of a query. Queries that are lighter tend to have larger slopes. They are more sensitive to changing I/O availability and exhibit greater variance in their latency under different concurrent mixes. In contrast, heavier queries (with long durations) tend to be less perturbed by concurrency; their longer lifespans average out brief concurrency-induced interruptions. Isolated latency also has the highest correlation with the y-intercept. This relationship makes sense, since queries that are I/O-bound (and have a higher y-intercept) tend to be more sensitive to concurrency; those that have a fixed high overhead imply CPU-boundedness. Based on these results, we use isolated latency to predict the QS model parameters for new templates.

Contender Prediction Pipeline This prediction framework for a new templates operates in two phases, shown in Figure 5. First, it trains on a known workload and a given MPL n, i.e., for each known template t, it evaluates the isolated and spoiler latency. We also use LHS to collect limited examples of concurrent query mixes (Step 1) of size n. Based on these mixes, Contender builds its reference QS models for each known template t (Step 2).

Once a new template t' appears, we use linear regression to estimate the slope, $\mu_{t'}$, based on the query's isolated latency, $l_{m_{t'}}$. The input to our linear regression includes the isolated latency of the known templates and the slope (μ) coefficient of our reference QS models. Using this estimated slope, we learn the y-intercept, b'_t, using a second regression step (Step 3).

Having estimated the coefficients of the QS model, we predict t''s continuum point for a mix of concurrent queries m. We do this by first parsing the concurrent queries and calculate the CQI value for the template query t', $r_{t',m}$ (Equation 5, Step 4). We then apply our QS coefficients, $\mu_{t'}$ and $b_{t'}$, to the CQI value to predict the continuum point, $c_{t',m}$, based on the regression model in Equation 7.

Given the continuum point, we use the isolated and spoiler latencies of t' to scale it into a latency estimate, reversing Equation 6. Using this technique, we predict end-to-end latency for new templates using only the template's isolated and spoiler latencies (Step 5), and semantic information from the query execution plans of all members of a mix.

5.4 Reduced Sampling Time

One of the appealing aspects of Contender is that it requires very limited sampling to achieve high-quality predictions. Furthermore, it is a simple and flexible framework for making predictions on *new* templates.

Prior work [8] required significant sampling of unseen templates before making any predictions. This is brittle and incurs a polynomial growth in sampling requirements as the number of distinct workload templates rises. That system used LHS in its training phase for each template. For a workload with t templates with m MPLs and k samples taken at each MPL, this approach necessitates $t \times m \times k$ samples ($O(n^3)$) before it can start making predictions. Incorporating a new template requires executing numerous sample mixes with the previous workload templates in order to collect statistics on their performance. Specifically, that work required at least $2 \times m \times k$ additional samples per template to determine how it interacts with the pre-existing workload. Adding a new template is similarly expensive when using traditional machine learning techniques for producing predictions, as described in Section 3. For instance, the cost of adding a new template for our experiments with KCCA and SVM was 109 hours on average to achieve static workload accuracy levels. Therefore, previous work on performance prediction [8, 10, 11] cannot efficiently handle ad-hoc templates.

In contrast *our approach reduces sampling to linear time*. We only require one sample per MPL, i.e., the spoiler latency. Our experimental study shows that this dramatically reduces our sampling time for known workloads to 23% compared with the cases where we need to sample the spoiler latency for each template. In addition, this approach does not predispose our modeling towards a small subset of randomly selected mixes. Rather, we profile how the individual template responds to concurrency generically. This makes Contender particularly suitable to handle dynamic workloads that include both known and new query templates. In the next section, we introduce a technique to predict the spoiler latency, further reducing the sampling requirements of this framework.

5.5 Predicting Spoiler Latency for New Templates

Contender relies on sampling the primary template performance at every MPL for which it makes prediction. Hence, for each new template, it samples the query once per concurrency level in order to get the spoiler latency. This process can be cumbersome. We address this limitation by learning the spoiler latency from isolated template execution statistics.

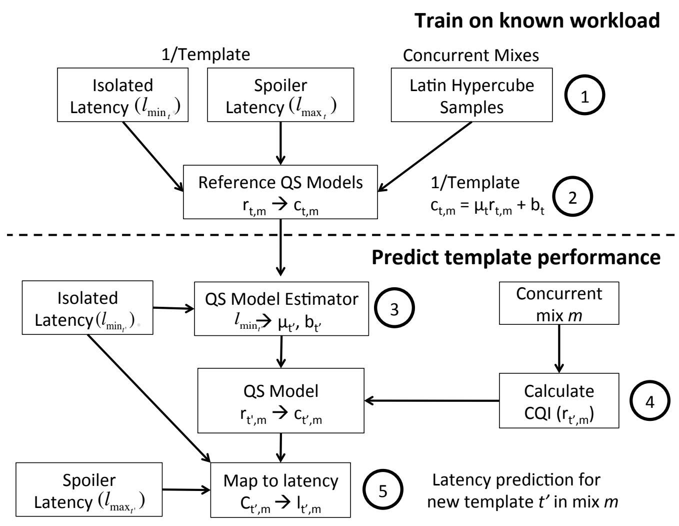
Spoiler-MPL Relationship Our first step is to determine whether template spoiler latencies are predicted based on the concurrency level. We theorized that latency would increase proportionally to our simulated MPL. Qualitatively, we found that templates tend to occupy one of three categories. One example of each is plotted in Figure 6. The first category is demonstrated by Template 62. It is a lightweight, simple template. It has one fact table scan and very small intermediate results. This query is subject to slow growth as contention increases because it is not strictly I/O-bound. In isolation it uses 87% of the I/O bandwidth. The medium category is shown with Template 71. It is I/O-bound, using greater than 99% of its isolated execution time on I/O. T71, however, does not have large intermediate results. Because of these two factors, this template exhibits modest linear growth as we increase the MPL. The final query types are heavy; they are shaped by large intermediate results which necessitate swapping to disk as contention increases. Such queries have a high slope. These templates still exhibit linear increases in latency, however their growth rate is much faster than the other two categories.

In addition, we test this hypothesis using linear regression on each template. The training set consisted of spoiler latencies at MPLs 1-3 and testing was performed on MPLs 4-5. We found that on average we could predict spoiler latency within 8% of the correct elapsed time by using the MPL as the independent variable. Therefore, Contender predicts the spoiler latency for a new template t' at MPL n by using the following linear model:

$$l_{max_{t'}} = \mu_{t'} \times n + b_{t'}. \quad (8)$$

In the next paragraph we discuss how this model is generated for each new template.

Learning Spoiler Growth Models We discovered empirically that spoiler latency for individual templates grows linearly proportional



Train on known workload
1/Template
Concurrent Mixes
Isolated Latency ($l_{\min_t}$)
Spoiler Latency ($l_{\max_t}$)
Latin Hypercube Samples
1
Reference QS Models $r_{t,m} \rightarrow c_{t,m}$
1/Template $c_{t,m} = \mu_t r_{t,m} + b_t$
2
Predict template performance
Isolated Latency ($l_{\min_{t'}}$)
QS Model Estimator $l_{\min_{t'}} \rightarrow \mu_{t'}, b_{t'}$
3
Concurrent mix m
QS Model $r_{t',m} \rightarrow c_{t',m}$
Calculate CQI ($r_{t',m}$)
4
Spoiler Latency ($l_{\max_{t'}}$)
Map to latency $c_{t',m} \rightarrow l_{t',m}$
5
Latency prediction for new template t' in mix m

Figure 5: Contender CQPP framework for new templates.

to the simulated MPL. Individual templates, however, have very different spoiler latencies. This implies that we must learn the coefficients of the linear model ($\mu_{t'}$ and $b_{t'}$) for each new template t' independently. Contender predicts these coefficients by evaluating how comparable t' is to known templates in our workload. In particular, we rely on comparing the I/O behavior of templates. To compare different templates, the model predicts *spoiler latency growth rate* as opposed to spoiler latencies. This figure is calculated by dividing the spoiler latency by the template's isolated duration, producing a scale-independent value. The intuition here is that queries with the same percent of time spent executing I/O have similar rates of slowdown as resources become scarce.

To test this hypothesis, we evaluated whether the percent of execution time spent on I/O, p_t, is correlated with the spoiler latency. We found that p_t corresponds moderately with the model coefficients (Equation 8), having an R^2 of 0.63. Likewise, the working set size indicates the rate at which a template will swap its intermediate results as resources become less available. We also checked the correlation of this metric to the spoiler latency and it had an R^2 of 0.41. Although it is not as well-correlated as I/O rate, this is still a useful indicator and gives us another dimension with which to predict the model's coefficients.

Based on these two metrics (I/O rate and working set size) collected during isolated query execution, we devised a technique for predicting spoiler latency. Specifically, Contender uses a k-nearest neighbors (KNN) approach to learn spoiler model coefficients for the new template based on similar ones. The KNN predictor first projects each known template in a two-dimensional space, with their working set size in one dimension and I/O time, p_t, on the other. For each known template, we have a linear model for predicting their spoiler latency (Equation 8). Next it projects a new template into this space and identifies the k nearest templates based on Euclidean distance. The predictor then averages their model coefficients for the new template's parameters. This way, we learn the spoiler latency patterns of new queries by comparing them directly to members of the prior workload.

This approach learns models for new templates leveraging only isolated performance statistics (i.e., the query's working size and I/O rate when executed in isolation). In doing so, we reduce our training time for new templates from linear (sampling each spoiler at each MPL) to constant (executing the new query only once independently of the MPL).

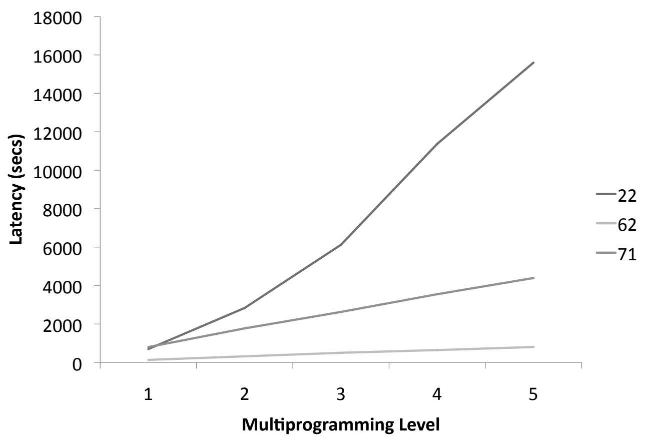


Figure 6: Spoiler latency under increasing concurrency levels.

6. EXPERIMENTAL RESULTS

In this section we evaluate the effectiveness of the Contender framework. We begin by reviewing our experimental setup. After that we evaluate the effectiveness of our CQI metric in predicting query performance, and study the accuracy of our QS model for both known and new templates. Finally, we demonstrate the efficacy of predicting our spoiler latencies to reduce sampling time, and show how this step impacts our overall performance modeling.

6.1 Experimental Setup

In this work we experimented with TPC-DS [17], a decision support benchmark, in its 100 GB configuration. We executed the benchmark on PostgreSQL 8.4.3 on Ubuntu 9.04 using an 8 core Intel i7 2.8 GHz processor and 8 GB of RAM.

Our workload queries have significant variety. Individual templates access between one and three fact tables. Several of our queries are I/O-bound (e.g., templates 26, 33, 61 and 71); templates 17, 25 and 32 execute random I/O, and there are also examples of where the CPU is the limiting factor, such as templates 62 and 65. Contender supports this scenario well because the CPU time is captured in the Query Sensitivity model. Finally, we have a handful of queries that are memory-bound, templates 2 and 22. Memory-intensive queries have significant working set sizes, on the order of several GB. They are the exception in our primarily I/O-bound workload; such templates typically produce higher prediction errors. Nonetheless, we report their prediction accuracy in our results unless otherwise noted.

Our spoiler latency is designed to provide an upper bound for each template's execution time. In a few cases, however, the observed latency exceeds the spoiler's estimate. This occurred when long running queries are paired with very short ones; it is an artifact of our steady state sampling. The cost of restarting, including generating the query plan and re-caching dimension tables, becomes a significant part of its resource consumption. Empirically we found that cases where the query latency is greater than 105% of spoiler latency occurs at a frequency of 4%. Identifying these outliers is possible (e.g., [18]), however it is beyond the scope of our work. We omit them from our evaluation because they measurably exceed the continuum. Addressing this limitation is left to future work.

6.2 Concurrent Query Intensity

First we evaluate the accuracy of using CQI to predict query latency. We build one model per template per MPL. For a given template (the primary), our training set consists of sampled mixes for a given MPL that include this template. For each mix we collect the CQI with its resulting continuum point (the primary's performance in that mix) and generate a linear regression model for predicting the query latency. Figure 7 shows the prediction error for the models of the 25 workload templates at MPL 4. On average we predict latency within 19% of the correct value, demonstrating a well-fitted model for these complex query interactions. The prediction accuracy was similar for MPL levels of 3 and 5. We omit their graphs due to space limitations.

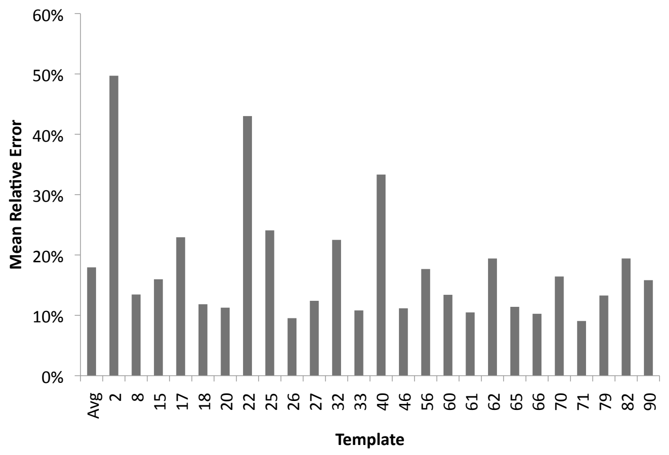


Figure 7: Prediction error rate at MPL 4 with CQI-only model.

Our results also reveal that we can predict the performance of extremely I/O-bound queries very well, within 10% of the correct latency. Recall that templates 26, 33, 61 and 71 are examples of such queries. They spend 97% or more of their execution time in I/O operations in isolation. The results demonstrate that CQI modeling is particularly suited for these queries.

The figure also shows that our prediction error is around 23% for queries that execute random I/O operations (i.e., index scans). Random seeks cause their performance under concurrency to exhibit higher variance. This is because I/O time under concurrency is sensitive to the speed at which the disk completes seeks–previous research shows that random I/O can vary by up to an order of magnitude per page fetched [8].

This workload also includes memory-intensive queries, 2 and 22, for which our prediction error was higher. These templates have large intermediate results that necessitate swapping to disk as contention grows; they demonstrate a different performance profile under limited I/O availability. We did not have enough similar queries in our workload to train such models, so we continue to use the less accurate linear modeling for such templates.

Also, we had a slightly higher error rate for Template 40. This template had two outlier samples in which it ran with queries that both had high memory and were primarily random I/O. Both of these factors reduced the prediction accuracy as discussed above. Without the two outliers, its relative error is reduced to 21%.

6.3 Query Sensitivity

In this section we study the accuracy of our performance prediction model (i.e., QS model) when applied on both known and unseen templates. Figure 8 shows our results for MPLs 2-5.

Known Templates In Figure 8, *Known-Templates* represents predictions for templates which have been sampled under concurrency. Here, we already know the QS model for the query template and

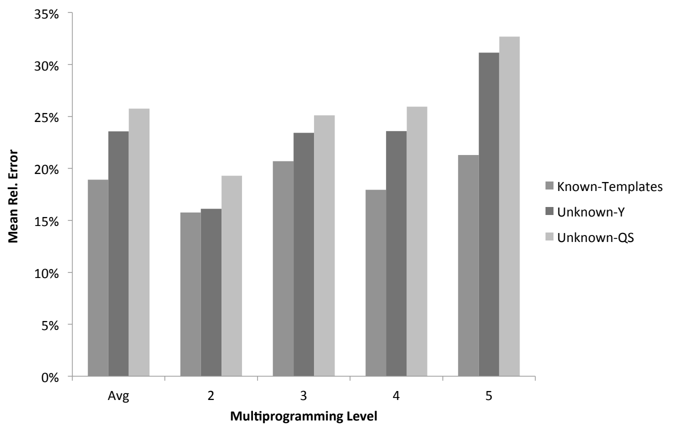


Figure 8: Latency MRE for known and unknown templates.

hence, we only need to evaluate its CQI metric for a given mix to produce a performance prediction. The QS model works very well for known templates having a prediction error of 19% on average, showcasing the effectiveness of this model. They also demonstrate that the CQI metric is robust to complex and changing concurrent execution conditions owing to the variety of this workload.

New Templates Next, we evaluate our prediction accuracy when working with unknown templates. Using the k-fold cross validation ($k = 5$), we train on 20 TPC-DS templates and estimate the performance of 5 unknown ones. Here, we evaluate the prediction error of the Contender approach that learns the entire QS model of new templates (*Unknown-QS*) from reference QS models as described in Section 5.3. Once a new template t' appears, this approach uses a linear regression model to estimate the slope $\mu_{t'}$ based on the query's isolated latency $l_{m,t'}$. The input to this linear regression includes the isolated latencies and slope ($\mu_{t'}$) coefficient of our reference QS models. It uses the learned $\mu_{t'}$ to predict the y-intercept $b_{t'}$ for the new template using a second regression step.

We compare Contender with the *Unknown-Y* approach. Here we start with a known set of QS models which we build for all templates as described in Section 5.2 (i.e., we learn the relationship between CQI and query latency). Once a new template t' appears, we get the $\mu_{t'}$ coefficient from the QS model on t' we produced. We still learn the the $b_{t'}$ coefficient using a second linear regression model, similarly to the Unknown-QS experiment.

The results in Figure 8 demonstrate the accuracy of all approaches. Unknown-Y has an average error of 23% while our approach (Unknown-QS) has an average error of 25%. This is because (as shown in Figure 4) the y-intercept and the slope are not perfectly correlated. Unknown-Y reflects a close linear relationship between the two coefficients demonstrated in Figure 4.

Prior Work Comparison Our accuracy for predicting latency for known and unknown templates is an improvement over the state-of-the-art on CQPP for analytical workloads [8]. Prior work predicted query latency with 25% prediction error on average but required LHS to collect examples of query mixes for each new template (i.e., therefore their 25% accuracy is for known templates). Given this limitation, that work is not fit to provide predictions for new, never before trained upon templates. We achieved approximately the same accuracy for unknown templates and better accuracy (only 19% prediction error) for known templates using linear time samplings. Moreover, our training does not require any sampling of concurrent mixes.

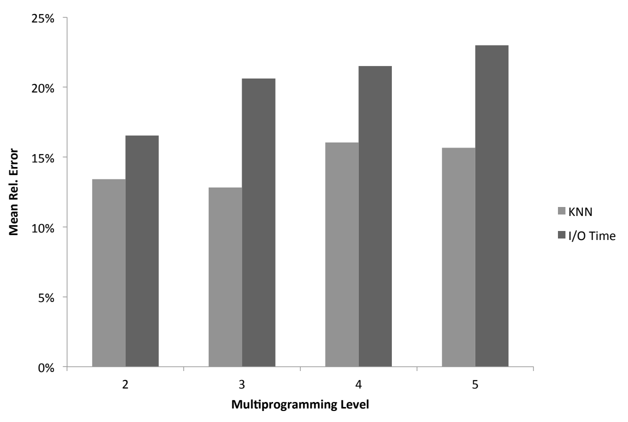


Figure 9: Spoiler prediction for new templates.

6.4 Spoiler Latency Prediction

This section gauges the effectiveness of our spoiler prediction model, which estimates the worst-case performance for a new template at a given MPL by leveraging the spoiler latencies of known templates. This approach reduces our training time for new templates from linear to constant. Contender's approach (*KNN*) predicts the spoiler latency of such templates based on the k nearest neighbors as described in Section 5.5. The closest neighbors are selected based on the working set size and I/O time of the queries when executed in isolation. Our experiments have a k of 3 for neighbor selection.

We compare our proposed technique with the *I/O Time* approach that predicts the spoiler latency using linear regression models. Specifically, this baseline builds on the observation that queries with the same percent of time spent on I/O when executing in isolation (p_t) have similar spoiler growth rates as the I/O bandwidth becomes scarce. We learn the two coefficients ($\mu_{t'}$, $b_{t'}$ in Equation 8) for the new template t' by building a linear regression models for each. Both models have $p_{t'}$ as their independent variable. In this experiment, we train on all but one template and evaluate our model on the excluded one. Both techniques rely on statistics collected only from *isolated executions* of the new template.

Figure 9 shows the prediction error for these techniques on different MPLs. In all cases, Contender's KNN approach delivers higher prediction accuracy. The prediction error is approximately 15% while the I/O Time has an error rate of 20% in average. The main advantage of our approach is that it employs two statistical metrics (i.e., working set size and I/O time) as opposed to a single indicator of the I/O Time approach; it is able to better characterize the worst-case performance of the new template based on similar previous queries.

6.5 Performance Prediction for New Templates

Lastly, we evaluate the accuracy of our overall framework when predicting latencies for new templates. In these experiments, Contender is using the QS model *and* the KNN technique to predict spoiler latency (as opposed to results discussed in Section 6.3 where the spoiler latency was known in advance). The experiments are shown in Figure 10. Here, we average over all templates except T2, the most memory-intensive query. We had too few points of comparison to build a robust model of its spoiler latency growth, however, using linear time sampling, Contender makes accurate predictions for the template.

In these experiments we train on all templates but one and test

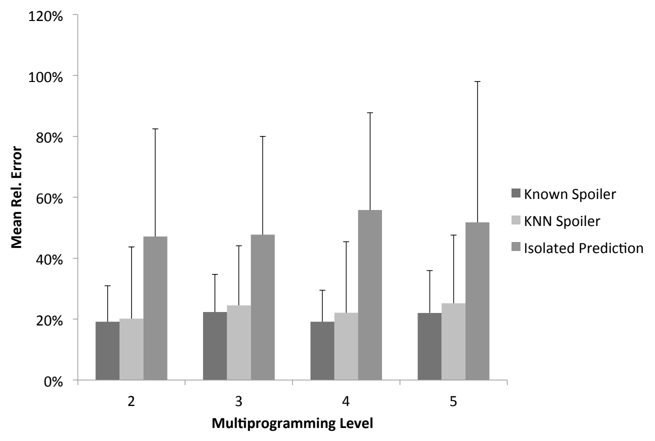


Figure 10: Latency prediction for new templates with standard deviations.

our model on the excluded one. We compare the prediction error of three approaches. *Known Spoiler* uses the QS model but the spoiler latency is measured empirically. It requires a linear time sampling of query executions. *Predicted Spoiler* uses the same CQPP models as the prior experiments, however the spoiler latency is predicted based on the KNN method of Section 5.5. This approach requires a single execution of the query in isolation. Finally, *Isolated Prediction* predicts the inputs for our models (working set size, latency, and I/O time) of a new template using a simulation of the prediction model proposed in [11]. These statistics are then provided to the KNN spoiler prediction technique and QS framework. This evaluation introduces a randomized error rate of $\pm 25\%$ to the isolated template execution statistics and is congruent with the finding in [11], but requires zero sample executions of the new template.

The results demonstrate that Contender's prediction error (which is represented by the KNN Spoiler approach) is 25% for MPL values of 2-5 and it is slightly higher than the prediction error when the spoiler latency is not predicted. In most cases the spoiler latency predictions were sufficiently close such that it did not significantly impact our ability to predict template latency in individual mixes. One side effect of predicting spoiler latency, as the figure shows, is that the standard deviation of our predictions increases. This is because our model becomes less resilient to high-stress mixes as we predict spoiler latencies from the KNN technique. Finally, the Isolated Prediction has the highest prediction error, proportional to its less accurate model sources. This demonstrates that one sample execution of the query in isolation (as Contender does) is more effective than relying on the existing prediction models for input.

The above experimental results clearly state Contender offers prediction errors comparable to the state-of-art in latency prediction for known templates without concurrency [11] and with concurrency [8]. Therefore, our framework offers a significant improvement over existing techniques as it can deliver comparable prediction accuracy on more complex scenarios (i.e., unknown templates executed concurrently) with significantly less overhead (i.e., constant sampling as opposed to exponential/linear).

7. RELATED WORK

There has been significant work in the field of query performance prediction and progress indicators. In this section, we outline the main advancements to date.

Query Progress Indicators Query progress indicators have been proposed in [19, 20, 21] and established solutions are surveyed in [22]. In [19] the authors model the percent of the query completed, however their solution does not directly address latency predictions for database queries and they do not consider concurrent workloads. [20, 21] take into account system strain and concurrency, but they focus on performance of queries in progress. In contrast, we predict the latency of queries *before* they begin.

Query Performance Prediction In [10, 11, 23, 24] the researchers use machine learning techniques to predict the performance of analytical queries. Although they predict query latency, they do not address concurrent workloads. In Section 3 we experimented with the same machine learning techniques and found our adaptation unsuitable for CQPP.

Query performance prediction under concurrency was first explored in [3, 18, 25]. The authors create concurrency-aware models to build schedules for batches of OLAP queries. Their solutions create regression models based on sampled query mixes. These systems generate optimal schedules for a set of OLAP queries to minimize end-to-end batch duration. [2, 4] extends the work in [3, 18] to predict the completion time of mixes of concurrent queries over time from given query batches. This work does not target individual query latencies either; instead the authors predict end-to-end workload latencies. Their models are designed for known templates. Our framework, however, provides finer-grained predictions that estimate the response time of *individual* queries. In [8], we proposed predictive latency models for concurrently executing queries. Although Contender also forecasts query duration, it handles new templates and does not require sampling how they interact with the preexisting workload.

An alternative approach was explored in [26]; the authors predicted the performance of queries that are already running. [7, 27] researched workload modeling for transactional databases (OLTP), however their solutions are tailored to the buffer management techniques of MySQL. [28] makes prediction for the throughput of analytical workloads on varying hardware. In [12, 29], the authors explored workload modeling under concurrency. [29] makes predictions about query latency under concurrency as a range, and [12] examines high-level workloads. None of these approaches make as precise query latency predictions as we do in this work.

8. CONCLUSIONS

We studied *Concurrent Query Performance Prediction* (CQPP) for dynamic, analytical workloads. This problem has many important applications in resource scheduling, cloud provisioning, and user experience management. This work demonstrated that the prior machine learning approaches for query performance prediction do not provide satisfactory solutions when extended for concurrency. The same is true for dynamic workloads that include new query templates due to their high sampling requirements.

We propose a novel framework for CQPP named *Contender*; it models the degree to which queries create and are affected by resource contention and is targeted for I/O-bound workloads. This approach uses a combination of semantic information and empirical evaluation to build a model for each template at different concurrency levels. In doing so, we show significant improvements over the black-box machine learning techniques.

We formally define Concurrent Query Intensity (CQI), a novel metric that quantifies I/O contention and demonstrate that is effective for predicting query latency. We then introduce our Query Sensitivity (QS) predictor that makes accurate predictions for known *and* arbitrary queries with low training overhead. QS relies on generalizing from similar queries in the preexisting workload to build its models. Its parameters convey how responsive a template is to changing availability of shared resources.

Our experimental results showed our predictions are within approximately 25% of the correct value using linear time sampling for new queries, and 19% for known templates. Our approach is thus competitive with alternative techniques in terms of predictive accuracy, yet it constitutes a substantial improvement over the state of the art, as it is both more practical and efficient (i.e., requires less training).

In future work, we would like to explore CQPP at the granularity of individual query execution plan nodes. This would make our models more flexible and finer-grained, however, it necessitates sophisticated sampling strategies to model arbitrary interactions among operators. The work of [24] has made the case for this promising line of research.

Another interesting direction for this work is developing models for predicting query performance on an expanding database. As database writes accumulate, this would enable the predictor to continue to provide important information to database users.

An additional open question is that of modeling interactions for distributed analytical workloads. Distributed query plans call for modeling their sub-plans as they are assigned to individual hosts as well as the time associated with assembling intermediate results. This would also require incorporating the cost of network traffic and coordination overhead.

9. ACKNOWLEDGEMENTS

The authors would like to thank Jason Pacheco and Mert Akdere for their advice on machine learning algorithms. This work was funded by NSF Grants IIS-0905553 and IIS-0916691.

10. REFERENCES

[1] M. Ahmad, A. Aboulnaga, S. Babu, and K. Munagala, "Interaction-aware scheduling of report-generation workloads," *VLDB J.*, vol. 20, no. 4, pp. 589–615, 2011.

[2] M. Ahmad, S. Duan, A. Aboulnaga, and S. Babu, "Predicting completion times of batch query workloads using interaction-aware models and simulation," in *EDBT*, 2011.

[3] M. Ahmad, A. Aboulnaga, S. Babu, and K. Munagala, "Modeling and exploiting query interactions in database systems," in *CIKM*, 2008.

[4] A. Ahmad, S. Duan, A. Aboulnaga, and S. Babu, "Interaction-aware prediction of business intelligence workload completion times," *ICDE '10*.

[5] J. Rogers, O. Papaemmanouil, and U. Cetintemel, "A generic auto-provisioning framework for cloud databases," in *SMDB '10*.

[6] A. Aboulnaga, K. Salem, A. A. Soror, U. F. Minhas, P. Kokosielis, and S. Kamath, "Deploying database appliances in the cloud," *IEEE Data Eng. Bull.*, vol. 32, no. 1, pp. 13–20, 2009.

[7] B. Mozafari, C. Curino, A. Jindal, and S. Madden, "Performance and resource modeling in highly-concurrent oltp workloads," *SIGMOD'13*.

[8] J. Duggan, U. Cetintemel, O. Papaemmanouil, and E. Upfal, "Performance prediction for concurrent database workloads," *SIGMOD '11*.

[9] W. Hsu, A. J. Smith, and H. Young, "I/o reference behavior of production database workloads and the tpc benchmarks - an analysis at the logical level," *ACM TODS*, vol. 26, no. 1, Mar. 2001.

[10] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson, "Predicting multiple metrics for queries: Better decisions enabled by machine learning," in *ICDE '09*.

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

[12] A. Mehta, C. Gupta, and U. Dayal, "BI batch manager: a system for managing batch workloads on enterprise data-warehouses," *EDBT '08*.

[13] M. Poess, R. O. Nambiar, and D. Walrath, "Why you should run tpc-ds: a workload analysis," in *VLDB '07*.

[14] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Trans. on Intelligent Systems and Tech.*, vol. 2, pp. 27:1–27:27, 2011.

[15] A. Karatzoglou, A. Smola, K. Hornik, and A. Zeileis, "kernlab – an S4 package for kernel methods in R," *J. of Statistical Software*, 2004.

[16] R Development Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, 2008.

[17] M. Poess, B. Smith, L. Kollar, and P. Larson, "Tpc-ds, taking decision support benchmarking to the next level," in *SIGMOD '02*.

[18] M. Ahmad, A. Aboulnaga, S. Babu, and K. Munagala, "Qshuffler: Getting the query mix right," *ICDE '08*.

[19] S. Chaudhuri, V. Narasayya, and R. Ramamurthy, "Estimating progress of execution for sql queries," in *SIGMOD '04*.

[20] G. Luo, J. Naughton, and P. Yu, "Multi-query sql progress indicators," in *EDBT 2006*.

[21] G. Luo, J. F. Naughton, C. J. Ellmann, and M. W. Watzke, "Toward a progress indicator for database queries," in *SIGMOD '04*.

[22] S. Chaudhuri, R. Kaushik, and R. Ramamurthy, "When can we trust progress estimators for SQL queries?" in *SIGMOD '05*.

[23] W. Wu, Y. Chi, S. Zhu, J. Tatemura, H. Hacigumus, and J. Naughton, "Predicting query execution time: Are optimizer cost models really unusable?" *ICDE'13*.

[24] W. Wu, Y. Chi, H. Hacigümüs, and J. F. Naughton, "Towards predicting query execution time for concurrent and dynamic database workloads," *PVLDB*, vol. 6, no. 10, pp. 925–936, 2013.

[25] M. Ahmad, A. Aboulnaga, and S. Babu, "Query interactions in database workloads," in *DBTest*, 2009.

[26] M. B. Sheikh, U. F. Minhas, O. Z. Khan, A. Aboulnaga, P. Poupart, and D. J. Taylor, "A bayesian approach to online performance modeling for database appliances using gaussian models," *IEEE ICAC '11*.

[27] B. Mozafari, C. Curino, and S. Madden, "Dbseer: Resource and performance prediction for building a next generation database cloud," *CIDR'13*.

[28] J. Duggan, Y. Chi, H. Hacigumus, S. Zhu, and U. Cetintemel, "Packing light: Portable workload performance prediction for the cloud," *DMC '13*.

[29] C. Gupta, A. Mehta, and U. Dayal, "PQR: Predicting Query Execution Times for Autonomous Workload Management," *IEEE ICAC '08*.